

Java 2

Curso de programación

Windows 9x/NT/2000, Macintosh y Unix (Linux, Solaris y otros)

- Programación orientada a objetos
- Elementos del lenguaje
- Sentencias de control
- Clases de uso común
- Matrices y cadenas
- Clases, subclases, paquetes e interfaces

- Excepciones
- Ficheros
- Estructuras dinámicas
- Algoritmos
- Hilos
- Internet, HTML, Applets, Swing, Servlets



Fco. Javier Ceballos



Incluye CD-ROM
con el JDK y
las aplicaciones
contenidas en el libro



Ra-Ma®

RESUMEN DEL CONTENIDO

PARTE 1. PROGRAMACIÓN BÁSICA	1
CAPÍTULO 1. FASES EN EL DESARROLLO DE UN PROGRAMA	3
CAPÍTULO 2. PROGRAMACIÓN ORIENTADA A OBJETOS	23
CAPÍTULO 3. ELEMENTOS DEL LENGUAJE	37
CAPÍTULO 4. ESTRUCTURA DE UN PROGRAMA	63
CAPÍTULO 5. CLASES DE USO COMÚN	89
CAPÍTULO 6. SENTENCIAS DE CONTROL	121
CAPÍTULO 7. MATRICES	163
CAPÍTULO 8. MÉTODOS	215
PARTE 2. PROGRAMACIÓN AVANZADA	251
CAPÍTULO 9. CLASES Y PAQUETES	253
CAPÍTULO 10. SUBCLASES E INTERFACES	329
CAPÍTULO 11. EXCEPCIONES	397
CAPÍTULO 12. TRABAJAR CON FICHEROS	419
CAPÍTULO 13. ESTRUCTURAS DINÁMICAS	495

CAPÍTULO 14. ALGORITMOS	585
CAPÍTULO 15. HILOS	633
PARTE 3. PROGRAMAS PARA INTERNET	685
CAPÍTULO 16. A PARTIR DE AQUÍ	687
PARTE 4. APÉNDICES	751
A. AYUDA.....	753
B. JAVA COMPARADO CON C/C++	755
C. PLATAFORMAS UNIX/LINUX	759
D. CONTENIDO DEL CD-ROM	761
E. CÓDIGOS DE CARACTERES	763
F. ÍNDICE	769

CONTENIDO

PRÓLOGO.....	XXIII
PARTE 1. PROGRAMACIÓN BÁSICA.....	1
CAPÍTULO 1. FASES EN EL DESARROLLO DE UN PROGRAMA	3
QUÉ ES UN PROGRAMA.....	3
LENGUAJES DE PROGRAMACIÓN	4
Compiladores	6
Intérpretes.....	6
¿QUÉ ES JAVA?	7
HISTORIA DE JAVA.....	8
¿POR QUÉ APRENDER JAVA?	9
REALIZACIÓN DE UN PROGRAMA EN JAVA	9
Cómo crear un programa	11
Interfaz de línea de órdenes	12
¿Qué hace este programa?.....	12
Guardar el programa escrito en el disco	13
Compilar y ejecutar el programa	13
Biblioteca de funciones	15
Guardar el programa ejecutable en el disco.....	15
Depurar un programa	16
Entorno de desarrollo integrado	16
EJERCICIOS RESUELTOS	19
EJERCICIOS PROPUESTOS	21

CAPÍTULO 2. PROGRAMACIÓN ORIENTADA A OBJETOS.....	23
MECANISMOS BÁSICOS DE LA POO	24
Objetos	24
Mensajes.....	24
Métodos.....	24
Clases	25
CÓMO CREAR UNA CLASE DE OBJETOS	26
CARACTERÍSTICAS DE LA POO	32
Abstracción.....	32
Encapsulamiento	33
Herencia	33
Polimorfismo.....	34
CONSTRUCTORES Y DESTRUCTORES.....	34
EJERCICIOS RESUELTOS	34
EJERCICIOS PROPUESTOS.....	36
CAPÍTULO 3. ELEMENTOS DEL LENGUAJE	37
PRESENTACIÓN DE LA SINTAXIS DE JAVA	37
CARACTERES DE JAVA.....	38
Letras, dígitos y otros	38
Espacios en blanco	38
Caracteres especiales y signos de puntuación	39
Secuencias de escape.....	39
TIPOS DE DATOS	40
Tipos primitivos	40
byte.....	41
short.....	41
int	41
long	41
char.....	42
float	42
double	43
boolean	43
Tipos referenciados	43
LITERALES.....	43
Literales enteros	44
Literales reales.....	44
Literales de un solo carácter	45
Literales de cadenas de caracteres	45
IDENTIFICADORES.....	46

PALABRAS CLAVE	47
COMENTARIOS	47
DECLARACIÓN DE CONSTANTES SIMBÓLICAS	48
¿Por qué utilizar constantes?	49
DECLARACIÓN DE UNA VARIABLE.....	49
Iniciación de una variable.....	50
EXPRESIONES NUMÉRICAS	51
CONVERSIÓN ENTRE TIPOS DE DATOS	51
OPERADORES	52
Operadores aritméticos	52
Operadores de relación	53
Operadores lógicos	54
Operadores unitarios.....	55
Operadores a nivel de bits	55
Operadores de asignación.....	56
Operador condicional	57
PRIORIDAD Y ORDEN DE EVALUACIÓN	58
EJERCICIOS RESUELTOS	58
EJERCICIOS PROPUESTOS	60
 CAPÍTULO 4. ESTRUCTURA DE UN PROGRAMA.....	 63
ESTRUCTURA DE UNA APLICACIÓN JAVA	63
Paquetes y protección de clases.....	67
Protección de una clase	68
Sentencia import.....	69
Definiciones y declaraciones	70
Sentencia simple.....	71
Sentencia compuesta o bloque.....	72
Métodos	72
Definición de un método	72
Método main	73
Crear objetos de una clase	73
Cómo acceder a los miembros de un objeto	75
Protección de los miembros de una clase	76
Miembro de un objeto o de una clase	77
Referencias a objetos	79
Pasando argumentos a los métodos	82
PROGRAMA JAVA FORMADO POR MÚLTIPLES FICHEROS	83
ACCESIBILIDAD DE VARIABLES	85
EJERCICIOS RESUELTOS	86
EJERCICIOS PROPUESTOS	88

CAPÍTULO 5. CLASES DE USO COMÚN.....	89
DATOS NUMÉRICOS Y CADENAS DE CARACTERES.....	89
ENTRADA Y SALIDA	91
Flujos de entrada	93
Flujos de salida.....	94
Excepciones.....	95
Flujos estándar de E/S	96
Determinar la clase a la que pertenece un objeto	97
BufferedInputStream.....	98
BufferedReader	99
PrintStream.....	100
Trabajar con tipos de datos primitivos	102
Clases que encapsulan los tipos primitivos	103
Clase Leer.....	106
¿DÓNDE SE UBICAN LAS CLASES QUE DAN SOPORTE?.....	109
Variable CLASSPATH	110
CARÁCTER FIN DE FICHERO	110
CARACTERES \r\n.....	112
MÉTODOS MATEMÁTICOS.....	114
EJERCICIOS RESUELTOS	116
EJERCICIOS PROPUESTOS	119
CAPÍTULO 6. SENTENCIAS DE CONTROL.....	121
SENTENCIA if	121
ANIDAMIENTO DE SENTENCIAS if.....	124
ESTRUCTURA else if	126
SENTENCIA switch	129
SENTENCIA while	133
Bucles anidados.....	136
SENTENCIA do ... while	139
SENTENCIA for	142
SENTENCIA break	146
SENTENCIA continue	146
ETIQUETAS	147
SENTENCIAS try ... catch	148
EJERCICIOS RESUELTOS	149
EJERCICIOS PROPUESTOS	159

CAPÍTULO 7. MATRICES.....	163
INTRODUCCIÓN A LAS MATRICES	164
MATRICES NUMÉRICAS UNIDIMENSIONALES	165
Declarar una matriz	165
Crear una matriz	166
Iniciar una matriz	167
Acceder a los elementos de una matriz.....	167
Métodos de una matriz	168
Trabajar con matrices unidimensionales	169
Matrices asociativas	172
CADENAS DE CARACTERES	175
Leer y escribir una cadena de caracteres	176
Trabajar con cadenas de caracteres	178
Clase String	181
String(String valor)	181
String toString()	182
String concat(String str)	182
int compareTo(String otroString).....	182
int length().....	184
String toLowerCase()	184
String toUpperCase().....	184
String trim().....	184
boolean startsWith(String prefijo).....	184
boolean endsWith(String sufijo)	184
String substring(int IndiceInicial, int IndiceFinal)	185
char charAt(int índice)	185
int indexOf(int car).....	185
int indexOf(String str)	185
String replace(char car, char nuevoCar).....	185
static String valueOf(tipo dato)	186
char[] toCharArray()	186
byte[] getBytes().....	186
Clase StringBuffer	186
StringBuffer([arg])	186
int length().....	187
int capacity().....	187
StringBuffer append(tipo x)	187
StringBuffer insert(int índice, tipo x)	187
StringBuffer delete(int p1, int p2)	188
StringBuffer replace(int p1, int p2, String str)	188
StringBuffer reverse()	188
String substring(int IndiceInicial, int IndiceFinal)	189

char charAt(int índice)	189
void setCharAt(int índice, char car)	189
String toString()	189
Clase StringTokenizer	189
Conversión de cadenas de caracteres a datos numéricos	191
MATRICES DE REFERENCIAS A OBJETOS	191
Matrices numéricas multidimensionales.....	192
Matrices de cadenas de caracteres	196
Matrices de objetos String	203
EJERCICIOS RESUELTOS	205
EJERCICIOS PROPUESTOS	210
CAPÍTULO 8. MÉTODOS.....	215
PASAR UNA MATRIZ COMO ARGUMENTO A UN MÉTODO.....	215
MATRIZ COMO VALOR RETORNADO POR UN MÉTODO	217
REFERENCIA A UN TIPO PRIMITIVO	219
ARGUMENTOS EN LA LÍNEA DE ÓRDENES	221
MÉTODOS RECURSIVOS	224
VISUALIZAR DATOS CON FORMATO	225
Dar formato a números	226
Localidad.....	227
Alineación	228
Clase para formatos numéricos	229
Dar formato a fechas/horas.....	231
Dar formato a mensajes	233
LA CLASE Arrays	233
binarySearch.....	233
equals.....	234
fill	234
sort.....	235
LA CLASE Object	235
boolean equals(Object obj).....	236
String toString().....	237
void finalize()	237
MÁS SOBRE REFERENCIAS Y OBJETOS String	238
EJERCICIOS RESUELTOS	240
EJERCICIOS PROPUESTOS	247

PARTE 2. PROGRAMACIÓN AVANZADA	251
CAPÍTULO 9. CLASES Y PAQUETES	253
DEFINICIÓN DE UNA CLASE.....	253
Atributos.....	255
Métodos de una clase	256
Control de acceso a los miembros de la clase.....	257
Acceso predeterminado	257
Acceso público	258
Acceso privado.....	258
Acceso protegido.....	258
IMPLEMENTACIÓN DE UNA CLASE.....	259
MÉTODOS SOBRECARGADOS	262
IMPLEMENTACIÓN DE UNA APLICACIÓN	264
CONTROL DE ACCESO A UNA CLASE	265
REFERENCIA this.....	266
VARIABLES, MÉTODOS Y CLASES FINALES	267
INICIACIÓN DE UN OBJETO	269
Constructor	269
Sobrecarga del constructor	272
Llamar a un constructor.....	274
Asignación de objetos	274
Constructor copia	275
DESTRUCCIÓN DE OBJETOS.....	276
Destructor	277
Ejecutar el recolector de basura.....	279
REFERENCIAS COMO MIEMBROS DE UNA CLASE.....	279
COMPARAR OBJETOS.....	287
Método equals	287
MIEMBROS STATIC DE UNA CLASE	288
Atributos static	289
Acceder a los atributos static	290
Métodos static	291
Iniciador estático	292
MATRICES DE OBJETOS.....	294
PAQUETES	303
Crear un paquete.....	304
UN EJEMPLO DE DISEÑO DE UNA CLASE	306
EJERCICIOS RESUELTOS	319
EJERCICIOS PROPUESTOS	326

CAPÍTULO 10. SUBCLASES E INTERFACES.....	329
CLASES Y MÉTODOS ABSTRACTOS	330
SUBCLASES Y HERENCIA	331
DEFINIR UNA SUBCLASE	334
Control de acceso a los miembros de las clases.....	336
Qué miembros hereda una subclase.....	336
ATRIBUTOS CON EL MISMO NOMBRE	341
REDEFINIR MÉTODOS DE LA SUPERCLASE	343
CONSTRUCTORES DE LAS SUBCLASES	345
DESTRUCTORES DE LAS SUBCLASES	347
JERARQUÍA DE CLASES	349
REFERENCIAS A OBJETOS DE UNA SUBCLASE	356
Conversiones implícitas.....	357
Conversiones explícitas.....	359
POLIMORFISMO	360
MÉTODOS EN LÍNEA	370
INTERFACES.....	371
Definir una interfaz	371
Un ejemplo: la interfaz IFecha	372
Utilizar una interfaz	374
Clase abstracta frente a interfaz	377
Utilizar una interfaz como un tipo	378
Interfaces frente a herencia múltiple	379
Para qué sirve una interfaz	380
Implementar múltiples interfaces	380
CLASES ANIDADAS	381
Clases internas	382
Clases definidas dentro de un método	383
Clases anónimas	384
EJERCICIOS RESUELtos	386
EJERCICIOS PROPUESTOS	396
CAPÍTULO 11. EXCEPCIONES	397
EXCEPCIONES DE JAVA.....	399
MANEJAR EXCEPCIONES	401
Lanzar una excepción	402
Atrapar una excepción	402
BLOQUE DE FINALIZACIÓN	405
DECLARAR EXCEPCIONES.....	406
CREAR Y LANZAR EXCEPCIONES.....	408

CUÁNDΟ UTILIZAR EXCEPCIONES Y CUÁNDΟ NO	413
EJERCICIOS RESUELTOΣ	413
EJERCICIOS PROPUESTOΣ	418
CAPÍTULO 12. TRABAJAR CON FICHEROS	419
VISIÓN GENERAL DE LOS FLUJOS DE E/S	421
Flujos que no procesan los datos de E/S.....	422
Flujos que procesan los datos de E/S.....	424
ABRIENDO FICHEROS PARA ACCESO SECUENCIAL	429
Flujos de bytes.....	430
FileOutputStream	430
InputStream.....	433
Clase File.....	434
Flujos de caracteres	437
Writer	438
Reader	439
Flujos de datos.....	440
DataOutputStream	441
DataInputStream	442
Un ejemplo de acceso secuencial	443
SERIACIÓN DE OBJETOS	448
Escribir objetos en un fichero.....	450
Leer objetos desde un fichero.....	451
Seriar objetos que referencian a objetos	453
ABRIENDO FICHEROS PARA ACCESO ALEATORIO	457
La clase RandomAccessFile	457
La clase CPersona	460
La clase CListaTfnos	461
Constructor CListaTfnos	461
Escribir un registro en el fichero	463
Añadir un registro al final del fichero	464
Leer un registro del fichero	464
Eliminar un registro del fichero.....	465
Buscar un registro en el fichero.....	465
Un ejemplo de acceso aleatorio a un fichero	466
Modificar un registro.....	469
Actualizar el fichero	471
UTILIZACIÓN DE DISPOSITIVOS ESTÁNDAR	472
EJERCICIOS RESUELTOΣ	473
EJERCICIOS PROPUESTOΣ	493

CAPÍTULO 13. ESTRUCTURAS DINÁMICAS	495
LISTAS LINEALES	496
Listas lineales simplemente enlazadas.....	496
Operaciones básicas	499
Inserción de un elemento al comienzo de la lista	500
Inserción de un elemento en general	502
Borrar un elemento de la lista.....	503
Recorrer una lista	504
Borrar todos los elementos de una lista.....	504
Buscar en una lista un elemento con un valor x	504
UNA CLASE PARA LISTAS LINEALES.....	505
Clase genérica para listas lineales	509
Clase LinkedList	519
LISTAS CIRCULARES.....	522
Clase CListaCircularSE.....	523
PILAS.....	527
COLAS.....	529
EJEMPLO	530
LISTA DOBLEMENTE ENLAZADA	533
Lista circular doblemente enlazada	534
Clase CListaCircularDE	534
Ejemplo	540
ÁRBOLES.....	542
Árboles binarios	543
Formas de recorrer un árbol binario	544
ÁRBOLES BINARIOS DE BÚSQUEDA	546
Clase CArbolBinB	547
Buscar un nodo en el árbol	550
Insertar un nodo en el árbol	551
Borrar un nodo del árbol	553
Utilización de la clase CArbolBinB	555
ÁRBOLES BINARIOS PERFECTAMENTE EQUILIBRADOS	558
Clase CArbolBinE	559
Utilización de la clase CArbolBinE.....	564
CLASES RELACIONADAS DE LA BIBLIOTECA JAVA	566
EJERCICIOS RESUELTOS	567
EJERCICIOS PROPUESTOS	581
CAPÍTULO 14. ALGORITMOS	585
RECURSIVIDAD	585

ORDENACIÓN DE DATOS	591
Método de la burbuja.....	592
Método de inserción	595
Método quicksort.....	596
Comparación de los métodos expuestos	600
BÚSQUEDA DE DATOS	600
Búsqueda secuencial.....	600
Búsqueda binaria	601
Búsqueda de cadenas	602
ORDENACIÓN DE FICHEROS EN DISCO	605
Ordenación de ficheros. Acceso secuencial.....	606
Ordenación de ficheros. Acceso aleatorio	614
ALGORITMOS HASH	616
Matrices hash.....	617
Método hash abierto	618
Método hash con desbordamiento	619
Eliminación de elementos.....	620
Clase CHashAbierto	620
Un ejemplo de una matriz hash	624
EJERCICIOS RESUELtos	627
EJERCICIOS PROPUESTOS	631
 CAPÍTULO 15. HILOS	633
 CONCEPTO DE PROCESO	633
HILOS	635
Estados de un hilo	636
Cuándo se debe crear un hilo.....	637
Cómo se crea un hilo	638
Hilo derivado de Thread.....	640
Hilo asociado con una clase	641
Demonios	643
Finalizar un hilo	644
Controlar un hilo	646
Preparado	647
Bloqueado	647
Dormido	648
Esperando	649
Planificación de hilos	649
¿Qué ocurre con los hilos que tengan igual prioridad?.....	650
Asignar prioridades a los hilos	651
SINCRONIZACIÓN DE HILOS	654

Secciones críticas	655
Crear una sección crítica	659
Monitor reentrant.....	662
Utilizar wait y notify	663
¿Por qué los métodos almacenar y obtener utilizan un bucle?	669
Interbloqueo	670
GRUPO DE HILOS	671
Grupo predefinido	671
Grupo explícito.....	673
TUBERÍAS	673
ESPERA ACTIVA Y PASIVA	678
EJERCICIOS RESUELTOS	678
EJERCICIOS PROPUESTOS.....	683
 PARTE 3. PROGRAMAS PARA INTERNET	685
 CAPÍTULO 16. A PARTIR DE AQUÍ	687
 ¿QUÉ ES INTERNET?	688
Intranet	689
Extranet	689
Terminología Internet.....	689
 SERVICIOS EN INTERNET	692
 PÁGINAS WEB	695
Qué es HTML.....	695
Etiquetas básicas HTML	696
Etiquetas de formato de texto	697
URL.....	699
Enlaces entre páginas	699
Gráficos	701
Marcos	702
Otros	703
 PÁGINAS WEB DINÁMICAS	703
 APPLETS	705
Crear un applet	705
La clase Applet.....	707
public void init().....	708
public void start()	708
public void paint(Graphics g).....	708
public void stop()	708
public void destroy()	709
Un ejemplo simple.....	709

Ciclo de vida de un applet	711
Pasar parámetros a un applet	711
Fuentes	713
Color	713
Mostrar una imagen	714
Reproducir un fichero de sonido	716
Mostrar información en la barra de estado	718
Crear una animación	718
Restricciones de seguridad con los applets	722
INTERFAZ GRÁFICA	723
Estructura de una aplicación	724
Crear un componente Swing	726
Manejo de eventos	727
Contenedores	729
Organizar los controles en un contenedor	730
Establecer la apariencia de las ventanas	731
Un ejemplo de una interfaz gráfica	732
Applets que utilizan componentes Swing	734
SERVLETS	737
Estructura de un servlet	738
Software necesario para ejecutar un servlet	742
Ejecutar un servlet	743
Invocando al servlet desde una página HTML	744
EJERCICIOS RESUELTOS	746
EJERCICIOS PROPUESTOS	749
 PARTE 4. APÉNDICES	 751
A. AYUDA	753
B. JAVA COMPARADO CON C/C++	755
C. PLATAFORMAS UNIX/LINUX	759
D. CONTENIDO DEL CD-ROM	761
E. CÓDIGOS DE CARACTERES	763
F. ÍNDICE	769

PART E 1

Programación básica

- Fases en el desarrollo de un programa
- Programación orientada a objetos
- Elementos del lenguaje
- Estructura de un programa
- Clases de uso común
- Sentencias de control
- Matrices
- Métodos

CAPÍTULO 1

© F.J.Ceballos/RA-MA

FASES EN EL DESARROLLO DE UN PROGRAMA

En este capítulo aprenderá lo que es un programa, cómo escribirlo y qué hacer para que el ordenador lo ejecute y muestre los resultados perseguidos. También adquirirá conocimientos generales acerca de los lenguajes de programación utilizados para escribir programas. Después, nos centraremos en un lenguaje de programación específico y objetivo de este libro, *Java*, presentando sus antecedentes y marcando la pauta a seguir para realizar un programa sencillo.

QUÉ ES UN PROGRAMA

Probablemente alguna vez haya utilizado un ordenador para escribir un documento o para divertirse con algún juego. Recuerde que en el caso de escribir un documento, primero tuvo que poner en marcha un procesador de textos, y que si quiso divertirse con un juego, lo primero que tuvo que hacer fue poner en marcha el juego. Tanto el procesador de textos como el juego son *programas* de ordenador.

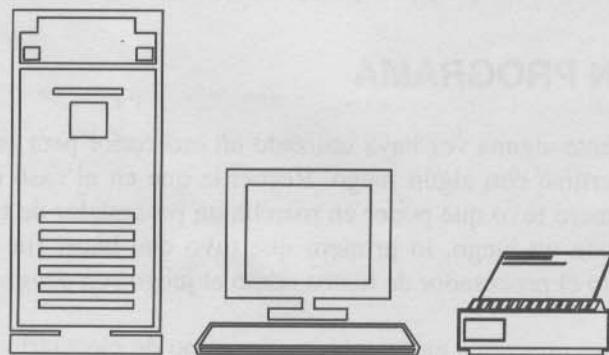
Poner un programa en marcha es sinónimo de ejecutarlo. Cuando ejecutamos un programa, nosotros sólo vemos los resultados que produce (el procesador de textos muestra sobre la pantalla el texto que escribimos; el juego visualiza sobre la pantalla las imágenes que se van sucediendo) pero no vemos el guión seguido por el ordenador para conseguir esos resultados. Ese guión es el programa.

Ahora, si nosotros escribimos un programa, entonces sí que sabemos cómo trabaja y por qué trabaja de esa forma. Esto es una forma muy diferente y curiosa de ver un programa de ordenador, lo cual no tiene nada que ver con la experiencia adquirida en la ejecución de distintos programas.

Ahora, piense en un juego cualquiera. La pregunta es ¿qué hacemos si queremos enseñar a otra persona a jugar? Lógicamente le explicamos lo que debe hacer; esto es, los pasos que tiene que seguir. Dicho de otra forma, le damos instrucciones de cómo debe actuar. Esto es lo que hace un programa de ordenador. Un *programa* no es nada más que una serie de instrucciones dadas al ordenador en un lenguaje entendido por él, para decirle exactamente lo que queremos que haga. Si el ordenador no entiende alguna instrucción, lo comunicará generalmente mediante mensajes visualizados en la pantalla.

LENGUAJES DE PROGRAMACIÓN

Un programa tiene que escribirse en un lenguaje entendible por el ordenador. Desde el punto de vista físico, un ordenador es una máquina electrónica. Los elementos físicos (memoria, unidad central de proceso, etc.) de que dispone el ordenador para representar los datos son de tipo binario; esto es, cada elemento puede diferenciar dos estados (dos niveles de voltaje). Cada estado se denomina genéricamente *bit* y se simboliza por *0* ó *1*. Por lo tanto, para representar y manipular información numérica, alfábética y alfanumérica se emplean cadenas de *bits*. Según esto, se denomina *byte* a la cantidad de información empleada por un ordenador para representar un carácter; generalmente un *byte* es una cadena de ocho *bits*.



Así, por ejemplo, cuando un programa le dice al ordenador que visualice un mensaje sobre el monitor, o que lo imprima sobre la impresora, las instrucciones correspondientes para llevar a cabo esta acción, para que puedan ser entendibles por el ordenador, tienen que estar almacenadas en la memoria como cadenas de *bits*. Esto hace pensar que escribir un programa utilizando ceros y unos (lenguaje máquina), llevaría mucho tiempo y con muchas posibilidades de cometer errores. Por este motivo, se desarrollaron los lenguajes *ensambladores*.

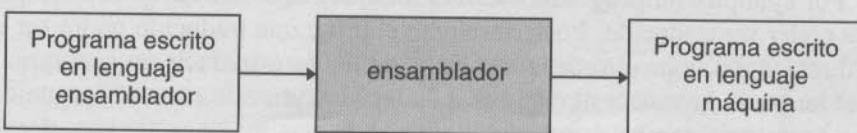
Un lenguaje *ensamblador* utiliza *códigos nemotécnicos* para indicarle al hardware (componentes físicos del ordenador) las operaciones que tiene que reali-

zar. Un código nemotécnico es una palabra o abreviatura fácil de recordar que representa una tarea que debe realizar el procesador del ordenador. Por ejemplo:

```
MOV AH, 4CH
```

El código *MOV* expresa una operación consistente en mover alguna información desde una posición de memoria a otra.

Para traducir un programa escrito en *ensamblador* a lenguaje máquina (código binario) se utiliza un programa llamado *ensamblador* que ejecutamos mediante el propio ordenador. Este programa tomará como datos nuestro programa escrito en lenguaje ensamblador y dará como resultado el mismo programa pero escrito en lenguaje máquina, lenguaje que entiende el ordenador.



Cada modelo de ordenador, dependiendo del procesador que utilice, tiene su propio lenguaje ensamblador. Debido a esto decimos que estos lenguajes están orientados a la máquina.

Hoy en día son más utilizados los lenguajes orientados al problema o lenguajes de alto nivel. Estos lenguajes utilizan una terminología fácilmente comprensible que se aproxima más al lenguaje humano. En este caso la traducción es llevada a cabo por otro programa denominado *compilador*.

Cada sentencia de un programa escrita en un lenguaje de alto nivel se descompone en general en varias instrucciones en ensamblador. Por ejemplo:

```
printf( "hola" );
```

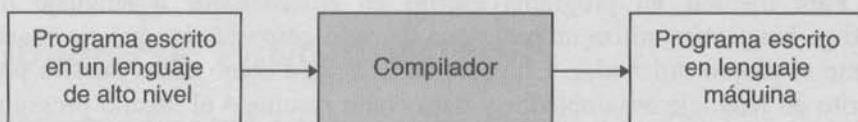
La función **printf** del lenguaje C le dice al ordenador que visualice en el monitor la cadena de caracteres especificada. Este mismo proceso escrito en lenguaje ensamblador necesitará de varias instrucciones. Lo mismo podríamos decir del método **println** de Java:

```
System.out.println( "hola" );
```

A diferencia de los lenguajes ensambladores, la utilización de lenguajes de alto nivel no requiere en absoluto del conocimiento de la estructura del procesador que utiliza el ordenador, lo que facilita la escritura de un programa.

Compiladores

Para traducir un programa escrito en un lenguaje de alto nivel (programa fuente) a lenguaje máquina se utiliza un programa llamado *compilador*. Este programa tomará como datos nuestro programa escrito en lenguaje de alto nivel y dará como resultado el mismo programa pero escrito en lenguaje máquina, programa que ya puede ejecutar directa o indirectamente el ordenador.



Por ejemplo, un programa escrito en el lenguaje C necesita del compilador C para poder ser traducido. Posteriormente el programa traducido podrá ser ejecutado directamente por el ordenador. En cambio, para traducir un programa escrito en el lenguaje Java necesita del compilador Java; en este caso, el lenguaje máquina no corresponde al del ordenador sino al de una máquina ficticia, denominada máquina virtual Java, que será puesta en marcha por el ordenador para ejecutar el programa.

¿Qué es una máquina virtual? Una máquina que no existe físicamente sino que es simulada en un ordenador por un programa. ¿Por qué utilizar una máquina virtual? Porque, por tratarse de un programa, es muy fácil instalarla en cualquier ordenador, basta con copiar ese programa en su disco duro, por ejemplo. Y, ¿qué ventajas reporta? Pues, en el caso de Java, que un programa escrito en este lenguaje y compilado, puede ser ejecutado en cualquier ordenador del mundo que tenga instalada esa máquina virtual. Esta solución hace posible que cualquier ordenador pueda ejecutar un programa escrito en Java independiente de la plataforma que utilice, lo que se conoce como *transportabilidad de programas*.

Intérpretes

A diferencia de un compilador, un intérprete no genera un programa escrito en lenguaje máquina a partir del programa fuente, sino que efectúa la traducción y ejecución simultáneamente para cada una de las sentencias del programa. Por ejemplo, un programa escrito en el lenguaje *Basic* necesita el intérprete *Basic* para ser ejecutado. Durante la ejecución de cada una de las sentencias del programa, ocurre simultáneamente la traducción.

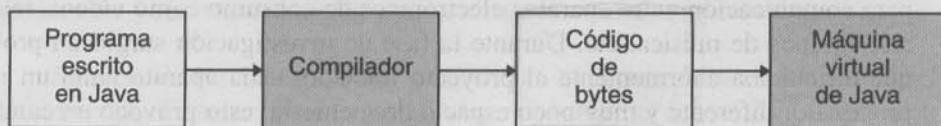
A diferencia de un compilador, un intérprete verifica cada línea del programa cuando se escribe, lo que facilita la puesta a punto del programa. En cambio la ejecución resulta más lenta ya que acarrea una traducción simultánea.

¿QUÉ ES JAVA?

Java es un lenguaje de programación de alto nivel con el que se pueden escribir tanto programas convencionales como para Internet.

Una de las ventajas significativas de Java sobre otros lenguajes de programación es que es independiente de la plataforma, tanto en código fuente como en binario. Esto quiere decir que el código producido por el compilador Java puede transportarse a cualquier plataforma (Intel, Sparc, Motorola, etc.) que tenga instalada una máquina virtual Java y ejecutarse. Pensando en Internet esta característica es crucial ya que esta red conecta ordenadores muy distintos. En cambio, C++, por ejemplo, es independiente de la plataforma sólo en código fuente, lo cual significa que cada plataforma diferente debe proporcionar el compilador adecuado para obtener el código máquina que tiene que ejecutarse.

Según lo expuesto, Java incluye dos elementos: un *compilador* y un *intérprete*. El compilador produce un código de bytes que se almacena en un fichero para ser ejecutado por el intérprete Java denominado *máquina virtual de Java*.



Los códigos de bytes de Java son un conjunto de instrucciones correspondientes a un lenguaje máquina que no es específico de ningún procesador, sino de la máquina virtual de Java. ¿Dónde se consigue esta máquina virtual? Hoy en día casi todas las compañías de sistemas operativos y de navegadores han implementado máquinas virtuales según las especificaciones publicadas por *Sun Microsystems*, propietario de Java, para que sean compatibles con el lenguaje Java. Para las aplicaciones de Internet (denominadas *applets*) la máquina virtual está incluida en el navegador y para las aplicaciones Java convencionales, puede venir con el sistema operativo, con el paquete Java, o bien puede obtenerla a través de Internet.

¿Por qué no se diseñó Java para que fuera un intérprete más entre los que hay en el mercado? La respuesta es porque la interpretación, si bien es cierto que proporciona independencia de la máquina, conlleva también un problema grave, y es la pérdida de velocidad en la ejecución del programa. Por esta razón la solución fue diseñar un compilador que produjera un lenguaje que pudiera ser interpretado a velocidades, si no iguales, sí cercanas a la de los programas nativos (programas en código máquina propio de cada ordenador), logro conseguido mediante la máquina virtual de Java.

Con todo, las aplicaciones todavía adolecen de una falta de rendimiento apreciable. Éste es uno de los problemas que siempre se ha achacado a Java. Afortunadamente, la diferencia de rendimiento con respecto a aplicaciones equivalentes escritas en código máquina nativo ha ido disminuyendo hasta márgenes muy reducidos gracias a la utilización de compiladores JIT (*Just In Time* - compilación al instante).

Un compilador JIT interacciona con la máquina virtual para convertir el código de bytes en código máquina nativo. Como consecuencia, se mejora la velocidad durante la ejecución. Sun sigue trabajando sobre este objetivo y prueba de ello son los resultados que se están obteniendo con el nuevo y potente motor de ejecución *HotSpot* (*HotSpot Performance Engine*) que ha diseñado, o por los microprocesadores específicos para la interpretación hardware de código de bytes.

HISTORIA DE JAVA

El lenguaje de programación Java fue desarrollado por *Sun Microsystems* en 1991. Nace como parte de un proyecto de investigación para desarrollar software para comunicación entre aparatos electrónicos de consumo como videos, televisores, equipos de música, etc. Durante la fase de investigación surgió un problema que dificultaba enormemente el proyecto iniciado: cada aparato tenía un microprocesador diferente y muy poco espacio de memoria; esto provocó un cambio en el rumbo de la investigación que desembocó en la idea de escribir un nuevo lenguaje de programación independiente del dispositivo que fue bautizado inicialmente como *Oak*.

La explosión de Internet en 1994, gracias al navegador gráfico *Mosaic* para la *World Wide Web* (*WWW*), no pasó desapercibida para el grupo investigador de Sun. Se dieron cuenta de que los logros alcanzados en su proyecto de investigación eran perfectamente aplicables a Internet. Comparativamente, Internet era como un gran conjunto de aparatos electrónicos de consumo, cada uno con un procesador diferente. Y es cierto; básicamente, Internet es una gran red mundial que conecta múltiples ordenadores con diferentes sistemas operativos y diferentes arquitecturas de microprocesadores, pero todos tienen en común un navegador que utilizan para comunicarse entre sí. Esta idea hizo que el grupo investigador abandonara el proyecto de desarrollar un lenguaje que permitiera la comunicación entre aparatos electrónicos de consumo y dirigiera sus investigaciones hacia el desarrollo de un lenguaje que permitiera crear aplicaciones que se ejecutaran en cualquier ordenador de Internet con el único soporte de un navegador.

A partir de aquí ya todo es conocido. Se empezó a hablar de Java y de sus aplicaciones, conocidas como *applets*. Un *applet* es un programa escrito en Java que se ejecuta en el contexto de una página *Web* en cualquier ordenador, indepen-

dientemente de su sistema operativo y de la arquitectura de su procesador. Para ejecutar un *applet* sólo se necesita un navegador que soporte la máquina virtual de Java como, por ejemplo, *Microsoft Internet Explorer* o *Netscape*. Utilizando un navegador de éstos, se puede descargar la página Web que contiene el *applet* y ejecutarlo. Precisamente en este campo, es donde Java como lenguaje de programación no tiene competidores. No obstante, con Java se puede programar cualquier cosa, razón por la que también puede ser considerado como un lenguaje de propósito general; pero, desde este punto de vista, hoy por hoy, Java tiene muchos competidores que le sobrepasan con claridad; por ejemplo Ada o C++.

¿POR QUÉ APRENDER JAVA?

Una de las ventajas más significativas de Java es su independencia de la plataforma. En el caso de que tenga que desarrollar aplicaciones que tengan que ejecutarse en sistemas diferentes esta característica es fundamental.

Otra característica importante de Java es que es un lenguaje de programación orientado a objetos (POO). Los conceptos en los que se apoya esta técnica de programación y sus ventajas serán expuestos en el capítulo siguiente.

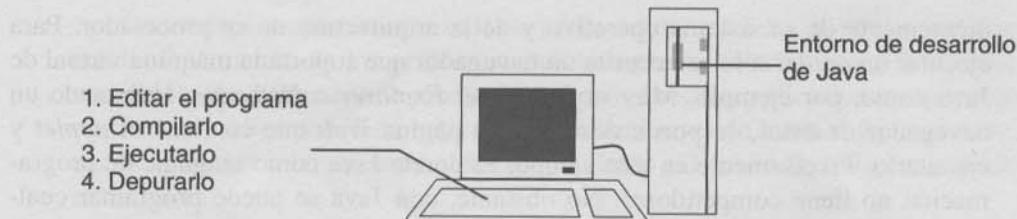
Además de ser transportable y orientado a objetos, Java es un lenguaje fácil de aprender. Tiene un tamaño pequeño que favorece el desarrollo y reduce las posibilidades de cometer errores; a la vez es potente y flexible.

Java está fundamentado en C++. Quiere esto decir que mucha de la sintaxis y diseño orientado a objetos se tomó de este lenguaje. Por lo tanto, a los lectores que estén familiarizados con C++ y la POO les será muy fácil aprender a desarrollar aplicaciones con Java. Quiero advertir a este tipo de potenciales usuarios de Java que en este lenguaje no existen punteros ni aritmética de punteros, las cadenas de caracteres son objetos y la administración de memoria es automática, lo que elimina la problemática que presenta C++ con las lagunas de memoria al olvidar liberar bloques de la misma que fueron asignados dinámicamente.

REALIZACIÓN DE UN PROGRAMA EN JAVA

En este apartado se van a exponer los pasos a seguir en la realización de un programa, por medio de un ejemplo.

La siguiente figura, muestra de forma esquemática lo que un usuario de Java necesita y debe hacer para desarrollar un programa.

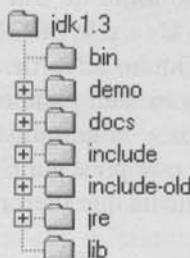


Evidentemente, para poder escribir programas se necesita un entorno de desarrollo Java. *Sun Microsystems*, propietario de Java, proporciona uno de forma gratuita, *Java Development Kit* (JDK), que se puede obtener en la dirección de Internet:

<http://www.sun.com>

Así mismo, el CD que acompaña al libro incluye *Java 2 SDK versión 1.3* para Windows 9x, Windows 2000/NT, y Unix. Se trata del nuevo JDK 1.3. En cualquier caso, en Internet se encuentran todas las versiones para Windows, Macintosh y Unix (Solaris y otros). Vea también el apéndice C.

Para instalar la versión que incluye el CD mencionado en una plataforma Windows, hay que ejecutar el fichero *j2sdk1_3_0-win.exe*. De manera predeterminada el paquete será instalado en *jdk1.3*, pero se puede instalar en cualquier otra carpeta. A continuación puede instalar en *jdk1.3\docs* la documentación que se proporciona en el fichero *j2sdk1_3_0-doc.zip*. Puede ver más detalles sobre la instalación al final del libro. Una vez finalizada la instalación, se puede observar el siguiente contenido:



- La carpeta *bin* contiene las herramientas de desarrollo. Esto es, los programas para compilar (*javac*), ejecutar (*java*), depurar (*jdb*), y documentar (*javadoc*) los programas escritos en el lenguaje de programación Java, y otras herramientas como *appletviewer* para ejecutar y depurar *applets* sin tener que utilizar un navegador, *jar* para manipular ficheros *.jar* (un fichero *.jar* es una colección de clases Java y otros ficheros empaquetados en uno solo), *javah* que es un fichero de cabecera para escribir métodos nativos, *javap* para descompilar ficheros compilados y *extcheck* para detectar conflictos *jar*.

- La carpeta *jre* es el entorno de ejecución de Java utilizado por el SDK. Es similar al intérprete de Java (*java*), pero destinado a usuarios finales que no requieran todas las opciones de desarrollo proporcionadas con la utilidad *java*. Incluye la máquina virtual, la biblioteca de clases, y otros ficheros que soportan la ejecución de programas escritos en Java.
- La carpeta *lib* contiene bibliotecas de clases adicionales y ficheros de soporte requeridos por las herramientas de desarrollo.
- La carpeta *demo* contiene ejemplos.
- La carpeta *include* contiene los ficheros de cabecera que dan soporte para añadir a un programa Java código nativo (código escrito en un lenguaje distinto de Java, por ejemplo Ada o C++).
- La carpeta *include-old* contiene los ficheros de cabecera que dan soporte para añadir a un programa Java código nativo utilizando interfaces antiguas.
- El nuevo JDK 1.3 incluye un compilador JIT para ejecutar el código en Java. Por omisión el compilador JIT empleado será *jre\bin\symjit.dll*.

Sólo falta un editor de código fuente Java. Es suficiente con un editor de texto sin formato; por ejemplo el *bloc de notas* de Windows. No obstante, todo el trabajo de edición, compilación, ejecución y depuración, se hará mucho más fácil si se utiliza un entorno de desarrollo con interfaz gráfica de usuario que integre las herramientas mencionadas, en lugar de tener que utilizar las interfaz de línea de órdenes del JDK, como veremos a continuación.

Entornos de desarrollo integrados para Java hay muchos: *Forte* de Sun, *Visual Café* de Symantec, *JBuilder* de Borland, *Kawa* de Tek-Tools, *Visual Age Windows* de IBM, *pcGRASP* de Auburn University, *Visual J++* de Microsoft, etc. Las versiones de demostración operativas de algunos de ellos las tiene en el CD que se adjunta con este libro. Concretamente, *pcGRASP* es un entorno integrado sencillo, que no requiere licencia, y que se ajusta a las necesidades de las aplicaciones que serán expuestas en este libro.

Cómo crear un programa

Un programa puede ser una *aplicación* o un *applet*. Con este libro aprenderá principalmente a escribir aplicaciones Java. Después aplicará lo aprendido para escribir algunos *applets*. Empecemos con la creación de una aplicación sencilla: el clásico ejemplo de mostrar un mensaje de saludo.

Esta sencilla aplicación la realizaremos desde los dos puntos de vista comentados anteriormente: utilizando la interfaz de línea de órdenes del JDK y utilizando un entorno de desarrollo integrado.

Interfaz de línea de órdenes

Empecemos por editar el fichero fuente Java correspondiente a la aplicación. Primeramente visualizaremos el editor de textos que vayamos a utilizar; por ejemplo, el *Block de notas* o el *Edit*. El nombre del fichero elegido para guardar el programa en el disco, debe tener como extensión *java*; por ejemplo *HolaMundo.java*.

Una vez visualizado el editor, escribiremos el texto correspondiente al programa fuente. Escríbalo tal como se muestra a continuación. Observe que cada *sentencia* del lenguaje Java finaliza con un *punto y coma* y que cada *línea del programa* se finaliza pulsando la tecla *Entrar (Enter o ↵)*.

```
class HolaMundo
{
    /*
     * Punto de entrada a la aplicación.
     *
     * args: matriz de parámetros pasados a la aplicación
     * mediante la línea de órdenes. Puede estar vacía.
     */
    public static void main(String[] args)
    {
        System.out.println("Hola mundo!!!");
    }
}
```

¿Qué hace este programa?

Comentamos brevemente cada línea de este programa. No apurarse si algunos de los términos no quedan muy claros ya que todos ellos se verán con detalle en capítulos posteriores.

La primera línea declara la clase de objetos *HolaMundo*, porque el esqueleto de cualquier aplicación Java se basa en la definición de una clase. A continuación se escribe el cuerpo de la clase encerrado entre los caracteres { y }. Ambos caracteres definen un bloque de código. Todas las acciones que va a llevar a cabo un programa Java se colocan dentro del bloque de código correspondiente a su clase. Las clases son la base de los programas Java. Aprenderemos mucho sobre ellas en los próximos capítulos.

Las siguientes líneas encerradas entre /* y */ son simplemente un comentario. Los comentarios no son tenidos en cuenta por el compilador, pero ayudan a entender un programa cuando se lee.

A continuación se escribe el método principal **main**. Observe que un método se distingue por el modificador () que aparece después de su nombre y que el bloque de código correspondiente al mismo define las acciones que tiene que ejecutar dicho método. Cuando se ejecuta una aplicación, Java espera que haya un método **main**. Este método define el punto de entrada y de salida de la aplicación.

El método **println** del objeto **out** miembro de la clase **System** de la biblioteca Java, escribe como resultado la expresión que aparece especificada entre comillas. Observe que la sentencia completa finaliza con punto y coma.

Guardar el programa escrito en el disco

El programa editado está ahora en la memoria. Para que este trabajo pueda tener continuidad, el programa escrito se debe grabar en el disco utilizando la orden correspondiente del editor. Muy importante: el nombre del programa fuente debe ser el mismo que el de la clase que contiene, respetando mayúsculas y minúsculas. En nuestro caso, el nombre de la clase es *HolaMundo*, por lo tanto el fichero debe guardarse con el nombre *HolaMundo.java*.

Compilar y ejecutar el programa

El siguiente paso es *compilar* el programa; esto es, traducir el programa fuente a código de bytes para posteriormente poder ejecutarlo. La orden para compilar el programa *HolaMundo.java* es la siguiente:

```
javac HolaMundo.java
```

Previamente, para que el sistema operativo encuentre la utilidad *javac*, utilizando la línea de órdenes hay que añadir a la variable de entorno *path* la ruta de la carpeta donde se encuentra esta utilidad y otras que utilizaremos a continuación. Por ejemplo:

```
path=%path%;c:\jdk1.3\bin
```

La expresión *%path%* representa el valor actual de la variable de entorno *path*. Una ruta va separada de la anterior por un punto y coma.

En la figura siguiente se puede observar el proceso seguido para compilar *HolaMundo.java* desde la línea de órdenes.

```

MS-DOS
Auto [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] A

Microsoft(C) Windows 98
(C)Copyright Microsoft Corp 1981-1999.

C:\WINDOWS>path=%path%;c:\java\jdk1.3\bin
C:\WINDOWS>cd C:\Cap01\HolaMundo
C:\Cap01\HolaMundo>dir
El volumen de la unidad C no tiene etiqueta
El número de serie del volumen es 2E1F-07E8
Directorio de C:\Cap01\HolaMundo
.
<DIR> 05/06/00 22:28 .
<DIR> 05/06/00 22:28 ..
1 archivo    299 05/06/00 0:00 HolaMundo.java
2 directorios   299 bytes
1.530.392.576 bytes libres
C:\Cap01\HolaMundo>javac HolaMundo.java
C:\Cap01\HolaMundo>

```

Obsérvese que para compilar un programa hay que especificar la extensión *.java*. El resultado de la compilación será un fichero *HolaMundo.class* que contiene el código de bytes que ejecutará la máquina virtual de Java.

Al compilar un programa, se pueden presentar *errores de compilación*, debidos a que el programa escrito no se adapta a la sintaxis y reglas del compilador. Estos errores se irán corrigiendo hasta obtener una compilación sin errores.

Para ejecutar el fichero resultante de la compilación, invocaremos desde la línea de órdenes al intérprete de código de bytes *java* con el nombre de dicho fichero como argumento, en nuestro caso *HolaMundo*, y pulsaremos *Entrar* para que se muestren los resultados.

En la figura siguiente se puede observar el proceso seguido para ejecutar *HolaMundo* desde la línea de órdenes. Hay que respetar las mayúsculas y las minúsculas en los nombres que se escriben. Así mismo, cabe resaltar que la extensión *.class* no tiene que ser especificada.

Una vez ejecutado, se puede observar que el resultado es el mensaje: *Hola mundo!!!*.

```

MS-DOS
Auto
<DIR> 05/06/00 22:28 .
<DIR> 05/06/00 22:28 ..
HOLAMUNDO JRU 299 05/06/00 0:00 HolaMundo.java
1 archivos 299 bytes
2 directorios 1.530.392.576 bytes libres

C:\Cap01\HolaMundo>javac HolaMundo.java

C:\Cap01\HolaMundo>dir

El volumen de la unidad C no tiene etiqueta
El n mero de serie del volumen es 2E1F-07E8
Directorio de C:\Cap01\HolaMundo

<DIR> 05/06/00 22:28 .
<DIR> 05/06/00 22:28 ..
HOLAMUNDO JRU 299 05/06/00 0:00 HolaMundo.java
HOLAMUNDO CLA 425 05/06/00 22:40 HolaMundo.class
2 archivos 724 bytes
2 directorios 1.520.766.976 bytes libres

C:\Cap01\HolaMundo>java HolaMundo
Hola mundo!!!
C:\Cap01\HolaMundo>

```

Biblioteca de funciones

Java carece de instrucciones de E/S, de instrucciones para manejo de cadenas de caracteres, etc. con lo que este trabajo queda para la biblioteca de clases provista con el compilador. Una biblioteca es un fichero separado en el disco (con extensi n *.lib*, *.jar* o *.dll*) que contiene las clases que definen las tareas m s comunes, para que nosotros no tengamos que escribirlas. Como ejemplo, hemos visto anteriormente el m todo **println** del objeto **out** miembro de la clase **System**. Si este m todo no existiera, ser a labor nuestra el escribir el c digo necesario para visualizar los resultados en la ventana.

En el c digo escrito anteriormente se puede observar que para utilizar un m todo de una clase de la biblioteca simplemente hay que invocarlo para un objeto de su clase y pasare los argumentos necesarios entre par ntesis. Por ejemplo:

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

Guardar el programa ejecutable en el disco

Como hemos visto, cada vez que se realiza el proceso de *compilaci n* del programa actual, Java genera autom ticamente sobre el disco un fichero *.class*. Este fichero puede ser ejecutado directamente desde el sistema operativo, con el soporte de la m quina virtual de Java, que se lanza invocando a la utilidad *java* con el nombre del fichero como argumento.

Al ejecutar el programa, pueden ocurrir *errores durante la ejecución*. Por ejemplo, puede darse una división por cero. Estos errores solamente pueden ser detectados por Java cuando se ejecuta el programa y serán notificados con el correspondiente mensaje de error.

Hay *otro tipo de errores* que no dan lugar a mensaje alguno. Por ejemplo: un programa que no termine nunca de ejecutarse, debido a que presenta un lazo, donde no se llega a dar la condición de terminación. Para detener la ejecución se tienen que pulsar las teclas *Ctrl+C* (en un entorno integrado se ejecutará una orden equivalente a *Detener ejecución*).

Depurar un programa

Una vez ejecutado el programa, la solución puede ser incorrecta. Este caso exige un análisis minucioso de cómo se comporta el programa a lo largo de su ejecución; esto es, hay que entrar en la fase de *depuración* del programa.

La forma más sencilla y eficaz para realizar este proceso, es utilizar un programa *depurador*. El entorno de desarrollo de Java proporciona para esto la utilidad *jdb*. Éste es un depurador de línea de órdenes un tanto complicado de utilizar, por lo que, en principio, tiene escasa aceptación. Normalmente los entornos de desarrollo integrados que anteriormente hemos mencionado (excepto *FreeJava*) incorporan las órdenes necesarias para invocar y depurar un programa.

Para depurar un programa Java debe compilarlo con la opción *-g*. Por ejemplo, desde la línea de órdenes esto se haría así:

```
javac -g Aritmetica.java
```

Desde un entorno integrado, habrá que establecer las opción correspondiente del compilador.

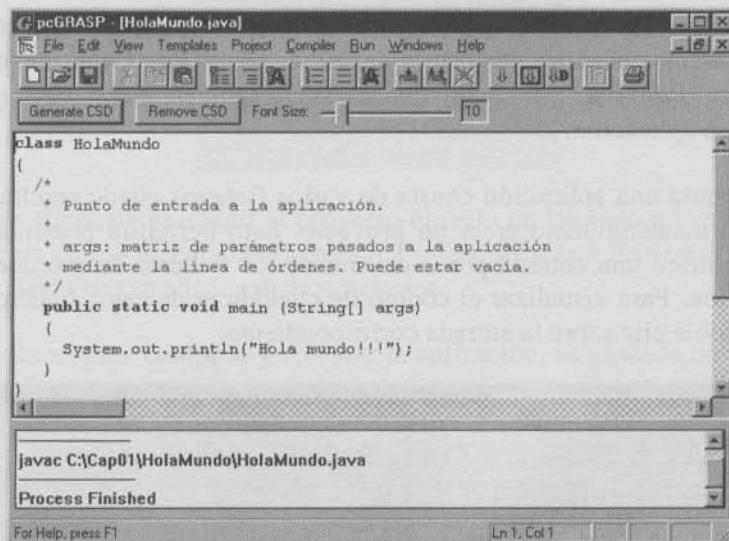
Una vez compilado el programa, se inicia la ejecución en modo depuración y se continúa la ejecución con las órdenes típicas de ejecución paso a paso.

Entorno de desarrollo integrado

Cuando se utiliza un entorno de desarrollo integrado lo primero que hay que hacer una vez instalado dicho entorno es asegurarse de que las opciones que indican las rutas de las herramientas Java, de las bibliotecas, de la documentación y de los fuentes, o bien simplemente de la ruta donde se instaló el JDK, están establecidas. Por ejemplo, si utiliza el entorno de desarrollo integrado *pcGRASP* que se proporciona en el CD, en el menú *File* del mismo encontrará una orden *Global Preferences*.

rencias... que le permitirá especificar la ruta de la carpeta donde ha instalado el JDK. Otros entornos proporcionarán una orden *Opciones* o equivalente.

En la siguiente figura se puede observar el aspecto del entorno de desarrollo integrado *pcGRASP*.



Para editar y ejecutar la aplicación *HolaMundo* anterior utilizando este entorno de desarrollo integrado, los pasos a seguir se indican a continuación:

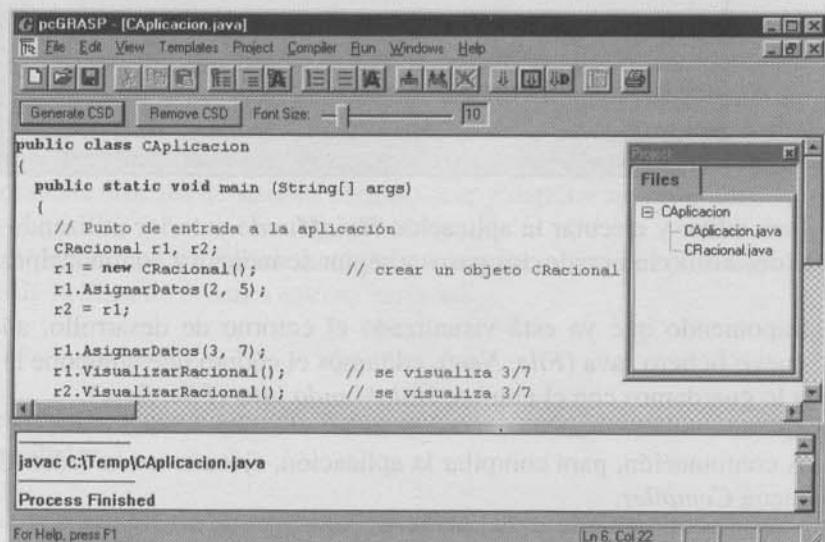
1. Suponiendo que ya está visualizado el entorno de desarrollo, añadimos un nuevo fichero Java (*File, New*), editamos el código que compone la aplicación y lo guardamos con el nombre *HolaMundo.java* (*File, Save*).
2. A continuación, para compilar la aplicación, ejecutamos la orden *Compile* del menú *Compiler*.
3. Finalmente, para ejecutar la aplicación, suponiendo que la compilación se efectuó satisfactoriamente, seleccionaremos la orden *Run Captured*, o bien *Run*, del menú *Run*.

En capítulos posteriores, implementará aplicaciones compuestas por varios ficheros fuente. En este caso, los pasos a seguir son los siguientes:

1. Suponiendo que ya está visualizado el entorno de desarrollo, añadimos (*File, New*) y editamos cada uno de los ficheros que componen la aplicación y los guardamos con los nombres adecuados (*File, Save*).

- Para compilar la aplicación visualizaremos el fichero que contiene la clase aplicación (la clase que contiene el método **main**) y ejecutaremos la orden *Compile* del menú *Compiler*. Se puede observar que todas las clases que son requeridas por esta clase principal son compiladas automáticamente, independientemente de que estén, o no, en diferentes ficheros.
- Finalmente, suponiendo que la compilación se efectuó satisfactoriamente, ejecutaremos la aplicación seleccionando la orden *Run Captured*, o bien *Run*, del menú *Run*. Este paso exige que se esté visualizando el fichero que contiene la clase aplicación.

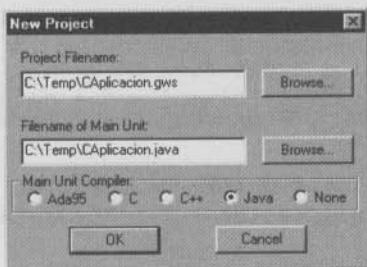
Cuando una aplicación consta de varios ficheros puede resultar más cómodo para su mantenimiento crear un proyecto. Esto permitirá presentar una ventana que mostrará una entrada por cada uno de los ficheros fuente que componen la aplicación. Para visualizar el código de cualquiera de estos ficheros, bastará con hacer doble clic sobre la entrada correspondiente.



Para crear un proyecto, los pasos a seguir son los siguientes:

- Suponiendo que ya está visualizado el entorno de desarrollo, se añaden (*File, New*) y editan cada uno de los ficheros que componen la aplicación y se guardan con los nombres adecuados (*File, Save*).
- Después se crea un nuevo proyecto (*Project, New Project Workspace*). En la ventana que se visualiza, haciendo clic en el botón *Browse* correspondiente, se introduce un nombre para el proyecto y se selecciona el nombre de la clase aplicación (la clase que contiene el método **main**). Posteriormente, para vi-

sualizar la ventana del proyecto, se ejecuta la orden *Show Project Workspace Window* del menú *Project*.



3. El paso siguiente es añadir al proyecto el resto de los ficheros que lo componen. Para ello se ejecuta la orden *Edit Project* del menú *Project*, y a través del botón *Add Files* se añaden dichos ficheros.
4. Finalmente, para compilar y ejecutar la aplicación, se procede como se explicó anteriormente.

EJERCICIOS RESUELTOS

Para practicar con un programa más, escriba el siguiente ejemplo y pruebe los resultados. Hágalo primero desde la línea de órdenes y después con el entorno de desarrollo integrado preferido por usted. El siguiente ejemplo visualiza como resultado la suma, la resta, la multiplicación y la división de dos cantidades enteras.

Edición

Abra el procesador de textos o el editor de su entorno integrado y edite el programa ejemplo que se muestra a continuación. Recuerde, el nombre del fichero fuente tiene que coincidir con el nombre de la clase, *CAritmetica*, respetando mayúsculas y minúsculas, y debe tener extensión *.java*.

```
class CAritmetica
{
/*
 * Operaciones aritméticas
 */
public static void main (String[] args)
{
    int dato1, dato2, resultado;
    dato1 = 20;
    dato2 = 10;
```

```
// Suma  
resultado = dato1 + dato2;  
System.out.println(dato1 + " + " + dato2 + " = " + resultado);  
  
// Resta  
resultado = dato1 - dato2;  
System.out.println(dato1 + " - " + dato2 + " = " + resultado);  
  
// Producto  
resultado = dato1 * dato2;  
System.out.println(dato1 + " * " + dato2 + " = " + resultado);  
  
// Cociente  
resultado = dato1 / dato2;  
System.out.println(dato1 + " / " + dato2 + " = " + resultado);  
}  
}
```

Una vez editado el programa, guárdelo en el disco con el nombre *CAritmetica.java*.

¿Qué hace este programa?

Fijándonos en el método principal, **main**, vemos que se han declarado tres variables enteras (de tipo **int**): *dato1*, *dato2* y *resultado*.

```
int dato1, dato2, resultado;
```

El siguiente paso asigna el valor 20 a la variable *dato1* y el valor 10 a la variable *dato2*.

```
dato1 = 20;  
dato2 = 10;
```

A continuación se realiza la suma de esos valores y se escriben los datos y el resultado.

```
resultado = dato1 + dato2;  
System.out.println(dato1 + " + " + dato2 + " = " + resultado);
```

El método **println** escribe un resultado de la forma:

20 + 10 = 30

Observe que la expresión resultante está formada por cinco elementos: *dato1*, " + ", *dato2*, " = ", y *resultado*. Unos elementos son numéricos y otros son cons-

tantes de caracteres. Para unir los cinco elementos en uno solo, se ha empleado el operador +.

Un proceso similar se sigue para calcular la diferencia, el producto y el cociente.

Para finalizar, compile, ejecute la aplicación y observe los resultados.

EJERCICIOS PROPUESTOS

Practique la edición, la compilación y la ejecución con un programa similar al programa *Aritmetica.java* realizado en el apartado anterior. Por ejemplo, modifíquelo para que ahora realice las operaciones de sumar, restar y multiplicar con tres datos: *dato1*, *dato2*, *dato3*. En un segundo intento, puede también combinar las operaciones aritméticas.

CAPÍTULO 2

© F.J.Ceballos/RA-MA

PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos (POO) es un modelo de programación que utiliza objetos, ligados mediante mensajes, para la solución de problemas. Puede considerarse como una extensión natural de la programación estructurada en un intento de potenciar los conceptos de modularidad y reutilización del código.

¿A qué objetos nos referimos? Si nos paramos a pensar en un determinado problema que intentamos resolver podremos identificar entidades de interés, las cuales pueden ser objetos potenciales que poseen un conjunto de *propiedades* o atributos, y un conjunto de *métodos* mediante los cuales muestran su comportamiento. Y no sólo eso, también podremos ver, a poco que nos fijemos, un conjunto de interrelaciones entre ellos conducidas por mensajes a los que responden mediante métodos.

Veamos un ejemplo. Considere una entidad bancaria. En ella identificamos entidades que son cuentas: cuenta del cliente 1, cuenta del cliente 2, etc. Pues bien, una cuenta puede verse como un objeto que tiene unos atributos, *nombre*, *número de cuenta* y *saldo*, y un conjunto de métodos como *IngresarDinero*, *RetirarDinero*, *AbonarIntereses*, *SaldoActual*, *Transferencia* etc. En el caso de una transferencia:

```
cuenta01.Transferencia(cuenta02);
```

Transferencia sería el mensaje que el objeto *cuenta02* envía al objeto *cuenta01*, solicitando le sea hecha una transferencia, siendo la respuesta a tal mensaje la ejecución del método *Transferencia*. Trabajando a este nivel de abstracción, manipular una entidad bancaria resultará algo muy sencillo.

MECANISMOS BÁSICOS DE LA POO

Los mecanismos básicos de la programación orientada a objetos son: *objetos*, *mensajes*, *métodos* y *clases*.

Objetos

Un programa orientado a objetos se compone solamente de *objetos*, entendiendo por objeto una encapsulación genérica de datos y de los métodos para manipularlos. Dicho de otra forma, un *objeto* es una entidad que tiene unos atributos particulares, las *propiedades*, y unas formas de operar sobre ellos, los *métodos*.

Por ejemplo, una ventana de una aplicación Windows es un objeto. El color de fondo, la anchura, la altura, etc. son propiedades. Las rutinas, lógicamente transparentes al usuario, que permiten maximizar la ventana, minimizarla, etc. son métodos.

Mensajes

Cuando se ejecuta un programa orientado a objetos, los objetos están recibiendo, interpretando y respondiendo a *mensajes* de otros objetos. Esto marca una clara diferencia con respecto a los elementos de datos pasivos de los sistemas tradicionales. En la POO un *mensaje* está asociado con un método, de tal forma que cuando un objeto recibe un mensaje la respuesta a ese mensaje es ejecutar el método asociado.

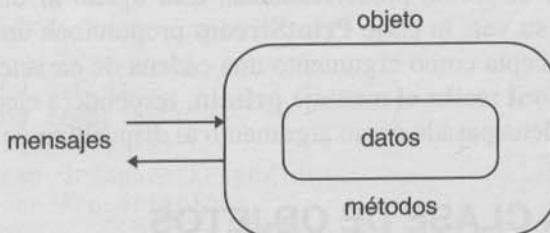
Por ejemplo, cuando un usuario quiere maximizar una ventana de una aplicación Windows, lo que hace simplemente es pulsar el botón de la misma que realiza esa acción. Eso, provoca que Windows envíe un mensaje a la ventana para indicar que tiene que maximizarse. Como respuesta a este mensaje se ejecutará el método programado para ese fin.

Métodos

Un *método* se implementa en una *clase* de objetos y determina cómo tiene que actuar el objeto cuando recibe el *mensaje* vinculado con ese método. A su vez, un *método* puede también enviar *mensajes* a otros objetos solicitando una acción o información.

En adición, las *propiedades* (atributos) definidas en la clase permitirán almacenar información para dicho objeto.

Cuando se diseña una clase de objetos, la estructura más interna del *objeto* se oculta a los usuarios que lo vayan a utilizar, manteniendo como única conexión con el exterior, los *mensajes*. Esto es, los datos que están dentro de un objeto solamente podrán ser manipulados por los *métodos* asociados al propio objeto.



Según lo expuesto, podemos decir que la ejecución de un programa orientado a objetos realiza fundamentalmente tres cosas:

1. Crea los objetos necesarios.
2. Los mensajes enviados a unos y a otros objetos dan lugar a que se procese internamente la información.
3. Finalmente, cuando los objetos no son necesarios, son borrados, liberándose la memoria ocupada por los mismos.

Clases

Una *clase* es un tipo de objetos definido por el usuario. Una *clase* equivale a la generalización de un tipo específico de objetos. Por ejemplo, piense en un molde para hacer flanes; el molde es la *clase* y los flanes los *objetos*.

Un *objeto* de una determinada clase se crea en el momento en que se define una variable de dicha *clase*. Por ejemplo, la siguiente línea declara el objeto *cliente01* de la clase o tipo *CCuenta*.

```
CCuenta cliente01 = new CCuenta(); // nueva cuenta
```

Algunos autores emplean el término *instancia* (traducción directa de *instance*), en el sentido de que una instancia es la representación concreta y específica de una clase; por ejemplo, *cliente01* es un instancia de la clase *CCuenta*. Desde este punto de vista, los términos *instancia* y *objeto* son lo mismo. El autor prefiere utilizar el término *objeto*, o bien *ejemplar*.

Cuando escribe un programa utilizando un lenguaje orientado a objetos, no se definen objetos verdaderos, se definen clases de objetos, donde una clase se ve como una plantilla para múltiples objetos con características similares.

Afortunadamente no tendrá que escribir todas las clases que necesite en su programa, porque Java proporciona una biblioteca de clases estándar para realizar las operaciones más habituales que podamos requerir. Por ejemplo, en el capítulo anterior, vimos que la clase **System** tenía un atributo **out** que era un objeto de la clase **PrintStream** que, de forma predeterminada, está ligado al dispositivo de salida (a la pantalla). A su vez, la clase **PrintStream** proporciona un método denominado **println** que acepta como argumento una cadena de caracteres. De esta forma, cuando el objeto **out** reciba el mensaje **println**, responderá ejecutando este método, que envía la cadena pasada como argumento al dispositivo de salida.

CÓMO CREAR UNA CLASE DE OBJETOS

Según lo expuesto hasta ahora, un objeto contiene, por una parte, atributos que definen su estado, y por otra, operaciones que definen su comportamiento. También sabemos que un objeto es la representación concreta y específica de una clase. ¿Cómo se escribe una clase de objetos? Como ejemplo, podemos crear una clase *COrdenador*. Abra su entorno de programación integrado favorito y escriba paso a paso el ejemplo que a continuación empezamos a desarrollar:

```
class COrdenador
{
    // ...
}
```

Observamos que para declarar una clase hay que utilizar la palabra reservada **class** seguida del nombre de la clase y del cuerpo de la misma. El cuerpo de la clase incluirá entre { y } los atributos y los métodos u operaciones que definen su comportamiento.

Los atributos son las características individuales que diferencian un objeto de otro. El color de una ventana Windows, la diferencia de otras; el D.N.I. de una persona la identifica entre otras; el modelo de un ordenador le distingue entre otros; etc.

La clase *COrdenador* puede incluir los siguientes atributos:

- ◊ Marca: Mitac, Toshiba, Ast
- ◊ Procesador: Intel, AMD
- ◊ Pantalla: TFT, DSTN, STN

Los atributos también pueden incluir información sobre el estado del objeto; por ejemplo, en el caso de un ordenador, si está encendido o apagado, si la presentación en pantalla está activa o inactiva, etc.

- ◊ Dispositivo: encendido, apagado
- ◊ Presentación: activa, inactiva

Todos los atributos son definidos en la clase por variables:

```
class COrdenador
```

```
{
    String Marca;
    String Procesador;
    String Pantalla;
    boolean OrdenadorEncendido;
    boolean Presentación;
```

```
// ...
```

```
}
```

Observe que se han definido cinco atributos: tres de ellos, *Marca*, *Procesador* y *Pantalla*, pueden contener una cadena de caracteres (una cadena de caracteres es un objeto de la clase **String** perteneciente a la biblioteca estándar). Los otros dos atributos, *OrdenadorEncendido* y *Presentación*, son de tipo **boolean** (un atributo de tipo **boolean** puede contener un valor **true** o **false**; verdadero o falso). Debe respetar las mayúsculas y las minúsculas.

No vamos a profundizar en los detalles de la sintaxis de este ejemplo ya que el único objetivo ahora es entender la definición de una clase con sus partes básicas. El resto de la sintaxis y demás detalles se irán exponiendo poco a poco en sucesivos capítulos.

El comportamiento define las acciones que el objeto puede emprender. Por ejemplo, pensando acerca de un objeto de la clase *COrdenador*, esto es, de un ordenador, algunas acciones que éste puede hacer son:

- ◊ Ponerse en marcha
- ◊ Apagarse
- ◊ Desactivar la presentación en la pantalla
- ◊ Activar la presentación en la pantalla
- ◊ Cargar una aplicación

Para definir este comportamiento hay que crear métodos. Los métodos son rutinas de código definidas dentro de la clase, que se ejecutan en respuesta a alguna acción tomada desde dentro de un objeto de esa clase o desde otro objeto de la misma o de otra clase. Recuerde que los objetos se comunican mediante mensajes.

Como ejemplo, vamos a agregar a la clase *COrdenador* un método que responda a la acción de ponerlo en marcha:

```

void EncenderOrdenador()
{
    if (OrdenadorEncendido == true) // si está encendido...
        System.out.println("El ordenador ya está en marcha.");
    else // si no está encendido, encenderlo.
    {
        OrdenadorEncendido = true;
        System.out.println("El ordenador se ha encendido.");
    }
}

```

Como se puede observar un método consta de su nombre precedido por el tipo del valor que devuelve cuando finalice su ejecución (la palabra reservada **void** indica que el método no devuelve ningún valor) y seguido por una lista de parámetros separados por comas y encerrados entre paréntesis (en el ejemplo, no hay parámetros). Los paréntesis indican a Java que el identificador (*EncenderOrdenador*) se refiere a un método y no a un atributo. A continuación se escribe el cuerpo del método encerrado entre { y }. Usted ya conoce algunos métodos, llamados en otros contextos funciones; seguro que conoce la función logaritmo que devuelve un valor real correspondiente al logaritmo del valor pasado como argumento.

El método *EncenderOrdenador* comprueba si el ordenador está encendido; si lo está, simplemente visualiza un mensaje indicándolo; si no lo está, se enciende y lo comunica mediante un mensaje.

Agreguemos un método más para que el objeto nos muestre su estado:

```

void Estado()
{
    System.out.println("\nEstado del ordenador:" +
                       "\nMarca " + Marca +
                       "\nProcesador " + Procesador +
                       "\nPantalla " + Pantalla + "\n");
    if (OrdenadorEncendido == true) // si el ordenador está encendido...
        System.out.println("El ordenador está encendido.");
    else // si no está encendido...
        System.out.println("El ordenador está apagado.");
}

```

El método *Estado* visualiza los atributos específicos de un objeto. La secuencia de escape \n, así se denomina, introduce un retorno de carro más un avance de línea (caracteres ASCII, CR LF).

En este instante, si nuestras pretensiones sólo son las expuestas hasta ahora, ya tenemos creada la clase *COrdenador*. Para poder crear objetos de esta clase y trabajar con ellos, tendremos que escribir un programa, o bien añadir a esta clase

el método **main**. Siempre que se trate de una aplicación (no de un *applet*) es obligatorio que la clase que define el comienzo de la misma incluya un método **main**. Cuando se ejecuta una clase Java compilada que incluye un método **main**, éste es lo primero que se ejecuta.

Hagamos lo más sencillo, añadir el método **main** a la clase *COrdenador*. El código completo, incluyendo el método **main**, se muestra a continuación:

```
class COrdenador
{
    String Marca;
    String Procesador;
    String Pantalla;
    boolean OrdenadorEncendido;
    boolean Presentación;

    void EncenderOrdenador()
    {
        if (OrdenadorEncendido == true) // si está encendido...
            System.out.println("El ordenador ya está encendido.");
        else // si no está encendido, encenderlo.
        {
            OrdenadorEncendido = true;
            System.out.println("El ordenador se ha encendido.");
        }
    }

    void Estado()
    {
        System.out.println("\nEstado del ordenador:" +
                           "\nMarca " + Marca +
                           "\nProcesador " + Procesador +
                           "\nPantalla " + Pantalla + "\n");
        if (OrdenadorEncendido == true) // si el ordenador está encendido...
            System.out.println("El ordenador está encendido.");
        else // si no está encendido...
            System.out.println("El ordenador está apagado.");
    }

    public static void main (String[] args)
    {
        COrdenador MiOrdenador = new COrdenador();
        MiOrdenador.Marca = "Ast";
        MiOrdenador.Procesador = "Intel Pentium";
        MiOrdenador.Pantalla = "TFT";
        MiOrdenador.EncenderOrdenador();
        MiOrdenador.Estado();
    }
}
```

El método **main** siempre se declara público y estático, no devuelve un resultado y tiene un parámetro *args* que es una matriz de una dimensión de cadenas de caracteres. En los capítulos siguientes aprenderá para qué sirve. Analicemos el método **main** para que tenga una idea de lo que hace:

- La primera línea crea un objeto de la clase *COrdenador* y almacena una referencia al mismo en la variable *MiOrdenador*. Esta variable la utilizaremos para acceder al objeto en las siguientes líneas. Ahora quizás empieze a entender por qué anteriormente decíamos que un programa orientado a objetos se compone solamente de objetos.
- Las tres líneas siguientes establecen los atributos del objeto referenciado por *MiOrdenador*. Se puede observar que para acceder a los atributos o propiedades del objeto se utiliza el operador punto (.). De esta forma quedan eliminadas las ambigüedades que surgirían si hubiéramos creado más de un objeto.
- En las dos últimas líneas el objeto recibe los mensajes *EncenderOrdenador* y *Estado*. La respuesta a esos mensajes es la ejecución de los métodos respectivos, que fueron explicados anteriormente. Aquí también se puede observar que para acceder a los métodos del objeto se utiliza el operador punto.

En general, para acceder a un miembro de una clase (atributo o método) se utiliza la sintaxis siguiente:

nombre_objeto.nombre_miembro

Guarde la aplicación con el nombre *COrdenador.java*. Después compílela y ejecútela. Se puede observar que los resultados son los siguientes:

El ordenador se ha encendido.

Estado del ordenador:

Marca Ast

Procesador Intel Pentium

Pantalla TFT

El ordenador está encendido.

Otra forma de crear objetos de una clase y trabajar con ellos es incluir esa clase en el mismo fichero fuente de una clase aplicación, entendiendo por clase aplicación una que incluya el método **main** y cree objetos de otras clases. Por ejemplo, volvamos al instante justo antes de añadir el método **main** a la clase *COrdenador* y añadamos una nueva clase pública denominada *CMiOrdenador* que incluya el método **main**. El resultado tendrá el esqueleto que se observa a continuación:

```

public class CMiOrdenador
{
    public static void main (String[] args)
    {
        // ...
    }
}

class COrdenador
{
    // ...
}

```

En el capítulo anterior aprendimos que una aplicación está basada en una clase cuyo nombre debe coincidir con el del programa fuente que la contenga, respetando mayúsculas y minúsculas. Por lo tanto, guardemos el código escrito en un fichero fuente denominado *CMiOrdenador.java*. Finalmente, completamos el código como se observa a continuación, y compilamos y ejecutamos la aplicación. Ahora es la clase *CMiOrdenador* la que crea un objeto de la clase *COrdenador*. El resto del proceso se desarrolla como se explicó en la versión anterior. Lógicamente, los resultados que se obtengan serán los mismos que obtuvimos con la versión anterior.

```

public class CMiOrdenador
{
    public static void main (String[] args)
    {
        COrdenador MiOrdenador = new COrdenador();
        MiOrdenador.Marca = "Ast";
        MiOrdenador.Procesador = "Intel Pentium";
        MiOrdenador.Pantalla = "TFT";
        MiOrdenador.EncenderOrdenador();
        MiOrdenador.Estado();
    }
}

class COrdenador
{
    String Marca;
    String Procesador;
    String Pantalla;
    boolean OrdenadorEncendido;
    boolean Presentación;

    void EncenderOrdenador()
    {
        if (OrdenadorEncendido == true) // si está encendido...
            System.out.println("El ordenador ya está encendido.");
    }
}

```

```

        else // si no está encendido, encenderlo.
    {
        OrdenadorEncendido = true;
        System.out.println("El ordenador se ha encendido.");
    }
}

void Estado()
{
    System.out.println("\nEstado del ordenador:" +
                       "\nMarca " + Marca +
                       "\nProcesador " + Procesador +
                       "\nPantalla " + Pantalla + "\n");
    if (OrdenadorEncendido == true) // si el ordenador está encendido...
        System.out.println("El ordenador está encendido.");
    else // si no está encendido...
        System.out.println("El ordenador está apagado.");
}
}

```

La aplicación *CMiOrdenador.java* que acabamos de completar tiene dos clases: la clase aplicación *CMiOrdenador* y la clase *COrdenador*. Observe que la clase aplicación es pública (**public**) y la otra no. Cuando incluyamos varias clases en un fichero fuente, sólo una puede ser pública y su nombre debe coincidir con el del fichero donde se guardan. Al compilar este fichero, Java creará tanto ficheros *.class* como clases separadas hay.

Según lo expuesto hasta ahora, en esta nueva versión también tenemos un fichero, el que almacena la aplicación, que tiene el mismo nombre que la clase que incluye el método **main**, que es por donde se empezará a ejecutar la aplicación.

CARACTERÍSTICAS DE LA POO

Las características fundamentales de la POO son: *abstracción, encapsulamiento, herencia y polimorfismo*.

Abstracción

Por medio de la abstracción conseguimos no detenernos en los detalles concretos de las cosas que no interesen en cada momento, sino generalizar y centrarse en los aspectos que permitan tener una visión global del problema. Por ejemplo, el estudio de un ordenador podemos realizarlo a nivel de funcionamiento de sus circuitos electrónicos, en términos de corriente, tensión, etc., o a nivel de transferencia entre registros, centrándose así el estudio en el flujo de información entre las unidades que lo componen (memoria, unidad aritmética, unidad de control, registros,

etc.), sin importarnos el comportamiento de los circuitos electrónicos que componen estas unidades.

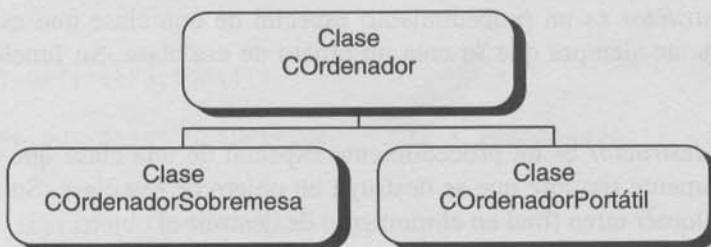
Encapsulamiento

Esta característica permite ver un objeto como una caja negra en la que se ha introducido de alguna manera toda la información relacionada con dicho objeto. Esto nos permitirá manipular los objetos como unidades básicas, permaneciendo oculta su estructura interna.

La abstracción y la encapsulación están representadas por la *clase*. La *clase* es una abstracción, porque en ella se definen las propiedades o atributos de un determinado conjunto de objetos con características comunes, y es una encapsulación porque constituye una caja negra que encierra tanto los datos que almacena cada objeto como los métodos que permiten manipularlos.

Herencia

La herencia permite el acceso automático a la información contenida en otras clases. De esta forma, la reutilización del código está garantizada. Con la herencia todas las clases están clasificadas en una jerarquía estricta. Cada clase tiene su superclase (la clase superior en la jerarquía), y cada clase puede tener una o más subclases (las clases inferiores en la jerarquía).



Las clases que están en la parte inferior en la jerarquía se dice que *heredan* de las clases que están en la parte superior en la jerarquía.

El término heredar significa que las subclases disponen de todos los métodos y propiedades de su superclase. Este mecanismo proporciona una forma rápida y cómoda de extender la funcionalidad de una clase.

En Java cada clase sólo puede tener una superclase, lo que se denomina *herencia simple*. En otros lenguajes orientados a objetos, como C++, las clases pue-

den tener más de una superclase, lo que se conoce como *herencia múltiple*. En este caso, una clase comparte los métodos y propiedades de varias clases. Esta característica, proporciona un poder enorme a la hora de crear clases, pero complica excesivamente la programación, por lo que es de escasa o nula utilización. Java, intentando facilitar las cosas, soluciona este problema de comportamiento compartido utilizando *interfaces*.

Una *interfaz* es una colección de nombres de métodos, sin incluir sus definiciones, que puede ser añadida a cualquier clase para proporcionarla comportamientos adicionales no incluidos en los métodos propios o heredados.

Todo esto será objeto de un estudio amplio en capítulos posteriores.

Polimorfismo

Esta característica permite implementar múltiples formas de un mismo método, dependiendo cada una de ellas de la clase sobre la que se realice la implementación. Esto hace que se pueda acceder a una variedad de métodos distintos (todos con el mismo nombre) utilizando exactamente el mismo medio de acceso. Más adelante, cuando estudie en profundidad las clases y subclases, estará en condiciones de entender con claridad la utilidad de esta característica.

CONSTRUCTORES Y DESTRUCTORES

Un *constructor* es un procedimiento especial de una clase que es llamado automáticamente siempre que se crea un objeto de esa clase. Su función es iniciar el objeto.

Un *destructor* es un procedimiento especial de una clase que es llamado automáticamente siempre que se destruye un objeto de esa clase. Su función es realizar cualquier tarea final en el momento de destruir el objeto.

EJERCICIOS RESUELTOS

Para practicar con una aplicación más, escriba el siguiente ejemplo y pruebe los resultados. Hágalo primero desde la línea de órdenes y después con el entorno de desarrollo integrado preferido por usted. El siguiente ejemplo muestra una clase para representar números racionales. Esta clase puede ser útil porque muchos números no pueden ser representados exactamente utilizando un número fraccionario. Por ejemplo, el número racional $1/3$ representado como un número

fraccionario sería $0,333333$, valor más fácil de manipular, pero a costa de perder precisión. Evidentemente, $1/3 * 3 = 1$, pero $0,333333 * 3 = 0,999999$.

Pensando en un número racional como si de un objeto se tratara, es fácil deducir que sus atributos son dos: el *numerador* y el *denominador*. Y los métodos aplicables sobre los números racionales son numerosos: suma, resta, multiplicación, simplificación, etc. Pero en base a los conocimientos adquiridos, sólo añadiremos dos métodos sencillos: uno, *AsignarDatos*, para establecer los valores del numerador y del denominador; y otro, *VisualizarRacional*, para visualizar un número racional.

Edición

Abra el procesador de textos o el editor de su entorno integrado y edite la aplicación propuesta, como se muestra a continuación:

```
class CRacional
{
    int Numerador;
    int Denominador;

    void AsignarDatos(int num, int den)
    {
        Numerador = num;
        if (den == 0) den = 1; // el denominador no puede ser cero
        Denominador = den;
    }

    void VisualizarRacional()
    {
        System.out.println(Numerador + "/" + Denominador);
    }

    public static void main (String[] args)
    {
        // Punto de entrada a la aplicación
        CRacional r1 = new CRacional(); // crear un objeto CRacional

        r1.AsignarDatos(2, 5);
        r1.VisualizarRacional();
    }
}
```

Una vez editado el programa, guárdelo en el disco con el nombre *CRacional.java*.

¿Qué hace esta aplicación?

Fijándonos en el método principal, **main**, vemos que se ha declarado un objeto *r1* de la clase *CRacional*.

```
CRacional r1 = new CRacional();
```

En el siguiente paso se envía el mensaje *AsignarDatos* al objeto *r1*. El objeto responde a este mensaje ejecutando su método *AsignarDatos* que almacena el valor 2 en su numerador y el valor 5 en su denominador; ambos valores han sido pasados como argumentos.

```
r1.AsignarDatos(2, 5);
```

Finalmente, se envía el mensaje *VisualizarRacional* al objeto *r1*. El objeto responde a este mensaje ejecutando su método *VisualizarRacional* que visualiza sus atributos numerador y denominador en forma de quebrado; en nuestro caso, el número racional 2/5.

```
r1.VisualizarRacional();
```

Para finalizar, compile, ejecute la aplicación y observe que el resultado es el esperado.

EJERCICIOS PROPUESTOS

1. Añada a la aplicación *COrdenador.java* el método *ApagarOrdenador*.
2. Diseñe una clase *CCoche* que represente coches. Incluya los atributos *marca*, *modelo* y *color*; y los métodos que simulen, enviando mensajes, las acciones de arrancar el motor, cambiar de velocidad, acelerar, frenar y parar el motor.

CAPÍTULO 3

© F.J.Ceballos/RA-MA

ELEMENTOS DEL LENGUAJE

En este capítulo veremos los elementos que aporta Java (caracteres, secuencias de escape, tipos de datos, operadores, etc.) para escribir un programa. El introducir este capítulo ahora es porque dichos elementos los tenemos que utilizar desde el principio; algunos ya han aparecido en los ejemplos del capítulo 1 y 2. Consideré este capítulo como soporte para el resto de los capítulos; esto es, lo que se va a exponer en él, lo irá utilizando en menor o mayor medida en los capítulos sucesivos. Por lo tanto, límítense ahora simplemente a realizar un estudio con el fin de informarse de los elementos con los que contamos.

PRESENTACIÓN DE LA SINTAXIS DE JAVA

Las palabras clave aparecerán en negrita y cuando se utilicen deben escribirse exactamente como aparecen. En cambio, el texto que aparece en cursiva, significa que ahí debe ponerse la información indicada por ese texto. Por ejemplo:

```
if (expresión booleana)
    sentencia(s) a ejecutar si la expresión booleana es verdad;
else
    sentencia(s) a ejecutar si la expresión booleana es falsa;
```

Los corchetes “[]” indican que la información encerrada entre ellos es opcional, y los puntos suspensivos “...” que pueden aparecer más elementos de la misma forma. Por ejemplo, la sintaxis para definir una constante es:

```
final static tipo idctel = cte1[. idcte2 = cte2]...;
```

Cuando dos o más opciones aparecen entre llaves “{ }” separadas por “|”, se elige una, la necesaria para la expresión que se desea construir. Por ejemplo:

```
constante_entera[{1|L}].
```

CARACTERES DE JAVA

Los caracteres de Java pueden agruparse en letras, dígitos, espacios en blanco, caracteres especiales, signos de puntuación y secuencias de escape.

Letras, dígitos y otros

Estos caracteres son utilizados para formar las *constantes*, los *identificadores* y las *palabras clave* de Java. Son los siguientes:

- Letras mayúsculas de los alfabetos internacionales:
A - Z (son válidas las letras acentuadas y la Ñ)
- Letras minúsculas de los alfabetos internacionales:
a - z (son válidas las letras acentuadas y la ñ)
- Dígitos de los alfabetos internacionales, entre los que se encuentran:
0 1 2 3 4 5 6 7 8 9
- Caracteres: “_”, “\$” y cualquier carácter Unicode por encima de 00C0.

El compilador Java trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo los identificadores *Año* y *año* son diferentes.

Espacios en blanco

Los caracteres espacio en blanco (ASCII SP), tabulador horizontal (ASCII HT), avance de página (ASCII FF), nueva línea (ASCII LF), retorno de carro (ASCII CR) o CR LF (estos dos caracteres son considerados como uno solo: \n), son caracteres denominados *espacios en blanco*, porque la labor que desempeñan es la misma que la del espacio en blanco: actuar como separadores entre los elementos de un programa, lo cual permite escribir programas más legibles. Por ejemplo, el siguiente código:

```
public static void main (String[] args) { System.out.print(  
"Hola, qué tal estás.\n"); }
```

puede escribirse de una forma más legible así:

```
public static void main (String[] args)  
{  
    System.out.print("Hola, qué tal estás.\n");  
}
```

Los espacios en blanco en exceso son ignorados por el compilador. Por ejemplo, el código siguiente se comporta exactamente igual que el anterior:

```

    líneas en blanco      espacios en blanco
public static void main (String[] args)
{
    System.out.print ("Hola, qué tal estás.\n");
}
  
```

Caracteres especiales y signos de puntuación

Este grupo de caracteres se utiliza de diferentes formas; por ejemplo, para indicar que un identificador es una función o un array; para especificar una determinada operación aritmética, lógica o de relación, etc. Son los siguientes:

, . ; : ? ' " () [] { } < ! | / \ ~ + % & ^ * - = >

Secuencias de escape

Cualquier carácter de los anteriores puede también ser representado por una *secuencia de escape*. Una secuencia de escape está formada por el carácter \ seguido de una *letra* o de una *combinación de dígitos*. Son utilizadas para acciones como nueva línea, tabular y para hacer referencia a caracteres no imprimibles.

El lenguaje Java tiene predefinidas las siguientes secuencias de escape:

Secuencia	ASCII	Definición
\n	CR+LF	Ir al principio de la siguiente línea
\t	HT	Tabulador horizontal
\b	BS	Retroceso (<i>backspace</i>)
\r	CR	Retorno de carro sin avance de línea
\f	FF	Alimentación de página (sólo para impresora)
\'	'	Comilla simple
\"	"	Comilla doble
\\\	\	Barra invertida (<i>backslash</i>)
\ddd		Carácter ASCII. Representación octal
\udddd		Carácter ASCII. Representación Unicode
\u0007	BEL	Alerta, pitido
\u000B	VT	Tabulador vertical (sólo para impresora)

Observe en el ejemplo anterior la secuencia de escape `\n` en la llamada al método `print`.

TIPOS DE DATOS

Recuerde las operaciones aritméticas que realizaba el programa `Aritmetica.java` que vimos en un capítulo anterior. Por ejemplo, una de las operaciones que realizábamos era la suma de dos valores:

```
dato1 = 20; dato2 = 10; resultado = dato1 + dato2;
```

Para que el compilador Java reconozca esta operación es necesario especificar previamente el tipo de cada uno de los operandos que intervienen en la misma, así como el tipo del resultado. Para ello, escribiremos una línea como la siguiente:

```
int dato1, dato2, resultado;
```

La declaración anterior le dice al compilador Java que `dato1`, `dato2` y `resultado` son de tipo entero (`int`).

Los tipos de datos en Java se clasifican en: tipos *primitivos* y *tipos referenciados*.

Tipos primitivos

Hay ocho tipos primitivos de datos que podemos clasificar en: tipos numéricos y el tipo **boolean**. A su vez, los tipos numéricos se clasifican en tipos enteros y tipos reales.

Tipos enteros: **byte**, **short**, **int**, **long** y **char**.

Tipos reales: **float** y **double**.

Cada tipo primitivo tiene un rango diferente de valores positivos y negativos, excepto el **boolean** que sólo tiene dos valores: **true** y **false**. El tipo de datos que se seleccione para declarar las variables de un determinado programa dependerá del rango y tipo de valores que vayan a almacenar cada una de ellas y de si éstos son enteros o fraccionarios.

Se les llama primitivos porque están integrados en el sistema y en realidad no son objetos, lo cual hace que su uso sea más eficiente. Más adelante veremos también que la biblioteca Java proporciona las clases: **Byte**, **Character**, **Short**, **Integer**, **Long**, **Float**, **Double** y **Boolean**, para encapsular cada uno de los tipos expuestos, proporcionando así una funcionalidad añadida para manipularlos.

byte

El tipo **byte** se utiliza para declarar datos enteros comprendidos entre -128 y $+127$. Un *byte* se define como un conjunto de 8 bits, independientemente de la plataforma en que se ejecute el *código byte* de Java. El siguiente ejemplo declara la variable *b* de tipo **byte** y le asigna el valor inicial 0. Es recomendable iniciar toda variable que se declare.

```
byte b = 0;
```

short

El tipo **short** se utiliza para declarar datos enteros comprendidos entre -32768 y $+32767$. Un valor **short** se define como un dato de 16 bits de longitud, independientemente de la plataforma en la que resida el *código byte* de Java. El siguiente ejemplo declara *i* y *j* como variables enteras de tipo **short**:

```
short i = 0, j = 0;
```

int

El tipo **int** se utiliza para declarar datos enteros comprendidos entre -2147483648 y $+2147483647$. Un valor **int** se define como un dato de 32 bits de longitud, independientemente de la plataforma en la que se ejecute el *código byte* de Java. El siguiente ejemplo declara e inicia tres variables *a*, *b* y *c*, de tipo **int**:

```
int a = 2000;
int b = -30;
int c = 0xF003; /* valor en hexadecimal */
```

En general, el uso de enteros de cualquier tipo produce un código compacto y rápido. Así mismo, podemos afirmar que la longitud de un **short** es siempre menor o igual que la longitud de un **int**.

long

El tipo **long** se utiliza para declarar datos enteros comprendidos entre los valores -9223372036854775808 y $+9223372036854775807$. Un valor **long** se define como un dato de 64 bits de longitud, independientemente de la plataforma en la que se ejecute el *código byte* de Java. El siguiente ejemplo declara e inicia las variables *a*, *b* y *c*, de tipo **long**:

```
long a = -1L; /* L indica que la constante -1 es long */
long b = 125;
```

```
long c = 0x1F00230F; /* valor en hexadecimal */
```

En general, podemos afirmar que la longitud de un **int** es menor o igual que la longitud de un **long**.

char

El tipo **char** es utilizado para declarar datos enteros en el rango `\u0000` a `\xFFFF` en *Unicode* (0 a 65535). Los valores 0 a 127 se corresponden con los caracteres ASCII del mismo código (ver los apéndices). El juego de caracteres ASCII conforman una parte muy pequeña del juego de caracteres *Unicode*.

```
char car = 0;
```

En Java para representar los caracteres se utiliza el código *Unicode*. Se trata de un código de 16 bits (esto es, cada carácter ocupa 2 bytes) con el único propósito de internacionalizar el lenguaje. El código *Unicode* actualmente representa los caracteres de la mayoría de los idiomas escritos conocidos en todo el mundo.

El siguiente ejemplo declara la variable *car* de tipo **char** a la que se le asigna el carácter ‘a’ como valor inicial (observe que hay una diferencia entre ‘a’ y *a*; *a* entre comillas simples es interpretada por el compilador Java como un valor, un carácter, y *a* sin comillas sería interpretada como una variable). Las cuatro declaraciones siguientes son idénticas:

```
char car = 'a';
char car = 97;      /* la 'a' es el decimal 97 */
char car = 0x0061;  /* la 'a' es el hexadecimal 0061 */
char car = '\u0061'; /* la 'a' es el Unicode 0061 */
```

Un carácter es representado internamente por un entero, que puede ser expresado en decimal, hexadecimal u octal, como veremos más adelante.

float

El tipo **float** se utiliza para declarar un dato en coma flotante de 32 bits en el formato IEEE 754 (este formato utiliza 1 bit para el signo, 8 bits para el exponente y 24 para la mantisa). Los datos de tipo **float** almacenan valores con una precisión aproximada de 7 dígitos. Para especificar que una constante (un literal) es de tipo **float**, hay que añadir al final de su valor la letra ‘f’ o ‘F’. El siguiente ejemplo declara las variables *a*, *b* y *c*, de tipo real de precisión simple:

```
float a = 3.14159F;
float b = 2.2e-5F; /* 2.2e-5 = 2.2 por 10 elevado a 5 */
float c = 2/3F; /* 0,6666667 */
```

double

El tipo **double** se utiliza para declarar un dato en coma flotante de 64 bits en el formato IEEE 754 (1 bit para el signo, 11 bits para el exponente y 52 para la mantisa). Los datos de tipo **double** almacenan valores con una precisión aproximada de 16 dígitos. Para especificar explícitamente que una constante (un literal) es de tipo **double**, hay que añadir al final de su valor la letra ‘d’ o ‘D’; por omisión, una constante es considerada de tipo **double**. El siguiente ejemplo declara las variables *a*, *b* y *c*, de tipo real de precisión doble:

```
double a = 3.14159; /* una constante es double por omisión */
double b = 2.2e+5;  /* 2.2e-5 = 2.2 por 10 elevado a 5 */
double c = 2/3D;
```

boolean

El tipo **boolean** se utiliza para indicar si el resultado de la evaluación de una expresión booleana es verdadero o falso. Los dos posibles valores de una expresión booleana son **true** y **false**. Los literales **true** y **false** son constantes definidas como palabras clave en el lenguaje Java. Por tanto, se pueden utilizar las palabras **true** y **false** como valores de retorno, en expresiones condicionales, en asignaciones y en comparaciones con otras variables booleanas.

El contenido de una variable booleana no se puede convertir a otros tipos, pero sí se puede convertir en una cadena de caracteres.

Tipos referenciados

Hay tres clases de tipos referenciados: clases, interfaces y *arrays*. Todos ellos serán objeto de estudio en capítulos posteriores.

LITERALES

Un literal es la expresión de un valor de un tipo primitivo, de un tipo **String** (cadena de caracteres) o la expresión **null** (valor nulo o desconocido). Por ejemplo, son literales: 5, 3.14, ‘a’, “*hola*” y **null**. En realidad son valores constantes.

Un literal en Java puede ser: un entero, un real, un valor booleano, un carácter, una cadena de caracteres y un valor nulo.

Literales enteros

El lenguaje Java permite especificar un literal entero en base 10, 8 y 16.

En general, el signo + es opcional si el valor es positivo y el signo – estará presente siempre que el valor sea negativo. Un literal entero es de tipo **int** a no ser que su valor absoluto sea mayor que el de un **int** o se especifique el sufijo **l** o **L**, en cuyo caso será **long**. Lo expuesto queda resumido en la línea siguiente:

`{[+]|-}literal_entero[{l|L}]`

Un *literal entero decimal* puede tener uno o más dígitos del 0 a 9, de los cuales el primero de ellos es distinto de 0. Por ejemplo:

4326 constante entera int
4326L constante entera long
3426000000 constante entera long

Un *literal entero octal* puede tener uno o más dígitos del 0 a 7, precedidos por 0 (cero). Por ejemplo:

0326 constante entera int en base 8

Un *literal entero hexadecimal* puede tener uno o más dígitos del 0 a 9 y letras de la A a la F (en mayúsculas o en minúsculas) precedidos por 0x o 0X (cero seguido de x). Por ejemplo:

256 número decimal 256
0400 número decimal 256 expresado en octal
0x100 número decimal 256 expresado en hexadecimal
-0400 número decimal -256 expresado en octal
-0x100 número decimal -256 expresado en hexadecimal

Literales reales

Un literal real está formado por una *parte entera*, seguido por un *punto decimal*, y una *parte fraccionaria*. También se permite la notación científica, en cuyo caso se añade al valor una e o E, seguida por un exponente positivo o negativo.

`{[+]|-}parte-entera.parte-fraccionaria[{e|E}{[+]|-}exponente]`

donde *exponente* representa cero o más dígitos del 0 al 9 y E o e es el símbolo de exponente de la base 10 que puede ser positivo o negativo ($2E-5 = 2 \times 10^{-5}$). Si

la constante real es positiva no es necesario especificar el signo y si es negativa lleva el signo menos (-). Por ejemplo:

```
-17.24
17.244283
.008e3
27E-3
```

Una constante real tiene siempre tipo **double**, a no ser que se añada a la misma una *f* o *F*, en cuyo caso será de tipo **float**. Por ejemplo:

```
17.24F      constante real de tipo float
```

También se pueden utilizar los sufijos *d* o *D* para especificar explícitamente que se trata de una constante de tipo **double**. Por ejemplo:

```
17.24D      constante real de tipo double
```

Literales de un solo carácter

Los literales de un solo carácter son de tipo **char**. Este tipo de literales está formado por un único carácter encerrado entre *comillas simples*. Una secuencia de escape es considerada como un único carácter. Algunos ejemplos son:

' '	espacio en blanco
'x'	letra minúscula x
'\n'	retorno de carro más avance de línea
'\u0007'	pítido
'\u001B'	carácter ASCII Esc

El valor de una constante de un solo carácter es el valor que le corresponde en el juego de caracteres de la máquina.

Literales de cadenas de caracteres

Un literal de cadena de caracteres es una secuencia de caracteres encerrados entre *comillas dobles* (incluidas las secuencias de escape como \""). Por ejemplo:

```
"Esto es una constante de caracteres"
"3.1415926"
"Paseo de Pereda 10, Santander"
""                      /* cadena vacía */
"Lenguaje \"Java\" "    /* produce: Lenguaje "Java" */
```

En el ejemplo siguiente el carácter \n fuerza a que la cadena “*O pulse Entrar*” se escriba en una nueva línea:

```
System.out.print("Escriba un número entre 1 y 5\n0 pulse Entrar");
```

Las cadenas de caracteres en Java son objetos de la clase **String** que estudiaremos más adelante. Esto es, cada vez que en un programa se utilice un literal de caracteres, Java crea de forma automática un objeto **String** con el valor del literal.

Las cadenas de caracteres se pueden concatenar (unir) empleando el operador +. Por ejemplo, la siguiente sentencia concatena las cadenas “*Distancia:*”, *distan-**cia*, y “*Km.*”.

```
System.out.println("Distancia: " + distancia + " Km.");
```

Si alguna de las expresiones no se corresponde con una cadena, como se supone que ocurre con *distancia*, Java la convierte de forma automática en una cadena de caracteres. Más adelante aprenderá el porqué de esto.

IDENTIFICADORES

Los identificadores son nombres dados a tipos, literales, variables, clases, interfaces, métodos, paquetes y sentencias de un programa. La sintaxis para formar un identificador es la siguiente:

$$(\text{letra}|_{$})[(\text{letra}|\text{dígito}|_|$)]\dots$$

lo cual indica que un identificador consta de uno o más caracteres (véase el apartado anterior “Letras, dígitos y otros”) y que el *primer carácter* debe ser una *letra*, el *carácter de subrayado* o el *carácter dólar* (\$). No pueden comenzar por un dígito ni pueden contener caracteres especiales (véase el apartado anterior “Caracteres especiales”).

Las letras pueden ser mayúsculas o minúsculas. Para Java una letra mayúscula es un carácter diferente a esa misma letra en minúscula. Por ejemplo, los identificadores *Suma*, *suma* y *SUMA* son diferentes.

Los identificadores pueden tener cualquier número de caracteres. Algunos ejemplos son:

```
Suma
Cálculo_Números_Primos
$ordenar
VisualizarDatos
```

PALABRAS CLAVE

Las palabras clave son identificadores predefinidos que tienen un significado especial para el compilador Java. Por lo tanto, un identificador definido por el usuario, no puede tener el mismo nombre que una palabra clave. El lenguaje Java, tiene las siguientes palabras clave:

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

Las palabras clave deben escribirse siempre en minúsculas, como están.

COMENTARIOS

Un comentario es un mensaje a cualquiera que lea el código fuente. Añadiendo comentarios se hace más fácil la comprensión de un programa. La finalidad de los comentarios es explicar el código fuente. Java soporta tres tipos de comentarios:

- *Comentario tradicional.* Un comentario tradicional empieza con los caracteres /* y finaliza con los caracteres */. Estos comentarios pueden ocupar más de una línea, pero no pueden anidarse. Por ejemplo:

```
/*
 * La ejecución del programa comienza con el método main().
 * La llamada al constructor de la clase no tiene lugar a menos
 * que se cree un objeto del tipo 'CElementosJava'
 * en el método main().
 */
```

- *Comentario de una sola línea.* Este tipo de comentario comienza con una doble barra // y se extiende hasta el final de la línea. Por ejemplo:

```
// Agregar aquí el código de iniciación
```

- *Comentario de documentación.* Este tipo de comentario comienza con /** y termina con */. Son comentarios especiales que javadoc utiliza para generar la

documentación acerca del programa, aunque también se pueden emplear de manera idéntica a los comentarios tradicionales.

```
/**  
 * Punto de entrada principal para la aplicación.  
 *  
 * Parámetros:  
 * args: Matriz de parámetros pasados a la aplicación  
 * a través de la línea de órdenes.  
 */
```

DECLARACIÓN DE CONSTANTES SIMBÓLICAS

Declarar una constante simbólica significa decirle al compilador Java el nombre de la constante y su valor. Esto se hace utilizando el calificador **final** y/o el **static**.

```
class CElementosJava  
{  
    final static int cte1 = 1;  
    final static String cte2 = "Pulse una tecla para continuar";  
  
    void Test()  
    {  
        final double cte3 = 3.1415926;  
        // ...  
    }  
    // ...  
}
```

Como se observa en el ejemplo anterior, declarar una constante simbólica supone anteponer el calificador **final**, o bien los calificadores **final** y **static**, al tipo y nombre de la constante, que será iniciada con el valor deseado. Distinguimos dos casos: que la constante esté definida en el cuerpo de la clase, fuera de todo método, como sucede con *cte1* y *cte2*, o que esté definida dentro de un método, como sucede con *cte3*. En el primer caso, la constante puede estar calificada, además de con **final**, con **static**; en este caso, sólo existirá una copia de la constante para todos los objetos que se declaren de esa clase (en nuestro caso, la clase es *CElementosJava*). Si no se especifica **static**, cada objeto incluiría su propia copia de la constante; es claro que esta forma de proceder no parece lógica por tratarse de la misma constante, razón por la que no se hace uso de ella. En el segundo caso no se puede utilizar **static**, la constante sólo es visible dentro del método, y sólo existe durante la ejecución del mismo; en este caso se dice que la constante es *local* al método. Una constante local no pueden ser declarada **static**.

Una vez que se haya declarado una constante, por definición, no se le puede asignar otro valor. Por ello, cuando se declara una constante debe ser iniciada con un valor. Por ejemplo, después de haber declarado *cte3* según se muestra en el ejemplo anterior, una sentencia como la siguiente daría lugar a un error:

```
cte3 = 3.14;
```

¿Por qué utilizar constantes?

Utilizando constantes es más fácil modificar un programa. Por ejemplo, supongamos que un programa utiliza N veces una constante de valor 3.14. Si hemos definido dicha constante como *final static double Pi = 3.14* y posteriormente necesitamos cambiar el valor de la misma a 3.1416, sólo tendremos que modificar una línea, la que define la constante. En cambio, si no hemos declarado *Pi*, sino que hemos utilizado el valor 3.14 directamente N veces, tendríamos que realizar N cambios.

DECLARACIÓN DE UNA VARIABLE

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo. El valor de una variable, a diferencia de una constante, puede cambiar durante la ejecución de un programa. Para utilizar una variable en un programa, primero hay que declararla. La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo:

```
tipo identificador[, identificador]...
```

En el ejemplo siguiente se declaran tres variables de tipo **short**, una variable de tipo **int**, y dos variables de tipo **String**:

```
class CElementosJava
{
    short dia, mes, año;

    void Test()
    {
        int contador = 0;
        String Nombre = "", Apellidos = "";
        dia = 20;
        Apellidos = "Ceballos";
        // ...
    }
    // ...
}
```

El tipo, primitivo o referenciado, determina los valores que puede tomar la variable así como las operaciones que con ella pueden realizarse. Los operadores serán expuestos un poco más adelante.

Por definición, una variable declarada dentro de un bloque, entendiendo por bloque el código encerrado entre los caracteres '{' y '}', es accesible directamente, esto es, sin un objeto, sólo dentro de ese bloque. Más adelante, cuando tratemos con objetos matizaremos el concepto de accesibilidad.

Según la definición anterior, las variables *día*, *mes* y *año* son accesibles desde todos los métodos no **static** de la clase *CElementosJava*. Estas variables, declaradas en el bloque de la clase pero fuera de cualquier otro bloque, se denominan *variables miembro* de la clase (atributos de la clase).

En cambio, las variables *contador*, *Nombre* y *Apellidos* han sido declaradas en el bloque de código correspondiente al cuerpo del método *Test*. Por lo tanto, aplicando la definición anterior, sólo serán accesibles en este bloque. En este caso se dice que dichas variables son *locales* al bloque donde han sido declaradas. Una variable local se crea cuando se ejecuta el bloque donde se declara y se destruye cuando finaliza la ejecución de dicho bloque; dicho de otra forma, una variable local se destruye cuando el flujo de ejecución sale fuera del ámbito de la variable. Una variable local no puede ser declarada **static**.

Iniciación de una variable

Las variables miembro de una clase son iniciadas por omisión por el compilador Java para cada objeto que se declare de la misma: las variables numéricas con 0, los caracteres con '0' y las referencias a las cadenas de caracteres y el resto de las referencias a otros objetos con **null**. También pueden ser iniciadas explícitamente. En cambio, las variables locales no son iniciadas por el compilador Java. Por lo tanto, es nuestra obligación iniciarlas, de lo contrario el compilador visualizará un mensaje de error en todas las sentencias que hagan referencia a esas variables.

```
class CElementosJava
{
    short var1;
    void Test()
    {
        int var2;
        System.out.println(var2);      // error: variable no iniciada
        System.out.println(var1);      // correcto: var1 es igual a 0
    }
    // ...
}
```

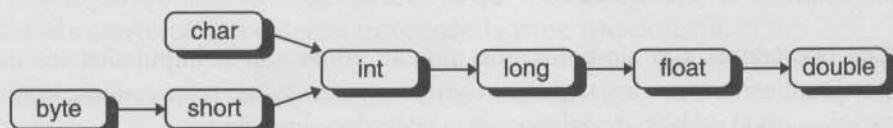
EXPRESIONES NUMÉRICAS

Una expresión es un conjunto de operandos unidos mediante operadores para especificar una operación determinada. Todas las expresiones cuando se evalúan retornan un valor. Por ejemplo:

```
a + 1
suma + c
cantidad * precio
7 * Math.sqrt(a) - b / 2      (sqrt indica raíz cuadrada)
```

CONVERSIÓN ENTRE TIPOS DE DATOS

Cuando Java tiene que evaluar una expresión en la que intervienen operandos de diferentes tipos, primero convierte, sólo para realizar las operaciones solicitadas, los valores de los operandos al tipo del operando cuya precisión sea más alta. Cuando se trate de una asignación, convierte el valor de la derecha al tipo de la variable de la izquierda siempre que no haya pérdida de información. En otro caso, Java exige que la conversión se realice explícitamente. La figura siguiente resume los tipos colocados de izquierda a derecha de menos a más precisos; las flechas indican las conversiones implícitas permitidas:



```
// Conversión implícita
byte bDato = 1; short sDato = 0; int iDato = 0; long lDato = 0;
float fDato = 0; double dDato = 0;

sDato = bDato;
iDato = sDato;
lDato = iDato;
fDato = lDato;
dDato = fDato + lDato - iDato * sDato / bDato;
System.out.println(dDato); // resultado: 1.0
```

Java permite una conversión explícita (conversión forzada) del tipo de una expresión mediante una construcción denominada *cast*, que tiene la forma:

(tipo) expresión

Cualquier valor de un tipo entero o real puede ser convertido a o desde cualquier tipo numérico. No se pueden realizar conversiones entre los tipos enteros o reales y el tipo **boolean**. Por ejemplo:

```
// Conversión explícita (cast)
byte bDato = 0; short sDato = 0; int iDato = 0; long lDato = 0;
float fDato = 0; double dDato = 2;

fDato = (float)dDato;
lDato = (long)fDato;
iDato = (int)lDato;
sDato = (short)iDato;
bDato = (byte)(sDato + iDato - lDato * fDato / dDato);
System.out.println(bDato); // resultado: 2
```

La expresión es convertida al tipo especificado si esa conversión está permitida; en otro caso, se obtendrá un error. La utilización apropiada de construcciones *cast* garantiza una evaluación consistente, pero siempre que se pueda, es mejor evitarla ya que suprime la verificación de tipo proporcionada por el compilador y por consiguiente puede conducir a resultados inesperados, o cuando menos, a una pérdida de precisión en el resultado. Por ejemplo:

```
float r;
r = (float)Math.sqrt(10); // el resultado se redondea perdiendo
                         // precisión ya que sqrt devuelve un
                         // valor de tipo double
```

OPERADORES

Los operadores son símbolos que indican cómo son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, relacionales, lógicos, unitarios, a nivel de bits, de asignación y operador condicional.

Operadores aritméticos

Los operadores aritméticos los utilizamos para realizar operaciones matemáticas y son los siguientes:

Operador	Operación
+	<i>Suma</i> . Los operandos pueden ser enteros o reales.
-	<i>Resta</i> . Los operandos pueden ser enteros o reales.
*	<i>Multiplicación</i> . Los operandos pueden ser enteros o reales.
/	<i>División</i> . Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resultado es entero. En el resto de los casos el resultado es real.
%	<i>Módulo</i> o resto de una división entera. Los operandos tienen que ser enteros.

El siguiente ejemplo muestra cómo utilizar estos operadores. Como ya hemos venido diciendo, observe que primero se declaran las variables y después se realizan las operaciones deseadas con ellas.

```
int a = 10, b = 3, c;
float x = 2.0F, y;
y = x + a;           // El resultado es 12.0 de tipo float
c = a / b;           // El resultado es 3 de tipo int
c = a % b;           // El resultado es 1 de tipo int
y = a / b;           // El resultado es 3 de tipo int. Se
                     // convierte a float para asignarlo a y
c = (int)(x / y); // El resultado es 0.6666667 de tipo float. Se
                     // convierte a int para asignarlo a c (c = 0)
```

Cuando en una operación aritmética los operandos son de diferentes tipos, ambos son convertidos al tipo del operando de precisión más alta. Por ejemplo, para realizar la suma $x+a$ el valor del entero a es convertido a **float**, tipo de x . No se modifica a , sino que su valor es convertido a **float** sólo para realizar la suma. Los tipos **short** y **byte** son convertidos de manera automática a **int**.

En una asignación, el resultado obtenido en una operación aritmética es convertido implícita o explícitamente al tipo de la variable que almacena dicho resultado (véase “Conversión entre tipos de datos”). Por ejemplo, del resultado de x/y sólo la parte entera es asignada a c , ya que c es de tipo **int**. Esto indica que los reales son convertidos a enteros, truncando la parte fraccionaria.

Un resultado real es redondeado independientemente del valor de la primera cifra decimal suprimida. Observe la operación x/y para x igual a 2 e y igual a 3. El resultado es 0.6666667 en lugar de 0.6666666.

Operadores de relación

Los operadores de relación o de comparación permiten evaluar la igualdad y la magnitud. El resultado de una operación de relación es un valor booleano **true** o **false**. Los operadores de relación son los siguientes:

Operador	Operación
<	¿Primer operando <i>menor que</i> el segundo?
>	¿Primer operando <i>mayor que</i> el segundo?
<=	¿Primer operando <i>menor o igual que</i> el segundo?
>=	¿Primer operando <i>mayor o igual que</i> el segundo?
!=	¿Primer operando <i>distinto que</i> el segundo?
==	¿Primer operando <i>igual que</i> el segundo?

Los operandos tiene que ser de un tipo primitivo. Por ejemplo:

```
int x = 10, y = 0;
boolean r;

r = x == y;    // r = false
r = x > y;    // r = true
r = x != y;   // r = true
```

Un operador de relación equivale a una pregunta relativa a cómo son dos operandos entre sí. Por ejemplo, la expresión `x==y` equivale a la pregunta ¿*x* es igual a *y*? Una respuesta sí equivale a un valor verdadero (**true**) y una respuesta no equivale a un valor falso (**false**).

Operadores lógicos

El resultado de una operación lógica (AND, OR, XOR y NOT) es un valor booleano verdadero o falso (**true** o **false**). Las expresiones que dan como resultado valores booleanos (véanse los operadores de relación) pueden combinarse para formar expresiones *booleanas* utilizando los operadores lógicos indicados a continuación. Los operandos deben ser expresiones que den un resultado **boolean**.

Operador	Operación
<code>&&</code> o <code>&</code>	<i>AND</i> . Da como resultado true si al evaluar cada uno de los operandos el resultado es true . Si uno de ellos es false , el resultado es false . Si se utiliza <code>&&</code> (no <code>&</code>) y el primer operando es false , el segundo operando no es evaluado.
<code> </code> o <code>l</code>	<i>OR</i> . El resultado es false si al evaluar cada uno de los operandos el resultado es false . Si uno de ellos es true , el resultado es true . Si se utiliza <code> </code> (no <code>l</code>) y el primer operando es true , el segundo operando no es evaluado (el carácter <code>l</code> es el ASCII 124).
<code>!</code>	<i>NOT</i> . El resultado de aplicar este operador es false si al evaluar su operando el resultado es true , y true en caso contrario.
<code>^</code>	<i>XOR</i> . Da como resultado true si al evaluar cada uno de los operandos el resultado de uno es true y el del otro false ; en otro caso el resultado es false .

El resultado de una operación lógica es de tipo **boolean**. Por ejemplo:

```
int p = 10, q = 0;
boolean r;

r = p != 0 && q != 0; // r = false
```

```
r = p != 0 || q > 0;      // r = true
r = q < p && p <= 10;    // r = true
r = !r;                   // si r = true, entonces r = false
```

Operadores unitarios

Los operadores unitarios se aplican a un solo operando y son los siguientes: `!`, `-`, `~`, `++` y `--`. El operador `!` ya lo hemos visto y los operadores `++` y `--` los veremos más adelante.

Operador	Operación
<code>~</code>	Complemento a 1 (cambiar ceros por unos y unos por ceros). El carácter <code>~</code> es el ASCII 126. El operando debe de ser de un tipo primitivo entero.
<code>-</code>	Cambia de signo al operando (esto es, se calcula el complemento a dos que es el complemento a 1 más 1). El operando puede ser de un tipo primitivo entero o real.

El siguiente ejemplo muestra cómo utilizar estos operadores:

```
int a = 2, b = 0, c = 0;

c = -a;    // resultado c = -2
c = ~b;    // resultado c = -1
```

Operadores a nivel de bits

Estos operadores permiten realizar con sus operandos las operaciones AND, OR, XOR y desplazamientos, bit por bit. Los operandos tienen que ser enteros.

Operador	Operación
<code>&</code>	Operación AND a nivel de bits.
<code> </code>	Operación OR a nivel de bits (carácter ASCII 124).
<code>^</code>	Operación XOR a nivel de bits.
<code><<</code>	Desplazamiento a la izquierda rellenando con ceros por la derecha.
<code>>></code>	Desplazamiento a la derecha rellenando con el bit de signo por la izquierda.
<code>>>></code>	Desplazamiento a la derecha rellenando con ceros por la izquierda.

Los operandos tienen que ser de un tipo primitivo entero.

```

int a = 255, r = 0, m = 32;

r = a & 017; // r=15. Pone a cero todos los bits de a
               // excepto los 4 bits de menor peso.
r = r | m;   // r=47. Pone a 1 todos los bits de r que
               // estén a 1 en m.
r = a & ~07; // r=248. Pone a 0 los 3 bits de menor peso de a.
r = a >> 7;  // r=1. Desplazamiento de 7 bits a la derecha.
r = m << 1;  // r=64. Equivale a r = m * 2
r = m >> 1;  // r=16. Equivale a r = m / 2

```

Operadores de asignación

El resultado de una operación de asignación es el valor almacenado en el operando izquierdo, lógicamente después de que la asignación se ha realizado. El valor que se asigna es convertido implícitamente o explícitamente al tipo del operando de la izquierda (véase el apartado “Conversión entre tipos de datos”). Incluimos aquí los operadores de incremento y decremento porque implícitamente estos operadores realizan una asignación sobre su operando.

Operador	Operación
<code>++</code>	Incremento.
<code>--</code>	Decremento.
<code>=</code>	Asignación simple.
<code>*=</code>	Multiplicación más asignación.
<code>/=</code>	División más asignación.
<code>%=</code>	Módulo más asignación.
<code>+=</code>	Suma más asignación.
<code>-=</code>	Resta más asignación.
<code><<=</code>	Desplazamiento a izquierdas más asignación.
<code>>>=</code>	Desplazamiento a derechas más asignación.
<code>>>></code>	Desplazamiento a derechas más asignación rellenando con ceros.
<code>&=</code>	Operación AND sobre bits más asignación.
<code> =</code>	Operación OR sobre bits más asignación.
<code>^=</code>	Operación XOR sobre bits más asignación.

Los operandos tienen que ser de un tipo primitivo. A continuación se muestran algunos ejemplos con estos operadores.

```

int x = 0, n = 10, i = 1;
n++;           // Incrementa el valor de n en 1.

```

```

++n;           // Incrementa el valor de n en 1.
x = ++n;       // Incrementa n en 1 y asigna el resultado a x.
x = n++;        // Asigna el valor de n a x y después
                 // incrementa n en 1.
i += 2;         // Realiza la operación i = i + 2.
x *= n - 3;     // Realiza la operación x = x * (n-3) y no
                 // x = x * n - 3.
n >>= 1;        // Realiza la operación n = n >> 1 la cual desplaza
                 // el contenido de n 1 bit a la derecha.

```

El operador de incremento incrementa su operando independientemente de que se utilice como sufijo o como prefijo; esto es, $n++$ y $++n$ producen el mismo resultado. Ídem para el operador de decremento.

Ahora bien, cuando el resultado de una operación de incremento se asigna a una variable, como se puede observar en $x = ++n$ y $x = n++$, si el operador de incremento se utiliza como prefijo primero se realiza la operación de incremento y después la asignación; y si se utiliza como sufijo, primero se realiza la operación de asignación y después la de incremento. Ídem para el operador de decremento.

Operador condicional

El operador condicional (`?:`), llamado también operador ternario, se utiliza en expresiones condicionales, que tienen la forma siguiente:

$$\text{operando1} ? \text{operando2} : \text{operando3}$$

La expresión `operando1` debe ser una expresión booleana. La ejecución se realiza de la siguiente forma:

- Si el resultado de la evaluación de `operando1` es **true**, el resultado de la expresión condicional es `operando2`.
- Si el resultado de la evaluación de `operando1` es **false**, el resultado de la expresión condicional es `operando3`.

El siguiente ejemplo asigna a `mayor` el resultado de $(a > b) ? a : b$, que será `a` si `a` es mayor que `b` y `b` si `a` no es mayor que `b`.

```

double a = 10.2, b = 20.5, mayor = 0;
mayor = (a > b) ? a : b;

```

PRIORIDAD Y ORDEN DE EVALUACIÓN

La tabla que se presenta a continuación, resume las reglas de prioridad y asociatividad de todos los operadores. Las líneas se han colocado de mayor a menor prioridad. Los operadores escritos sobre una misma línea tienen la misma prioridad.

Una expresión entre paréntesis, siempre se evalúa primero. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos.

Operador	Asociatividad
() [] .	izquierda a derecha
- ~ ! ++ --	derecha a izquierda
new (<i>tipo</i>) <i>expresión</i>	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >> >>>	izquierda a derecha
< <= > >= instanceof	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
	izquierda a derecha
&&	izquierda a derecha
	izquierda a derecha
?:	derecha a izquierda
= *= /= %= += -= <<= >>= >>>= &= = ^=	derecha a izquierda

En Java, todos los operadores binarios excepto los de asignación son evaluados de izquierda a derecha. En el siguiente ejemplo, primero se asigna *z* a *y* y a continuación *y* a *x*.

```
int x = 0, y = 0, z = 15;
x = y = z;      // resultado x = y = z = 15
```

EJERCICIOS RESUELTOS

La siguiente aplicación utiliza objetos de una clase *CEcuacion* para evaluar ecuaciones de la forma:

$$ax^3 + bx^2 + cx + d$$

Una ecuación se puede ver como un objeto que envuelve el exponente, los coeficientes y los métodos que permitan manipularla. Para hacer sencillo el ejemplo que tratamos de exponer, el exponente lo suponemos fijo de valor 3, los coeficientes serán variables, y añadiremos dos métodos: uno que permita establecer la ecuación con la que deseamos trabajar y otro que permita evaluarla para un valor de x dado. Resumiendo, los objetos *CEcuacion* tendrán unos atributos que serán los coeficientes y unos métodos *Ecuación* y *ValorPara* para manipularlos.

El método *Ecuación* simplemente asignará los valores pasados como argumentos a los atributos representativos de los coeficientes de la ecuación.

El método *ValorPara* evaluará la ecuación para el valor de x pasado como argumento. Este método, utilizando la sentencia **return**, devolverá como resultado el valor calculado. Observe que el tipo devuelto por el método es **double**:

```

public double ValorPara(double x)
{
    double resultado;
    // Realizar cálculos
    return resultado; ← Valor devuelto por
}                                     el método

```

Han aparecido algunos conceptos nuevos (argumentos pasados a un método y valor retornado por un método). No se preocupe, sólo se trata de un primer contacto. Más adelante estudiaremos todo esto con mayor profundidad. Para una mejor compresión de lo dicho, piense en el método o función llamado *logaritmo* que seguro habrá utilizado más de una vez a lo largo de sus estudios. Este método devuelve un valor real correspondiente al logaritmo del valor pasado como argumento: $x = \log(y)$. Bueno, pues compárelo con el método *ValorPara* y comprobará que estamos hablando de métodos análogos.

Según lo expuesto y aplicando los conocimientos adquiridos en el capítulo 2, escribamos en primer lugar la clase *CEcuacion* como se muestra a continuación. Observe que no es pública.

```

class CEcuacion
{
    // El término de mayor grado tiene exponente 3 fijo
    double c3, c2, c1, c0; // coeficientes
    public void Ecuación(double a, double b, double c, double d)
    {
        c3 = a; c2 = b; c1 = c; c0 = d;
    }
}

```

```

public double ValorPara(double x)
{
    double resultado;
    resultado = c3*x*x*x + c2*x*x + c1*x + c0;
    return resultado; // devolver el valor calculado
}
}

```

El siguiente paso es añadir al mismo fichero fuente una clase aplicación pública que utilice la clase de objetos *CEcuacion*. Esta clase aplicación puede ser de la forma siguiente:

```

public class CMiAplicacion
{
    public static void main(String[] args)
    {
        CEcuacion ecl = new CEcuacion();
        ecl.Ecuación(1, -3.2, 0, 7);

        double r = ecl.ValorPara(1);
        System.out.println(r);

        r = ecl.ValorPara(1.5);
        System.out.println(r);
    }
}

```

Recuerde que el método **main**; es por donde empieza a ejecutarse la aplicación. Este método crea un objeto *ecl* de la clase *CEcuacion*, envía al objeto *ecl* el mensaje *Ecuación* para establecer los coeficientes de la ecuación y a continuación le envía el mensaje *ValorPara* con el objetivo de evaluar la ecuación para el valor de *x* pasado como argumento.

Una vez escrita la aplicación debe guardarla con el nombre *CMiAplicacion.java* (nombre de la clase pública) y compilarla. Después puede ejecutarla y observar los resultados. Incluso puede atreverse a evaluar otras ecuaciones para distintos valores de *x*.

EJERCICIOS PROPUESTOS

1. Escriba una aplicación que visualice en el monitor los siguientes mensajes:

Bienvenido al mundo de Java.
Podrás dar solución a muchos problemas.

2. ¿Qué resultados se obtienen al realizar las operaciones siguientes? Si hay errores en la compilación, corríjalos y dé una explicación de por qué suceden.

```
int a = 10, b = 3, c = 1, d, e;
float x, y;
x = a / b;
c = a < b && c;
d = a + b++;
e = ++a - b;
y = (float)a / b;
```

3. Escriba las sentencias necesarias para evaluar la siguiente ecuación para valores de $a = 5$, $b = -1.7$, $c = 2$ y $x = 10.5$.

$$ax^3 + bx^2 - cx + 3$$

4. Escriba el valor ASCII de la ‘*q*’ y de la ‘*Q*’ sin consultar la tabla.
5. Decida qué tipos de datos necesita para escribir un programa que calcule la suma y la media de cuatro números de tipo **int**.
6. Escriba el código necesario para evaluar la expresión:

$$\frac{b^2 - 4ac}{2a}$$

para valores de $a = 1$, $b = 5$ y $c = 2$.

CAPÍTULO 4

ESTRUCTURA DE UN PROGRAMA

En este capítulo estudiaremos cómo es la estructura de un programa Java. Partiendo de un programa ejemplo sencillo analizaremos cada una de las partes que componen su estructura, así tendrá un modelo para realizar sus propios programas. También veremos cómo se construye un programa a partir de varios módulos de clase. Por último, estudiaremos los conceptos de ámbito y accesibilidad de las variables.

ESTRUCTURA DE UNA APLICACIÓN JAVA

Puesto que Java es un lenguaje orientado a objetos, un programa Java se compone solamente de objetos. Recuerde que un objeto es la concreción de una clase, y que una clase equivale a la generalización de un tipo específico de objetos. La clase define los atributos del objeto así como los métodos para manipularlos. Muchas de las clases que utilizaremos pertenecen a la biblioteca de Java, por lo tanto ya están escritas y compiladas. Pero otras tendremos que escribirlas nosotros mismos, dependiendo del problema que tratemos de resolver en cada caso.

Toda aplicación Java está formada por al menos una clase que define un método nombrado **main**, como se muestra a continuación:

```
public class CMiAplicacion
{
    public static void main(String[] args)
    {
        // escriba aquí el código que quiere ejecutar
    }
}
```

Una clase que contiene un método **main** es una plantilla para crear lo que vamos a denominar objeto aplicación, objeto que tiene como misión iniciar y fi-

nalizar la ejecución de la aplicación. Precisamente, el método **main** es el punto de entrada y de salida de la aplicación.

Según lo expuesto, la solución de cualquier problema no debe considerarse inmediatamente en términos de sentencias correspondientes a un lenguaje, sino de objetos naturales del problema mismo, abstraídos de alguna manera, que darán lugar a los objetos que intervendrán en la solución del programa. El empleo de este modelo de desarrollo de programas, nos conduce al diseño y programación orientada a objetos, modelo que ha sido empleado para desarrollar todos los ejemplos de este libro.

Para explicar cómo es la estructura de un programa Java, vamos a plantear un ejemplo sencillo de un programa que presente una tabla de equivalencia entre grados centígrados y grados *fahrenheit*, como indica la figura siguiente:

-30 C	-22.00 F
-24 C	-11.20 F
.	.
.	.
90 C	194.00 F
96 C	204.80 F

La relación entre los grados centígrados y los grados *fahrenheit* viene dada por la expresión $grados\ fahrenheit = 9/5 * grados\ centígrados + 32$. Los cálculos los vamos a realizar para un intervalo de -30 a 100 grados centígrados con incrementos de 6.

Analicemos el problema. ¿De qué trata el programa? De grados. Entonces podemos pensar en objetos “grados” que encapsulen un valor en grados centígrados y los métodos necesarios para asignar al objeto un valor en grados centígrados, así como para obtener tanto el dato grados centígrados como su equivalente en grados *fahrenheit*. En base a esto, podríamos escribir una clase *CGrados* como se puede observar a continuación:

```
class CGrados
{
    private float gradosC; // grados centígrados

    public void CentigradosAsignar(float gC)
    {
        // Establecer el atributo grados centígrados
        gradosC = gC;
    }
}
```

```

public float FahrenheitObtener()
{
    // Retornar los grados fahrenheit equivalentes a gradosC
    return 9F/5F * gradosC + 32;
}

public float CentigradosObtener()
{
    return gradosC; // retornar los grados centígrados
}
}

```

El código anterior muestra que un objeto de la clase *CGrados* tendrá una estructura interna formada por el atributo:

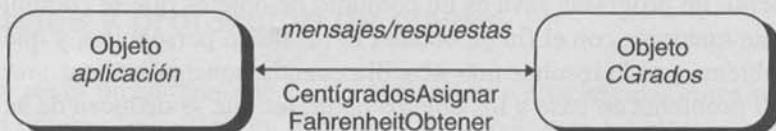
- *gradosC*, grados centígrados,

y una interfaz de acceso formada por los métodos:

- *CentigradosAsignar* que permite asignar a un objeto un valor en grados centígrados.
- *FahrenheitObtener* que permite retornar el valor grados *fahrenheit* equivalente a *gradosC* grados centígrados.
- *CentigradosObtener* que permite retornar el valor almacenado en el atributo *gradosC*.

Sin casi darnos cuenta estamos abstrayendo (separando por medio de una operación intelectual) los elementos naturales que intervienen en el problema a resolver y construyendo objetos que los representan.

Recordando lo visto anteriormente, una aplicación Java tiene que tener un objeto aplicación, que aporte un método **main**, por donde empezará y terminará la ejecución de la aplicación, además de otros que consideramos necesarios. ¿Cómo podemos imaginar esto de una forma gráfica? La figura siguiente da respuesta a esta pregunta:



Entonces, ¿qué tiene que hacer el objeto aplicación? Pues, visualizar cuántos grados *fahrenheit* son -30°C , -24°C , ..., n grados centígrados, ..., 96°C . Y, ¿cómo hace esto? Enviando al objeto *CGrados* los mensajes *CentígradosAsignar* y *FahrenheitObtener* una vez para cada valor desde -30 a 100 grados centígrados con

incrementos de 6. El objeto *CGrados* responderá ejecutando los métodos vinculados con los mensajes que recibe. Según esto, el código de la clase que dará lugar al objeto aplicación puede ser el siguiente:

```
import java.lang.System; // importar la clase System

public class CApGrados
{
    // Definición de constantes
    final static int limInferior = -30;
    final static int limSuperior = 100;
    final static int incremento = 6;

    public static void main(String[] args)
    {
        // Declaración de variables
        CGrados grados = new CGrados();
        int gradosCent = limInferior;
        float gradosFahr = 0;

        while (gradosCent <= limSuperior) // mientras ... hacer:
        {
            // Asignar al objeto grados el valor en grados centígrados
            grados.CentigradosAsignar(gradosCent);
            // Obtener del objeto grados los grados fahrenheit
            gradosFahr = grados.FahrenheitObtener();
            // Escribir la siguiente línea de la tabla
            System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
            // Siguiente valor
            gradosCent += incremento;
        }
    }
}
```

Seguro que pensará que todo el proceso se podría haber hecho utilizando solamente el objeto aplicación, escribiendo todo el código en el método **main**, lo cual es cierto. Pero, lo que se pretende es que pueda ver de una forma clara que, en general, un programa Java es un conjunto de objetos que se comunican entre sí mediante mensajes con el fin de obtener el resultado perseguido, y que la solución del problema puede resultar más sencilla cuando consiga realizar una representación del problema en base a los objetos naturales que se deducen de su enunciado. Piense que en la realidad se enfrentará a problemas mucho más complejos y, por lo tanto, la descomposición en objetos será vital para resolverlos.

Una vez analizado el problema, cree una nueva aplicación desde su entorno de desarrollo y escriba la clase *CGrados*; observe que no es pública. A continuación, escriba la clase aplicación *CApGrados* en el mismo fichero fuente; observe

que es pública. Después, guarde la aplicación que ha escrito en un fichero utilizando como nombre el de la clase aplicación; esto es, *CApGrados.java*. Finalmente, compile y ejecute la aplicación.

No se preocupe si no entiende todo el código. Ahora lo que importa es que aprenda cómo es la estructura de un programa, no por qué se escriben unas u otras sentencias, cuestión que aprenderá más tarde en éste y en sucesivos capítulos. Este ejemplo le servirá como plantilla para inicialmente escribir sus propios programas. Posiblemente su primera aplicación utilice solamente un objeto aplicación, pero con este ejemplo tendrá un concepto más real de lo que es una aplicación Java.

En el ejemplo realizado podemos observar que una aplicación Java consta de:

- Sentencias **import** (para establecer vínculos con otras clases de la biblioteca Java o realizadas por nosotros).
- Una clase aplicación pública (la que incluye el método **main**).
- Otras clases no públicas.

Sabemos también que una clase encapsula los atributos de los objetos que describe y los métodos para manipularlos. Pues bien, cada método consta de:

- Definiciones y/o declaraciones.
- Sentencias a ejecutar.

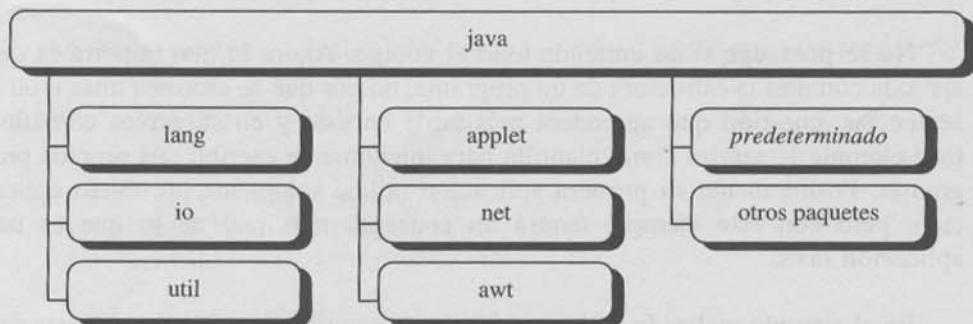
En un fichero se pueden incluir tantas definiciones de clase como se desee, pero sólo una de ellas puede ser declarada como pública (**public**). Recuerde que cada clase pública debe ser guardada en un fichero con su mismo nombre y extensión *.java*.

Los apartados que se exponen a continuación explican brevemente cada uno de estos componentes que aparecen en la estructura de un programa Java.

Paquetes y protección de clases

Un paquete es un conjunto de clases, lógicamente relacionadas entre sí, agrupadas bajo un nombre (por ejemplo, el paquete **java.io** agrupa las clases que permiten a un programa realizar la entrada y salida de información); incluso, un paquete puede contener a otros paquetes. Análogamente a como las carpetas o directorios ayudan a organizar los ficheros en un disco duro, los paquetes ayudan a organizar las clases en grupos para facilitar el acceso a las mismas cuando las necesitemos en un programa. Aprenderá a crear paquetes más adelante, ahora es suficiente con que aprenda a utilizar los paquetes de la biblioteca de Java.

La propia biblioteca de clases de Java está organizada en paquetes dispuestos jerárquicamente. En la figura siguiente se muestran algunos de ellos:



El nivel superior se denomina **java**. En el siguiente nivel tenemos paquetes como **lang**, **applet** o **io**.

Para referirnos a una *clase* de un paquete, tenemos que hacerlo utilizando su nombre completo, excepto cuando el paquete haya sido importado implícita o explícitamente, como veremos a continuación. Por ejemplo, **java.lang.System** hace referencia a la *clase System* del paquete **java.lang** ("java.lang" es el nombre completo del paquete **lang**).

Las clases que guardamos en un fichero cuando escribimos un programa, pertenecen al paquete **predeterminado** sin nombre. Por ejemplo, las clases *CGrados* y *CApGrados* de la aplicación anterior pertenecen, por omisión, a este paquete. De esta forma Java asegura que toda clase pertenece a un paquete.

Protección de una clase

La protección de una clase determina la relación que tiene con otras clases de otros paquetes. Distinguimos dos niveles de protección: *de paquete* y *público*. Una clase con nivel de protección de *paquete* sólo puede ser utilizada por las clases de su paquete (no está disponible para otros paquetes, ni siquiera para los subpaquetes). En cambio, una clase pública puede ser utilizada por cualquier otra clase de otro paquete. ¿Qué se entiende por utilizar? Que una clase puede crear objetos de otra clase y manipularlos utilizando sus métodos.

Por omisión una clase tiene el nivel de protección de *paquete*; por ejemplo, la clase *CGrados* del ejemplo anterior tiene este nivel de protección. En cambio, cuando se desea que una clase tenga protección *pública*, hay que calificarla como tal utilizando la palabra reservada **public**; la clase *CApGrados* del ejemplo anterior tiene este nivel de protección. Otro ejemplo: echando un vistazo a la docu-

mentación de Java, se puede observar que la clase **System** del paquete **java.lang** es pública, razón por la cual se ha podido utilizar en la aplicación *CApGrados*.

Sentencia import

Una clase de un determinado paquete puede hacer uso de otra clase de otro paquete de dos formas:

1. Utilizando su nombre completo en todas las partes del código donde haya que referirse a ella. Por ejemplo:

```
java.lang.System.out.println(gradosFahr);
```

2. Importando la clase, como se indica en el párrafo siguiente, lo que posibilita referirse a ella simplemente por su nombre. Por ejemplo:

```
System.out.println(gradosFahr);
```

Para importar una clase de un paquete desde un programa utilizaremos la sentencia **import**. En un programa Java puede aparecer cualquier número de sentencias **import**, las cuales deben escribirse antes de cualquier definición de clase. Por ejemplo:

```
import java.lang.System; // importar la clase System

public class CApGrados
{
    ...
    System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
    ...
}
```

Como se puede comprobar en el ejemplo anterior, importar una clase permite al programa referirse a ella más tarde sin utilizar el nombre del paquete. Esto es, la sentencia **import** sólo indica al compilador e intérprete de Java dónde encontrar las clases, no trae nada dentro del programa Java actual.

En el caso concreto del ejemplo expuesto, si eliminamos la sentencia **import**, todo seguirá funcionando igual. Esto es así porque las clases del paquete **java.lang** son importadas de manera automática para todos los programas, no sucediendo lo mismo con el resto de los paquetes, que tienen que ser importados explícitamente.

```
public class CApGrados
{
    // ...
    System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
    // ...
}
```

También puede importar un paquete completo de clases utilizando como comodín un asterisco en lugar del nombre específico de una clase. Por ejemplo:

```
import java.lang.*; // importar las clases públicas de este paquete
                    // a las que se refiera el código

public class CApGrados
{
    // ...
    System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
    // ...
}
```

En realidad, para ser exactos, la sentencia **import** del ejemplo anterior importa todas las clases *públicas* del paquete **java.lang** que realmente se usen en el código del programa.

Definiciones y declaraciones

Una declaración introduce uno o más identificadores en un programa. Una declaración es una definición, a menos que no haya asignación de memoria.

Toda variable debe ser definida antes de ser utilizada. La definición de una variable, declara la variable y además le asigna memoria:

```
int gradosCent;
float gradosFahr;

gradosCent = limInferior;
gradosFahr = 0;
```

Además, una variable puede ser iniciada en la propia definición:

```
int gradosCent = limInferior;
float gradosFahr = 0;
```

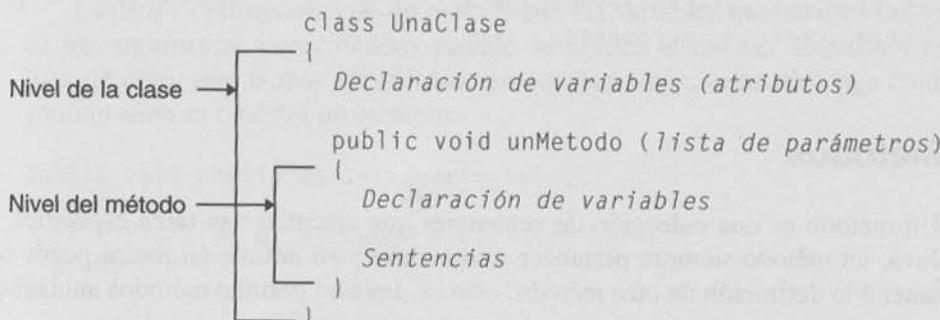
La definición de un método, declara el método y además incluye el cuerpo del mismo. En cambio, la declaración de un método se corresponde con la cabecera de dicho método (su aplicación podrá verla en clases abstractas e interfaces).

```

public float FahrenheitObtener()
{
    // Retornar los grados fahrenheit equivalentes a gradosC
    return 9F/5F * gradosC + 32;
}

```

La declaración o la definición de una variable pueden realizarse a *nivel de la clase* (atributos de la clase) o a *nivel del método* (dentro de la definición de un método). Pero, la definición de un método, siempre ocurre a nivel de la clase.



En un método, las definiciones o declaraciones se pueden realizar en cualquier lugar; o mejor dicho, en el lugar justo donde se necesiten y no necesariamente al principio del método, antes de todas las sentencias.

Sentencia simple

Una *sentencia simple* es la unidad ejecutable más pequeña de un programa Java. Las sentencias controlan el flujo u orden de ejecución. Una sentencia Java puede formarse a partir de: una palabra clave (**for**, **while**, **if ... else**, etc.), expresiones, declaraciones o llamadas a métodos. Cuando se escriba una sentencia hay que tener en cuenta las siguientes consideraciones:

- Toda sentencia simple termina con un punto y coma (;).
- Dos o más sentencias pueden aparecer sobre una misma línea, separadas una de otra por un punto y coma, aunque esta forma de proceder no es aconsejable porque va en contra de la claridad que se necesita cuando se lee el código de un programa.
- Una sentencia nula consta solamente de un punto y coma. Cuando veamos la sentencia **while**, podrá ver su utilización.

Sentencia compuesta o bloque

Una *sentencia compuesta* o *bloque*, es una colección de sentencias simples incluidas entre llaves - { } -. Un bloque puede contener a otros bloques. Un ejemplo de una sentencia de este tipo es el siguiente:

```
{
    grados.CentigradosAsignar(gradosCent);
    gradosFahr = grados.FahrenheitObtener();
    System.out.println(gradosCent + " C" + "\t" + gradosFahr + " F");
    gradosCent += incremento;
}
```

Métodos

Un método es una colección de sentencias que ejecutan una tarea específica. En Java, un método siempre pertenece a una clase y su definición nunca puede tener a la definición de otro método; esto es, Java no permite métodos anidados.

Definición de un método

La definición de un método consta de una *cabecera* y del *cuerpo* del método encerrado entre llaves. La sintaxis para escribir un método es la siguiente:

```
[modificador] tipo-resultado nombre-método ([lista de parámetros])
{
    declaraciones de variables locales;
    sentencias;
    [return [()expresión[]]];
}
```

Las variables declaradas en el cuerpo del método son locales a dicho método y por definición solamente son accesibles dentro del mismo.

Un *modificador* es una palabra clave que modifica el nivel de protección predeterminado del método. Véase el apartado “Protección de los miembros de una clase” expuesto un poco más adelante.

El *tipo del resultado* especifica qué tipo de valor retorna el método. Éste, puede ser cualquier tipo primitivo o referenciado. Para indicar que no se devuelve nada, se utiliza la palabra reservada **void**. El resultado de un método es devuelto a la sentencia que lo invocó, por medio de la siguiente sentencia:

```
return [()expresión[]];
```

La sentencia **return** puede ser o no la última y puede aparecer más de una vez en el cuerpo del método. En el caso de que el método no retorne un valor (**void**), se puede omitir o especificar simplemente **return**. Por ejemplo:

```
void mEscribir()
{
    // ...
    return;
}
```

La *lista de parámetros* de un método son las variables que reciben los valores de los argumentos especificados cuando se invoca al mismo. Consisten en una lista de cero, uno o más identificadores con sus tipos, separados por comas. A continuación se muestra un ejemplo:

```
public void CentigradosAsignar(float gC)
{
    // Establecer el atributo grados centígrados
    gradosC = gC;
}
```

Método main

Toda aplicación Java tiene un método denominado **main**, y sólo uno. Este método es el punto de entrada a la aplicación y también el punto de salida. Su definición es como se muestra a continuación:

```
public static void main(String[] args)
{
    // Cuerpo del método
}
```

Como se puede observar, el método **main** es público (**public**), estático (**static**), no devuelve nada (**void**) y tiene un argumento de tipo **String** que almacenará los argumentos pasados en la línea de órdenes cuando se invoca a la aplicación para su ejecución, concepto que estudiaremos posteriormente en otro capítulo. Para más detalles, puede ver un poco más adelante los apartados “Protección de los miembros de una clase” y “Miembro de un objeto o de una clase”.

Crear objetos de una clase

Sabemos que las clases son plantillas para crear objetos. Pero, ¿cómo se crea un objeto? Para crear un objeto de una clase hay que utilizar el operador **new**, análogamente a como muestra el ejemplo siguiente:

```
CGrados grados = new CGrados();
```

En este ejemplo se observa que para crear un objeto de la clase *CGrados* hay que especificar a continuación del operador **new** el nombre de la clase del objeto seguido de paréntesis. ¿Por qué paréntesis? ¿Es acaso *CGrados* un método? Así es. Más adelante aprenderá que toda clase tiene al menos un método predeterminado especial denominado igual que ella, que es necesario invocar para crear un objeto; ese método se denomina *constructor* de la clase.

Otro ejemplo; ahora con una clase de la biblioteca Java. El paquete **java.util** proporciona una clase denominada **GregorianCalendar**. Un objeto de esta clase representa una fecha, incluyendo también opcionalmente la hora. El siguiente código crea tres objetos de esta clase, *fh1*, *fh2* y *fh3*, iniciados, el primero con la fecha y hora actual por omisión, el segundo con la fecha especificada, y el tercero con la fecha y hora especificadas:

```
GregorianCalendar fh1 = new GregorianCalendar();
GregorianCalendar fh2 = new GregorianCalendar(2001, 1, 21);
GregorianCalendar fh3 = new GregorianCalendar(2001, 1, 21, 12, 30, 15);
```

Las sentencias anteriores son válidas porque, como puede comprobar si lo desea, la clase **GregorianCalendar** proporciona varias formas de construir un objeto: sin utilizar parámetros, con tres parámetros (año, mes y día), con seis parámetros (año, mes, día, hora, minutos y segundos), etc.

Cuando se crea un nuevo objeto utilizando **new**, Java asigna automáticamente la cantidad de memoria necesaria para ubicar ese objeto. Si no hubiera suficiente espacio de memoria disponible, el operador **new** lanzará una excepción **OutOfMemoryError** cuyo estudio posponemos. Después de saber esto quizás se pregunte: ¿Quién libera esa memoria y cuándo lo hace? La respuesta es otra vez la misma: Java se encarga de hacerlo en cuanto el objeto no se utilice, cosa que ocurre cuando ya no exista ninguna referencia al objeto. Por ejemplo, en el código que se muestra a continuación, la memoria asignada a los objetos *fh1*, *fh2* y *fh3* será liberada cuando finalice la ejecución del método **main**.

```
import java.util.*;
public class CFechaHora
{
    public static void main(String[] args)
    {
        GregorianCalendar fh1 = new GregorianCalendar();
        GregorianCalendar fh2 = new GregorianCalendar(2001, 1, 21);
        GregorianCalendar fh3 = new GregorianCalendar(2001, 1, 21, 12, 30, 15);
        // ...
    }
}
```

Ahora basta con que sepa que Java cuenta con una herramienta denominada *recolector de basura* que busca objetos que no se utilizan con el fin de destruirlos liberando la memoria que ocupan. Más adelante aprenderá sobre este mecanismo.

Cómo acceder a los miembros de un objeto

Para acceder desde un método de la clase aplicación o de cualquier otra clase a un miembro (atributo o método) de un objeto de otra clase diferente se utiliza la sintaxis siguiente: *objeto.miembro*. Por ejemplo:

```
miObjeto.atributo;
miObjeto.metodo([argumentos]);
```

Lógicamente, como pueden existir varios objetos de la misma clase, es necesario especificar de quién es el miembro. Si el miembro es a su vez un objeto, la sintaxis se extiende siguiendo la misma sintaxis: *objeto.mbroObjeto.miembro*. Recuerde que el operador punto (.) se evalúa de izquierda a derecha.

Cuando el miembro accedido es un método, la interpretación que se hace en programación orientada a objetos es que el objeto ha recibido un mensaje, el especificado por el nombre del método, y responde ejecutando ese método. Los mensajes que puede recibir un objeto se corresponden con los nombres de los métodos de su clase. Por ejemplo, una sentencia como:

```
grados.CentigradosAsignar(gradosCent);
```

se interpreta como que el objeto *grados* recibe el mensaje *CentigradosAsignar*. Entonces el objeto responde a ese mensaje ejecutando el método de su clase que tenga el mismo nombre. Lógicamente, como el método se ejecuta para un objeto concreto, el cuerpo del mismo no necesita especificar explícitamente de qué objeto es el miembro accedido. Esto es, en el ejemplo siguiente se sabe que *gradosC* pertenece al objeto que está respondiendo al mensaje *CentigradosAsignar*.

```
public void CentigradosAsignar(float gC)
{
    // Establecer el atributo grados centígrados
    gradosC = gC;
}
```

Es importante asimilar que un programa orientado a objetos sólo se compone de objetos que se comunican mediante mensajes. Desde este conocimiento, no tiene sentido pensar que un método se pueda invocar aisladamente, esto es, sin que exista un objeto para el que es invocado. Por ejemplo, si en el método **main** de nuestra aplicación ejemplo pudiéramos escribir:

```
CentígradosAsignar(gradosCent);
```

seguro que nos preguntaríamos ¿a quién se asigna el valor *gradosCent*? Los métodos **static** que estudiaremos más tarde son una excepción a la regla.

Protección de los miembros de una clase

Los miembros de una clase son los atributos y los métodos, y su nivel de protección determina quién puede acceder a los mismos. Los niveles de protección a los que nos referimos son: de *paquete*, *público*, *privado* y *protegido*. De este último hablaremos en un capítulo posterior.

Por ejemplo, en la clase *CGrados* de la aplicación realizada al principio de este capítulo, hemos definido los atributos privados y los métodos, públicos:

```
class CGrados
{
    private float gradosC; // grados centígrados

    public void CentígradosAsignar(float gC)
    {
        // Establecer el atributo grados centígrados
        gradosC = gC;
    }
    // ...
}
```

Un miembro de una clase declarado *privado* puede ser accedido únicamente por los métodos de su clase. En el ejemplo anterior se puede observar que el atributo *gradosC* es privado y es accedido por el método *CentígradosAsignar*.

Si un método de otra clase, por ejemplo el método **main** de la clase *CApGrados*, incluyera una sentencia como la siguiente,

```
grados.gradosC = 30;
```

el compilador Java mostraría un error indicando que el miembro *gradosC* no es accesible desde esta clase, por tratarse de un miembro privado de *CGrados*.

Un miembro de una clase declarado *público* es accesible desde cualquier método definido dentro o fuera de la clase o paquete actual. Por ejemplo, en la clase *CApGrados*, se puede observar cómo el objeto *grados* de la clase *CGrados* creado en el método **main** accede a su método *CentígradosAsignar* con el fin de modificar el valor de su miembro privado *gradosC*.

```

public class CApGrados
{
    // ...
    public static void main(String[] args)
    {
        // Declaración de variables
        CGrados grados = new CGrados();
        // ...
        while (gradosCent <= limSuperior) // while ... hacer:
        {
            // Asignar al objeto grados el valor en grados centígrados
            grados.CentigradosAsignar(gradosCent);
            // ...
        }
    }
}

```

Generalmente los atributos de una clase de objetos se declaran privados, escondiendo así ocultos para otras clases, siendo posible el acceso a los mismos únicamente a través de los métodos públicos de dicha clase. El mecanismo de ocultación de miembros se conoce en la programación orientada a objetos como *encapsulación*: proceso de ocultar la estructura interna de datos de un objeto y permitir el acceso sólo a través de la interfaz pública definida, entendiendo por interfaz pública el conjunto de miembros públicos de una clase. ¿Qué beneficios reporta la encapsulación? Que un usuario de una determinada clase no pueda escribir código en base a la estructura interna del objeto, sino sólo en base a la interfaz pública; esta forma de proceder obliga a pensar en objetos y a trabajar con ellos. En el capítulo “Clases” que expondremos más adelante abundaremos más sobre lo dicho y sobre otras muchas cuestiones.

El nivel de protección predeterminado para un miembro de una clase es el de *paquete*. Un miembro de una clase con este nivel de protección puede ser accedido desde todas las otras clases del mismo paquete.

Miembro de un objeto o de una clase

Sabemos que una clase agrupa los atributos y los métodos que definen a los objetos de esa clase. Pero, cada objeto que creemos de esa clase ¿mantiene una copia tanto de los atributos como de los métodos? Lógicamente, cada objeto mantiene su propia copia de los atributos para almacenar sus datos particulares; pero, de los métodos sólo hay una copia para todos los objetos, lo cual también es lógico, porque cada objeto sólo requiere utilizarlos; por ejemplo, cuando necesite modificar sus atributos. Desde este análisis se dice que los miembros son del objeto; esto es, un mismo atributo tiene un valor específico para cada objeto, y un objeto ejecuta un método en respuesta a un mensaje recibido.

Esta forma de concebir los objetos puede suponer, en ocasiones, un desperdicio de espacio de almacenamiento; por ejemplo, volviendo a la clase *COrdenador* que expusimos en el capítulo 2, podríamos pensar en añadir un nuevo atributo que fuera el tiempo de garantía. Si suponemos que este tiempo es el mismo para todos los ordenadores, sería más eficiente definir un atributo que no formara parte de la estructura de cada objeto, sino que fuera compartido por todos los objetos. En este caso, diremos que el atributo *es de la clase* de los objetos, no del objeto.

Un atributo de la clase almacena información común a todos los objetos de esa clase. Se define agregándole previamente la palabra reservada **static**, y existe aunque no haya objetos definidos de la clase.

Para acceder a un atributo **static** de la clase puede utilizar un objeto de la clase, o bien el nombre de la clase como puede verse en el ejemplo siguiente:

```
class COrdenador
{
    String Marca;
    String Procesador;
    String Pantalla;
    static byte Garantia;
    boolean OrdenadorEncendido;
    boolean Presentación;
    // ...

    public static void main (String[] args)
    {
        // Garantía existe aunque no haya objetos definidos de la clase
        COrdenador.Garantia = 1;
    }
}
```

Utilizar una expresión como *MiOrdenador.Garantía*, siendo *MiOrdenador* un objeto de la clase *COrdenador*, aunque sea correcta, no se aconseja porque puede resultar engañosa. Parece que nos estamos refiriendo al atributo *Garantía* del objeto *MiOrdenador*, cuando en realidad nos estamos refiriendo a todos los objetos que el programa haya creado de la clase *COrdenador*.

Análogamente, un método declarado **static** es un método de la clase. Este tipo de métodos no se ejecutan para un objeto particular, sino que se aplican en general donde se necesiten, lo que impide que puedan acceder a un miembro del objeto. Una aplicación puede acceder a un método estático de la misma forma que se ha expuesto para un atributo estático.

En el ejemplo que se muestra a continuación se puede observar que para establecer el valor del atributo privado *Garantía* se ha utilizado un método estático. Si

se hubiera utilizado un método no **static**, tendría que ser invocado a través de un objeto de la clase, lo que, siendo correcto, resultaría engañoso.

```
public class CMiAplicacion
{
    public static void main (String[] args)
    {
        COrdenador.EstablecerGarantia((byte)3);
    }
}

class COrdenador
{
    private String Marca;
    private String Procesador;
    private String Pantalla;
    private static byte Garantia;
    private boolean OrdenadorEncendido;
    private boolean Presentacion;
    // ...

    public static void EstablecerGarantia(byte g)
    {
        Garantia = g; // Garantia es un miembro de la clase
    }
}
```

El método *EstablecerGarantía* del ejemplo anterior puede acceder a *Garantía* porque es un miembro estático pero no podría incluir, por ejemplo, una sentencia como *Marca = "Ast"* porque *Marca* no es **static**.

Ahora puede comprender por qué el método **main** es **static**: para que pueda ser invocado aunque no exista un objeto de su clase. Por ejemplo, el método **main** de la clase *CMiAplicacion* anterior, es invocado cuando se ejecuta la aplicación, independientemente de que exista un objeto de esa clase.

Referencias a objetos

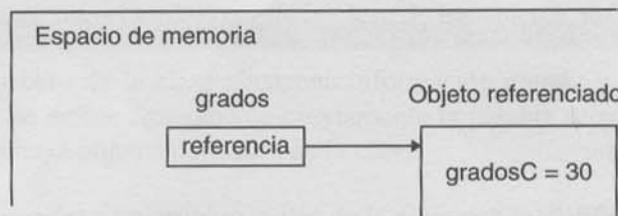
Según lo que hemos aprendido hasta ahora, para crear un objeto de una clase hay que hacerlo explícitamente utilizando el operador **new**. Por ejemplo:

```
CGrados grados = new CGrados();
```

El operador **new** devuelve una referencia al nuevo objeto, que se almacena en una variable del tipo del objeto. En el ejemplo anterior, la referencia devuelta por

el operador **new** es almacenada en la variable *grados* del tipo *CGrados*. La clase *CGrados* se encuadra dentro de lo que hemos denominado *tipos referenciados*.

Gráficamente puede imaginarse una referencia y el objeto referenciado, ubicados en algún lugar del espacio de memoria correspondiente a su aplicación, así:



En realidad una referencia es la posición de memoria donde se localiza un objeto. Observará que anteriormente nos hemos referido a la referencia *grados* como el objeto *grados*. Esto es una forma de abreviar que no crea confusión, ya que *grados* es única y referencia un único objeto *CGrados*. Una expresión como “el objeto referenciado por la variable *grados*” resulta demasiado larga y no aporta más información. Expresándonos en estos términos, cuando se asigne un objeto a otro, o bien se pasen objetos como argumentos a métodos, lo que se están copiando son referencias, no el contenido de los objetos.

El siguiente ejemplo aclarará este concepto. Se trata de la aplicación *CRacional* que expusimos al final del capítulo 2, compuesta por la clase *CRacional* a la que hemos añadido un nuevo método estático que permite sumar dos números racionales, devolviendo como resultado el número racional resultante de la suma.

```

class CRacional
{
    private int Numerador;
    private int Denominador;

    public void AsignarDatos(int num, int den)
    {
        Numerador = num;
        if (den == 0) den = 1; // el denominador no puede ser cero
        Denominador = den;
    }

    public void VisualizarRacional()
    {
        System.out.println(Numerador + "/" + Denominador);
    }
}
  
```

```

public static CRacional Sumar(CRacional a, CRacional b)
{
    CRacional r = new CRacional();
    int num = a.Numerador * b.Denominador +
              a.Denominador * b.Numerador;
    int den = a.Denominador * b.Denominador;
    r.AsignarDatos(num, den);
    return r;
}

public static void main (String[] args)
{
    // Punto de entrada a la aplicación
    CRacional r1, r2;
    r1 = new CRacional(); // crear un objeto CRacional
    r1.AsignarDatos(2, 5);
    r2 = r1;

    r1.AsignarDatos(3, 7);
    r1.VisualizarRacional(); // se visualiza 3/7
    r2.VisualizarRacional(); // se visualiza 2/5

    CRacional r3;
    r2 = new CRacional(); // crear un objeto CRacional
    r2.AsignarDatos(2, 5);
    r3 = CRacional.Sumar(r1, r2); // r3 = 3/7 + 2/5
    r3.VisualizarRacional(); // se visualiza 29/35
}
}

```

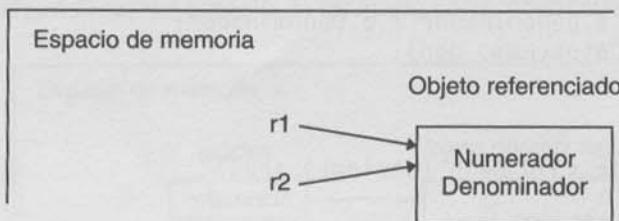
La clase *CRacional* encapsula una estructura de datos formada por dos enteros: *numerador* y *denominador*; y para acceder a esta estructura proporciona la interfaz pública formada por los métodos:

- *AsignarDatos* que permite establecer el numerador y el denominador de un número racional.
- *VisualizarRacional* que permite visualizar un racional en forma de quebrado.
- *Sumar* que devuelve el número racional resultante de sumar otros dos pasados como argumentos.

Analizada la clase *CRacional* pasemos a estudiar el método **main**. La primera parte de este método declara dos variables *r1* y *r2* de tipo *CRacional*, crea un nuevo objeto *r1* de tipo *CRacional* asignándole el valor $2/5$, y asigna el valor de *r1* a *r2*.

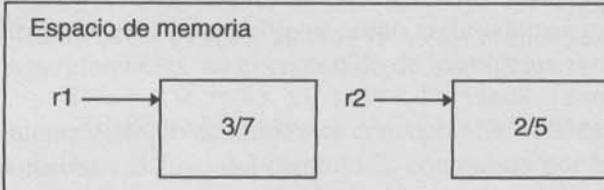
A continuación, asigna a *r1* un nuevo valor $3/7$, ¿cuál es el valor de *r2*? Comprobamos que es el mismo que el de *r1*. ¿Qué ha ocurrido? Que cuando se asignó

r1 a *r2*, simplemente se creó una nueva referencia al mismo objeto referenciado por *r1*. Por lo tanto, modificar el objeto al que se refiere *r1* es modificar el objeto al que se refiere *r2* porque *r1* y *r2* referencian el mismo objeto.



Si realmente lo que deseamos es que *r1* y *r2* señalen a objetos separados, hay que utilizar **new** con ambas referencias para crear objetos separados:

```
r1 = new CRacional(); // crear un objeto CRacional r1
r1.AsignarDatos(3, 7);
r2 = new CRacional(); // crear un objeto CRacional r2
r2.AsignarDatos(2, 5);
```



Como hemos visto, una variable de un tipo referenciado se puede asignar a otra del mismo tipo. En cambio, no existe aritmética de referencias (por ejemplo, a una referencia no se le puede sumar un entero) ni tampoco se puede asignar directamente un entero a una referencia.

Pasando argumentos a los métodos

La segunda parte del método **main** del ejemplo anterior, crea un objeto *r2* y le asigna el valor 2/5. A continuación, invoca al método estático *Sumar* pasándole como argumentos los objetos *r1* y *r2* que queremos sumar. El resultado devuelto por *Sumar* será un objeto *CRacional* que quedará referenciado por *r3*.

```
// ...
CRacional r3;
r2 = new CRacional(); // crear un objeto CRacional
r2.AsignarDatos(2, 5);
```

```
r3 = CRacional.Sumar(r1, r2); // r3 = 3/7 + 2/5
r3.VisualizarRacional(); // se visualiza 29/35
```

Analicemos el método *Sumar*. Este método tiene dos parámetros de tipo *CRacional*. Después de que el método ha sido invocado desde **main**, *a* y *b* señalan a los mismos objetos que *r1* y *r2*. Esto significa que los objetos pasados a los parámetros de un método son siempre referencias a dichos objetos, lo cual significa que cualquier modificación que se haga a esos objetos dentro del método afecta al objeto original. En cambio, las variables de un tipo primitivo pasan por valor, lo cual significa que se pasa una copia, por lo que cualquier modificación que se haga a esas variables dentro del método no afecta a la variable original.

Cuando se invoca a un método, el primer argumento es pasado al primer parámetro, el segundo argumento es pasado al segundo parámetro y así sucesivamente. En Java todos los argumentos que son objetos son pasados por referencia.

```
public static CRacional Sumar(CRacional a, CRacional b)
{
    CRacional r = new CRacional();
    int num = a.Numerador * b.Denominador +
              a.Denominador * b.Numerador;
    int den = a.Denominador * b.Denominador;
    r.AsignarDatos(num, den);
    return r;
}
```

A continuación, el método *Sumar* utiliza **new** para crear un nuevo objeto *r* al que asigna el resultado de la suma de los objetos *a* y *b*. Finalmente devuelve *r*. Otra vez más lo que se devuelve es una referencia que se copia en *r3*. Finalizado este proceso la variable *r* desaparece por ser local, no sucediendo lo mismo con el objeto que señalaba, ya que ahora está señalado por *r3*.

El recolector de basura de Java eliminará un objeto cuando no exista ninguna referencia al mismo.

PROGRAMA JAVA FORMADO POR MÚLTIPLES FICHEROS

Según lo que hemos visto, un programa Java es un conjunto de objetos que se comunican entre sí. Para crear los objetos, escribimos plantillas que denominamos clases. Por ejemplo, en la aplicación acerca de números racionales escribimos una sola clase, pero en la aplicación acerca de conversión de grados centígrados a *Fahrenheit*, escribimos dos clases. En ambas aplicaciones, almacenamos todo su código en un único fichero *.java*. Esto no debe inducirnos a pensar que todo programa tiene que estar escrito en un único fichero. De hecho no es así, ya que ge-

neralmente se almacena cada clase en un único fichero para favorecer su mantenimiento y posterior reutilización.

Como ejemplo, reconstruyamos la aplicación *CRacional* creando ahora dos clases separadas: *CRacional* y *CAplicacion*.

La clase *CRacional* incluirá su estructura de datos y su interfaz pública, excepto el método **main** que será ahora incluido en *CAplicacion*. Cuando haya escrito la clase *CRacional* guárdela en el fichero *CRacional.java*.

```
public class CRacional
{
    private int Numerador;
    private int Denominador;

    public void AsignarDatos(int num, int den)
    {
        Numerador = num;
        if (den == 0) den = 1; // el denominador no puede ser cero
        Denominador = den;
    }

    public void VisualizarRacional()
    {
        System.out.println(Numerador + "/" + Denominador);
    }

    public static CRacional Sumar(CRacional a, CRacional b)
    {
        CRacional r = new CRacional();
        int num = a.Numerador * b.Denominador +
                  a.Denominador * b.Numerador;
        int den = a.Denominador * b.Denominador;
        r.AsignarDatos(num, den);
        return r;
    }
}
```

Escriba ahora la clase *CAplicacion* que se muestra a continuación y guárdela en el fichero *CAplicacion.java*.

```
public class CAplicacion
{
    public static void main (String[] args)
    {
        // Punto de entrada a la aplicación
        CRacional r1, r2;
```

```

r1 = new CRacional();           // crear un objeto CRacional
r1.AsignarDatos(2, 5);
r2 = r1;

r1.AsignarDatos(3, 7);
r1.VisualizarRacional();       // se visualiza 3/7
r2.VisualizarRacional();       // se visualiza 3/7

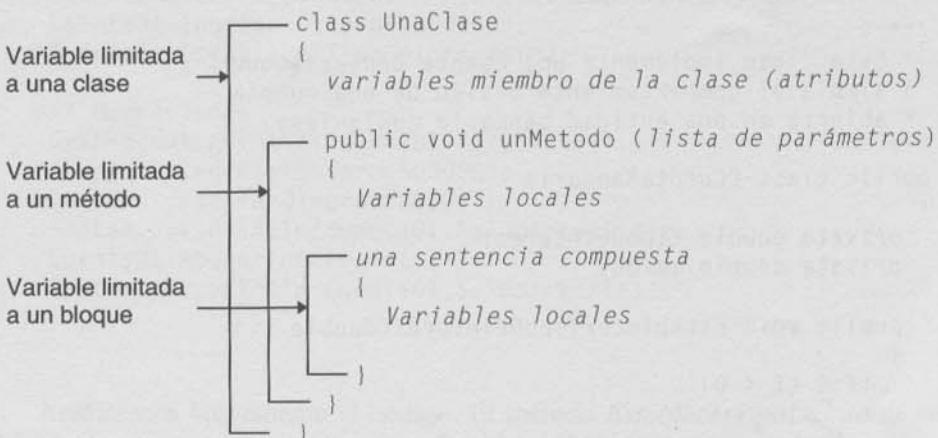
CRacional r3;
r2 = new CRacional();           // crear un objeto CRacional
r2.AsignarDatos(2, 5);
r3 = CRacional.Sumar(r1, r2);  // r3 = 3/7 + 2/5
r3.VisualizarRacional();       // se visualiza 29/35
}

```

Cuando se compile *CAplicacion*, que por omisión pertenece al paquete pre-determinado, puesto que necesita utilizar la clase *CRacional*, buscará también ésta en el mismo paquete, lo que supone buscar su fichero compilado, o en su defecto su fichero fuente, en el directorio actual de trabajo. Por lo tanto, antes de compilar la aplicación asegúrese de que el fichero *CRacional.class* o *CRacional.java* está en el mismo directorio que *CAplicacion.java*.

ACCESIBILIDAD DE VARIABLES

Aunque este tema ya ha sido tratado, realizamos ahora un resumen. Se denomina *ámbito* de una variable a la parte de un programa donde dicha variable puede ser referenciada por su nombre. Una variable puede ser limitada a una clase, a un método, o a un bloque de código correspondiente a una sentencia compuesta.



Una variable miembro de una clase puede ser declarada en cualquier sitio dentro de la clase siempre que sea fuera de todo método. La variable está disponible para todo el código de la clase.

Una variable declarada dentro de un método es una variable local al método. Los parámetros de un método son también variables locales al método. Y una variable declarada dentro de un bloque correspondiente a una sentencia compuesta también es una variable local a ese bloque.

En general, una *variable local* existe y tiene valor desde su punto de declaración hasta el final del bloque donde está definida. Cada vez que se ejecuta el bloque que la contiene, la variable local es nuevamente definida, y cuando finaliza la ejecución del mismo, la variable local deja de existir. Un elemento con carácter local es accesible solamente dentro del bloque al que pertenece.

EJERCICIOS RESUELTOS

Con los conocimientos que hemos adquirido hasta ahora vamos a realizar una aplicación sencilla para simular una cuenta bancaria.

Una cuenta bancaria vista como un objeto tiene, por una parte, atributos que definen su estado, como *Tipo de interés* y *Saldo*, y por otra, operaciones que definen su comportamiento, como *Establecer tipo de interés*, *Ingresar dinero*, *Retirar dinero*, *Saldo actual* o *Abonar intereses*.

Una vez abstraídas las características generales de la clase de objetos cuentas bancarias, el paso siguiente es escribir el código que da lugar a la implementación de dicha clase. Ésta puede ser más o menos así:

```
/**
 * Esta clase implementa una cuenta bancaria que
 * simula el comportamiento básico de una cuenta
 * abierta en una entidad bancaria cualquiera.
 */
public class CCuentaBancaria
{
    private double tipoDeInterés;
    private double saldo;

    public void EstablecerTipoDeInterés(double ti)
    {
        if ( ti < 0 )
        {
            System.out.println("El tipo de interés no puede ser negativo");
            return; // retornar
        }
    }
}
```

```
    tipoDeInterés = ti;
}

public void IngresarDinero(double ingreso)
{
    saldo += ingreso;
}

public void RetirarDinero(double cantidad)
{
    if ( saldo - cantidad < 0 )
    {
        System.out.println("No tiene saldo suficiente");
        return;
    }
    // Hay saldo suficiente. Retirar la cantidad.
    saldo -= cantidad;
}

public double SaldoActual()
{
    return saldo;
}

public void AbonarIntereses()
{
    saldo += saldo * tipoDeInterés / 100;
}

public static void main (String[] args)
{
    // Abrir una cuenta con 1.000.000 a un 2%
    CCuentaBancaria Cuenta01 = new CCuentaBancaria();
    Cuenta01.IngresarDinero(1000000);
    Cuenta01.EstablecerTipoDeInterés(2);

    // Operaciones
    System.out.println(Cuenta01.SaldoActual());
    Cuenta01.IngresarDinero(500000);
    Cuenta01.RetirarDinero(200000);
    System.out.println(Cuenta01.SaldoActual());
    Cuenta01.AbonarIntereses();
    System.out.println(Cuenta01.SaldoActual());
}
}
```

Analicemos brevemente el código. El método *EstablecerTipoDeInterés* verifica si el valor pasado como argumento es negativo, en cuyo caso lo notifica y termina. Si es positivo, lo asigna al miembro *tipoDeInterés*.

El método *IngresarDinero* acumula la cantidad pasada como argumento sobre el saldo actual.

El método *RetirarDinero* verifica si hay suficiente dinero como para poder retirar la cantidad solicitada. En caso negativo lo notifica y termina; en caso positivo, resta del saldo la cantidad retirada.

El método *SaldoActual* devuelve el valor del saldo actual en la cuenta.

El método *AbonarIntereses* acumula los intereses sobre el saldo actual.

Finalmente, el método **main** crea e inicia un objeto de la clase *CCuentaBancaria* y realiza sobre el mismo las operaciones programadas con el fin de comprobar su correcto funcionamiento.

EJERCICIOS PROPUESTOS

1. Escriba la aplicación *CApGrados.java* y compruebe los resultados.
2. En el capítulo 1 hablamos acerca del depurador. Si su entorno integrado favorito aporta la funcionalidad necesaria para depurar un programa, pruebe a ejecutar la aplicación *CApGrados.java* paso a paso y verifique los valores que van tomando las variables a lo largo de la ejecución.
3. Modifique los límites inferior y superior de los grados centígrados, el incremento, y ejecute de nuevo la aplicación.
4. Cargue en su entorno de desarrollo integrado la aplicación *CApGrados.java* y modifique la sentencia:

```
return 9F/5F * gradosC + 32;
```

correspondiente al método *FahrenheitObtener* de la clase *CGrados*, como se muestra a continuación:

```
return 9/5 * gradosC + 32;
```

Después, compile y ejecute la aplicación. Explique lo que sucede.

5. Reconstruya la aplicación *CApGrados.java* para que cada clase esté almacenada en un fichero: la clase *CGrados* en el fichero *CGrados.java* y la clase *CApGrados* en el fichero *CApGrados.java*.

CAPÍTULO 5

© F.J.Ceballos/RA-MA

CLASES DE USO COMÚN

Aunque las clases que hemos aprendido a escribir en los capítulos anteriores son la base de nuestras aplicaciones, la potencia, en la práctica, del lenguaje Java viene dada por su biblioteca de clases. Hay dos paquetes que destacan por las clases de propósito general que incluyen: **java.io** y **java.lang**.

El paquete **java.io** contiene las clases de objetos que proporcionan los métodos necesarios para escribir información en diversos dispositivos, por ejemplo en la salida estándar (que se corresponde normalmente con la pantalla de su ordenador) y para leer información desde otros dispositivos, por ejemplo, desde la entrada estándar (que es normalmente el teclado de su ordenador).

El paquete **java.lang** contiene clases que se aplican al lenguaje mismo. Por ejemplo, clases especiales que encapsulan los tipos primitivos de datos, la clase **System** que proporciona los objetos para manipular la entrada/salida (E/S) estándar, clases para manipular cadenas de caracteres, una clase que proporciona los métodos correspondientes a las funciones matemáticas de uso más frecuente, una clase para analizar otras clases, etc.

En este capítulo aprenderá cómo leer y escribir información desde sus aplicaciones, y a trabajar con las clases utilizadas más frecuentemente.

DATOS NUMÉRICOS Y CADENAS DE CARACTERES

La finalidad de una aplicación es procesar datos que, generalmente, serán obtenidos de algún medio externo por la propia aplicación (por ejemplo, del teclado o de un fichero en disco) y procesados por la misma con el fin de obtener unos resultados. Estos datos se pueden clasificar en: *numéricos* y *cadenas de caracteres*.

Tanto los datos leídos como los resultados obtenidos serán almacenados en variables pertenecientes a la estructura interna de uno o más objetos o declaradas en algún método. Los datos serán leídos a través de los métodos proporcionados por las clases de E/S y serán asignados a las variables directamente por ellos, o bien utilizando una sentencia de asignación de la forma:

variable operador_de_asignación valor

Una sentencia de asignación es asimétrica. Esto quiere decir que se evalúa la expresión de la derecha y el resultado se asigna a la variable especificada a la izquierda. Por ejemplo:

```
d = a + b * c; // el valor de a + b * c se asigna a d
```

Pero no sería válido escribir:

```
a + b * c = d; // el valor de d no se puede asignar a a + b * c
```

Los datos numéricos serán almacenados en variables de alguno de los tipos primitivos expuestos en el capítulo 3. Por ejemplo:

```
double radio, área;
// ...
área = 3.141592 * radio * radio;
```

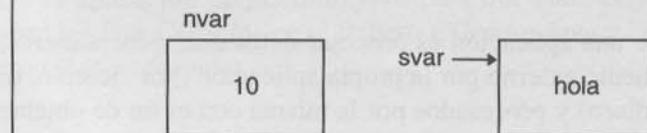
Las cadenas de caracteres serán almacenadas en objetos de la clase **String** o en matrices, cuyo estudio se ha pospuesto para un capítulo posterior. Un objeto de la clase **String** se define y se le asigna un valor, así:

```
String cadena; // cadena permite referenciar un objeto String
cadena = "hola"; // equivale a: cadena = new String("hola");
```

Cuando se asigna un valor a una variable estamos colocando ese valor en una localización de memoria asociada con esa variable.

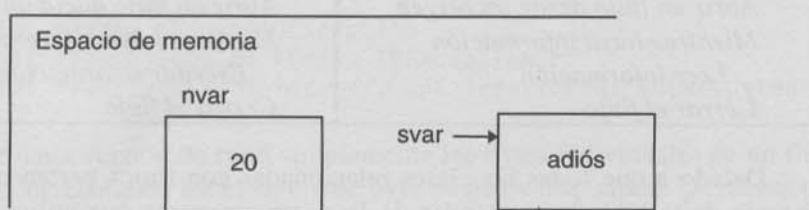
```
int nvar = 10; // variable de un tipo primitivo (int)
String svar = "hola"; // referencia a un objeto de tipo String
```

Espacio de memoria



Lógicamente, cuando la variable tiene asignado un valor y se le asigna uno nuevo, el valor anterior es destruido ya que el valor nuevo pasa a ocupar la misma localización de memoria. En el ejemplo siguiente, se puede observar con respecto a la situación anterior, que el contenido de *nvar* se modifica con un nuevo valor 20, y que la referencia *svar* también se modifica; ahora contiene la referencia a un nuevo objeto **String** “adiós”.

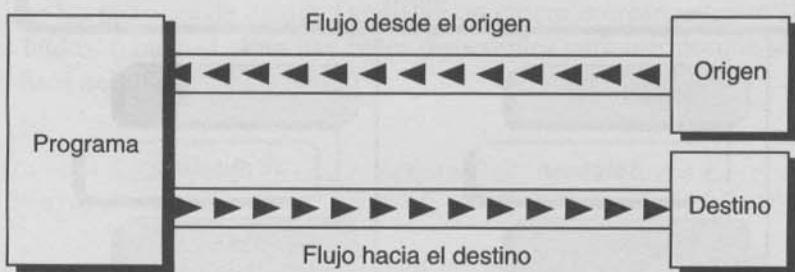
```
nvar = 20;
svar = "adiós";
```



ENTRADA Y SALIDA

Frecuentemente un programa necesitará obtener información desde un origen o enviar información a un destino. Por ejemplo, obtener información desde, o enviar información a: un fichero en el disco, la memoria del ordenador, otro programa, Internet, etc.

La comunicación entre el origen de cierta información y el destino, se realiza mediante un *flujo* de información (en inglés *stream*).



Un *flujo* es un objeto que hace de intermediario entre el programa, y el origen o el destino de la información. Esto es, el programa leerá o escribirá en el *flujo* sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que se traduce en una facilidad más a la hora de escribir programas.

Entonces, para que un programa pueda obtener información desde un origen tiene que abrir un flujo y leer la información. Análogamente, para que un programa puede enviar información a un destino tiene que abrir un flujo y escribir la información.

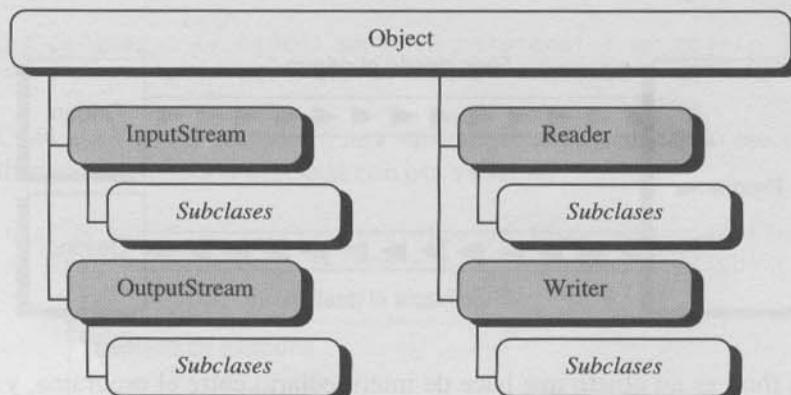
Los algoritmos para leer y escribir datos son siempre más o menos los mismos:

Leer	Escribir
<i>Abrir un flujo desde un origen</i>	<i>Abrir un flujo hacia un destino</i>
<i>Mientras haya información</i>	<i>Mientras haya información</i>
<i> Leer información</i>	<i> Escribir información</i>
<i>Cerrar el flujo</i>	<i>Cerrar el flujo</i>

Debido a que todas las clases relacionadas con flujos pertenecen al paquete **java.io** de la biblioteca estándar de Java, un programa que utilice flujos de E/S tendrá que importar este paquete:

```
import java.io.*;
```

Las clases del paquete **java.io** están divididas en dos grupos distintos, ambos derivados de la clase **Object** del paquete **java.lang**, según se muestra en la figura siguiente. El grupo de la izquierda ha sido diseñado para trabajar con datos de tipo **byte** y el de la derecha con datos de tipo **char**. Ambos grupos presentan clases análogas que tienen interfaces casi idénticas, por lo que se utilizan de la misma manera.



Las clases sombreadas son clases abstractas. Una clase abstracta no permite que se creen objetos de ella. Su misión es proporcionar miembros comunes que serán compartidos por todas sus subclases.

Flujos de entrada

La clase **InputStream** es una clase abstracta que es superclase de todas las clases que representan un flujo en el que un destino lee bytes de un origen. Cuando una aplicación define un flujo de entrada, la aplicación es destino de ese flujo de bytes, y es todo lo que se necesita saber.

El método más importante de esta clase es **read**. Este método se presenta de tres formas:

```
public int read() throws IOException
public int read(byte[] b) throws IOException
public int read(byte[] b, int off, int len) throws IOException
```

La primera versión de **read** simplemente lee bytes individuales de un flujo de entrada; concretamente lee el siguiente byte de datos disponible. Devuelve un entero (**int**) correspondiente al valor ASCII del carácter leído, al número de bytes leídos si se lee una matriz, o bien -1 cuando en un intento de leer datos se alcanza el final del flujo (esto es, no hay más datos).

Por ejemplo, suponiendo que tenemos definido un objeto *flujoE* (flujo de entrada) de alguna subclase de **InputStream**, el siguiente código lee un byte del origen vinculado con *flujoE*:

```
int n;
n = flujoE.read();
```

La segunda versión del método **read** lee un número de bytes de un flujo de entrada y los almacena en una matriz *b* (más adelante, dedicaremos un capítulo a explicar las matrices de datos). Devuelve un entero correspondiente al número de bytes leídos, o bien -1 si no hay bytes disponibles para leer porque se ha alcanzado el final del flujo.

```
int n;
byte[] b = new byte[128]; // matriz 'b' de 128 bytes
n = flujoE.read(b);      // n es el número de bytes leídos
```

La tercera versión del método **read** lee un máximo de *len* bytes a partir de la posición *off* de un flujo de entrada y los almacena en una matriz *b*.

Cada uno de estos métodos ha sido escrito para que bloquee la ejecución del programa que los invoque hasta que toda la entrada solicitada esté disponible.

Análogamente, la clase **Reader** es una clase abstracta que es superclase de todas las clases que representan un flujo para leer caracteres desde un origen. Sus

métodos son análogos a los de la clase **InputStream**, con la diferencia de que utilizan parámetros de tipo **char** en lugar de **byte**.

Flujos de salida

La clase **OutputStream** es una clase abstracta que es superclase de todas las clases que representan un flujo en el que un origen escribe bytes en un destino. Cuando una aplicación define un flujo de salida, la aplicación es origen de ese flujo de bytes (es la que envía los bytes), y es todo lo que se necesita saber.

El método más importante de esta clase es **write**. Este método se presenta de tres formas:

```
public void write(int b) throws IOException
public void write(byte[] b) throws IOException
public void write(byte[] b, int off, int len) throws IOException
```

La primera versión de **write** simplemente escribe el byte especificado en un flujo de salida. Puesto que su parámetro es de tipo **int**, lo que se escribe es el valor correspondiente a los 8 bits menos significativos, el resto son ignorados.

Por ejemplo, suponiendo que tenemos definido un objeto *flujoS* (flujo de salida) de alguna subclase de **OutputStream**, el siguiente código escribe el byte especificado en el destino vinculado con *flujoS*:

```
int n;
// ...
flujoS.write(n);
```

La segunda versión del método **write** escribe los bytes almacenados en la matriz *b* en un flujo de salida (más adelante, dedicaremos un capítulo a explicar las matrices de datos).

```
byte[] b = new byte[128]; // matriz 'b' de 128 bytes
flujoS.write(b);
```

La tercera versión del método **write** escribe un máximo de *len* bytes de una matriz *b* a partir de su posición *off*, en un flujo de salida.

Cada uno de estos métodos ha sido escrito para que bloquee la ejecución del programa que los invoque hasta que toda la salida solicitada haya sido escrita.

Análogamente, la clase **Writer** es una clase abstracta que es superclase de todas las clases que representan un flujo para escribir caracteres a un destino. Sus

métodos son análogos a los de la clase **OutputStream**, con la diferencia de que utilizan parámetros de tipo **char** en lugar de **byte**.

Excepciones

Cuando durante la ejecución de un programa ocurre un error que impide su continuación, por ejemplo, una entrada incorrecta de datos o una división por cero, Java lanza una excepción, que cuando no se captura da lugar a un mensaje acerca de lo ocurrido y detiene su ejecución (las excepciones se lanzan, no ocurren). Ahora, si lo que deseamos es que la ejecución del programa no se detenga, habrá que capturarla y manejarla adecuadamente en un intento de reanudar la ejecución.

Las excepciones en Java son objetos de subclases de **Throwable**. Por ejemplo, el paquete **java.io** define una clase de excepción general denominada **IOException** para excepciones de entrada salida.

Puesto que en Java hay muchas clases de excepciones, un método puede indicar los tipos de excepciones que posiblemente puede lanzar. Por ejemplo, puede observar que los métodos **read** y **write** que acabamos de exponer lanzan excepciones del tipo **IOException**. Entonces, cuando utilicemos alguno de esos métodos hay que escribir el código necesario para capturar las posibles excepciones que pueden lanzar. Esto es algo a lo que nos obliga el compilador Java, del mismo modo que él verifica si una variable ha sido iniciada antes de ser utilizada, o si el número y tipo de argumentos utilizados con un método son correctos, con la única intención de minimizar los posibles errores que puedan ocurrir.

Para capturar una excepción hay que hacer dos cosas: una, poner a prueba el código que puede lanzar excepciones dentro de un bloque **try**; y dos, manejar la excepción cuando se lance, en un bloque **catch**. Por ejemplo:

```
try
{
    // Código que puede lanzar una excepción
    n = flujoE.read(); // puede lanzar una excepción IOException
}
catch(IOException e)
{
    // Manejar una excepción de la clase IOException
    System.out.println("Error: " + e.getMessage());
}
```

En el ejemplo anterior, el manejo de la excepción se ha reducido a visualizar un mensaje del error ocurrido.

Esto es todo lo que necesita saber por ahora para poder utilizar los métodos involucrados en la E/S que lancen excepciones. Más adelante, dedicaremos un capítulo al estudio de excepciones.

Flujos estándar de E/S

La biblioteca de Java proporciona tres flujos estándar, manipulados por la clase **System** del paquete **java.lang**, que son automáticamente abiertos cuando se inicia un programa y cerrados cuando éste finaliza:

- **System.in**. Referencia a la entrada estándar del sistema, que normalmente coincide con el teclado. Se utiliza para leer datos introducidos por el usuario.
- **System.out**. Referencia a la salida estándar del sistema, que normalmente es el monitor. Se utiliza para mostrar datos al usuario.
- **System.err**. Referencia a la salida estándar de error del sistema, que normalmente es el monitor. Se utiliza para mostrar mensajes de error al usuario.

Los siguientes ejemplos ilustran la utilización de los flujos **in** y **out**:

```
int n;
n = System.in.read();           // entrada por teclado: A
System.out.println(n);          // salida por monitor: 65
```

El método **read** devuelve un entero (**int**) correspondiente al valor ASCII del carácter leído. Ahora este valor puede ser convertido a otro tipo como **byte**:

```
byte b;
b = (byte)System.in.read(); // entrada por teclado: A
System.out.println(b);      // salida por monitor: 65
```

El valor devuelto por el método **read** también puede ser convertido explícitamente al tipo **char** para manipular caracteres:

```
char c;
c = (char)System.in.read(); // entrada por teclado: A
System.out.println(c);     // salida por monitor: A
```

¿Qué otros métodos podemos utilizar con estos flujos? Para dar respuesta a esta pregunta primero tendremos que investigar de qué clases son estos objetos y después, analizar esas clases. Averiguar de qué clases son estos objetos es una tarea simple; basta con revisar la información de la biblioteca de Java, o bien utilizar un objeto **Class** como se indica en el apartado siguiente.

Determinar la clase a la que pertenece un objeto

La clase **Object** del paquete **java.lang** es la clase raíz de la jerarquía de clases de Java. Esto quiere decir que el resto de las clases se deriva directa o indirectamente de esta clase, lo que a su vez significa que todas heredan todos sus miembros. Por lo tanto, todos los objetos disponen de los métodos proporcionados por **Object**.

De los métodos a los que nos hemos referido, nos interesa ahora **getClass**. Para invocar este método puede hacerlo así:

```
Class ObjetoClass = cualquierObjeto.getClass();
```

La línea anterior indica que **getClass** devuelve un objeto de la clase **Class**, *ObjetoClass*, cuyos métodos permitirán obtener información acerca de la clase del objeto referenciado por *cualquierObjeto*. Por ejemplo, el método **getName** devuelve una cadena correspondiente al nombre de la clase; **getMethods** devuelve una matriz de la clase **Method** con los nombres de todos los métodos, etc.

```
import java.io.*;

class ClaseDeUnObj
{
    public static void main(String[] args)
    {
        int n;
        try
        {
            System.out.print("Dato: ");
            n = System.in.read(); // leer un carácter desde el teclado
            System.out.println((char)n); // visualizar el carácter

            // Investigamos
            Class ObjetoClass; // objeto Class
            ObjetoClass = System.in.getClass();
            System.out.println("Clase de in: " + ObjetoClass.getName());
            ObjetoClass = System.out.getClass();
            System.out.println("Clase de out: " + ObjetoClass.getName());
            ObjetoClass = System.err.getClass();
            System.out.println("Clase de err: " + ObjetoClass.getName());
        }
        catch(IOException e)
        {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

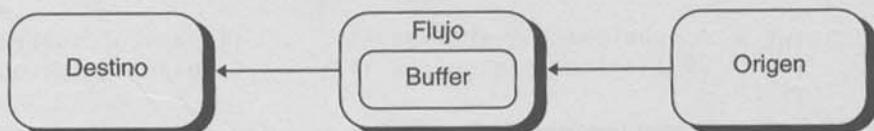
El método **main** de la aplicación anterior, primero solicita un dato que será introducido a través del teclado, después visualiza el dato, y finalmente obtiene y visualiza los nombres de las clases de los objetos correspondientes a los flujos estándar. Cuando ejecute la aplicación el resultado será similar al siguiente:

```
Dato: 1
1
Clase de in: class java.io.BufferedReader
Clase de out: class java.io.PrintStream
Clase de err: class java.io.PrintStream
```

La capacidad de conocer detalles de otras clases (incluidas las nuestras) a través de una clase habilitada por Java se conoce como *reflexión*.

BufferedInputStream

La clase **BufferedInputStream** se deriva indirectamente de **InputStream**, por lo tanto hereda todos los miembros de ésta; por ejemplo, el método **read** expuesto anteriormente. Esta clase, aunque no aporta métodos nuevos, sí aporta una característica muy interesante de la que se benefician todos sus métodos: un *buffer* que actúa como una memoria intermedia para lecturas futuras. Para entender esto observe la figura siguiente:



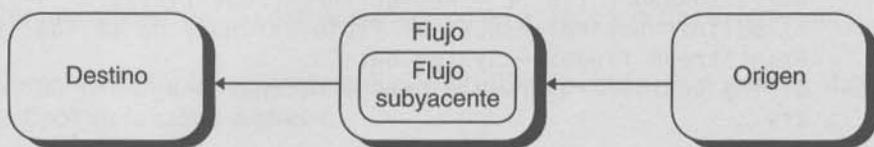
Según el esquema anterior, cuando una aplicación ejecute una sentencia de entrada (que solicite datos) los datos obtenidos del origen pueden ser depositados en el *buffer* en bloques más grandes que los que realmente está leyendo la aplicación (por ejemplo, cuando se leen datos de un disco la cantidad mínima de información transferida es un bloque equivalente a una unidad de asignación). Esto aumenta la velocidad de ejecución porque la siguiente vez que la aplicación necesite más datos no tendrá que esperar por ellos porque ya los tendrá en el *buffer*. Por otra parte, cuando se trate de una operación de salida, los datos no serán enviados al destino hasta que no se llene el *buffer* (o hasta que se fuerce el vaciado del mismo implícita o explícitamente), lo que reduce el número de accesos al dispositivo físico vinculado que siempre resulta mucho más lento que los accesos a memoria, aumentando por consiguiente la velocidad de ejecución.

Cuando el origen es el teclado y el destino el programa, el esquema es el mismo. Esto permite introducir los datos por anticipado para una aplicación en ejecución de la que se sabe que más adelante va a solicitarlos a través del teclado.

La clase análoga a **BufferedInputStream**, pero que permite trabajar con caracteres es **BufferedReader**, clase derivada de **Reader**.

BufferedReader

Bajo un flujo de la clase **BufferedReader** subyace otro flujo de caracteres (objeto de una clase derivada de **Reader**), o bien de bytes (objeto de una clase derivada de **InputStream**). Esto es, cada petición de lectura hecha a un flujo de la clase **BufferedReader** es dirigida a otro flujo de caracteres o de bytes subyacente.



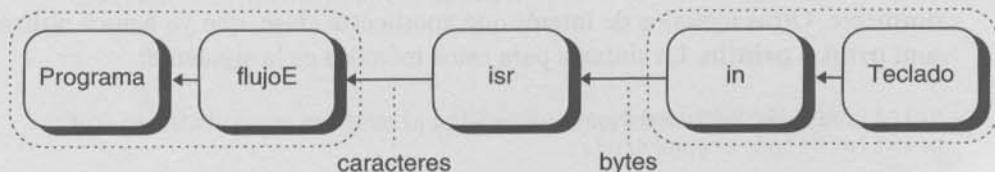
La figura anterior traducida a código dependiente de los flujos mencionados en el párrafo anterior, puede interpretarse así:

```
BufferedReader flujoE = new BufferedReader(isr);
```

El código anterior indica que el *flujoE* dirigirá todas las invocaciones de sus métodos al flujo subyacente *isr*; este flujo, en el caso de que el origen sea el teclado (dispositivo vinculado con **System.in**), deberá convertir los bytes leídos del teclado en caracteres. De esta forma *flujoE* podrá suministrar un flujo de caracteres al programa destino de los datos. Para ello hay que definir el flujo que hemos denominado *isr* así:

```
InputStreamReader isr = new InputStreamReader(System.in);
```

La clase **InputStreamReader** establece un puente para pasar flujos de bytes a flujos de caracteres.



La clase **BufferedReader** proporciona métodos análogos a **BufferedInputStream**, y además otros como **readLine**. Este método permite leer una línea de texto que devuelve en un objeto de la clase **String**. Se entiende por línea de texto la cadena formada por los caracteres que hay hasta encontrar uno de los siguientes: ‘\r’, ‘\n’ o ambos; estos caracteres son leídos pero no almacenados.

Como ejemplo, vamos a realizar una aplicación que lea una línea de texto introducida a través del teclado y la visualice en la pantalla.

```
import java.io.*;

public class LeerUnaCadena
{
    public static void main(String[] args)
    {
        // Definir un flujo de caracteres de entrada: flujoE
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader flujoE = new BufferedReader(isr);
        // Definir una referencia al flujo estándar de salida: flujoS
        PrintStream flujoS = System.out;
        String sdato; // variable para almacenar una línea de texto
        try
        {
            flujoS.print("Introduzca un texto: ");
            sdato = flujoE.readLine(); // leer una línea de texto
            flujoS.println(sdato); // escribir la línea leída
        }
        catch (IOException ignorada) { }
    }
}
```

Analicemos el método **main** de la aplicación anterior. Primeramente define un flujo de entrada, *flujoE*, del cual se podrán leer líneas de texto. Después, se define una referencia, *flujoS*, al flujo de salida estándar; esto permitirá utilizar la referencia *flujoS* en lugar de **System.out**. La última parte del cuerpo de **main** lee una línea de texto introducida a través del teclado y la visualiza.

PrintStream

La clase **PrintStream** se deriva indirectamente de **OutputStream**, por lo tanto hereda todos los miembros de ésta; por ejemplo el método **write** expuesto anteriormente. Otros métodos de interés que aporta esta clase, que ya hemos utilizado, son: **print** y **println**. La sintaxis para estos métodos es la siguiente:

```
print(tipo argumento);
println([tipo argumento]);
```

Los métodos **print** y **println** son esencialmente los mismos; ambos escriben su argumento en el flujo de salida. La única diferencia entre ellos es que **println** añade un carácter ‘\n’ (avance a la línea siguiente) al final de su salida, y **print** no. En otras palabras, la siguiente sentencia:

```
System.out.print("El valor no puede ser negativo\n");
```

es equivalente a esta otra:

```
System.out.println("El valor no puede ser negativo");
```

En el ejemplo anterior, se puede observar que **print** añade al final de la cadena de caracteres un carácter ‘\n’ que **println** no añade.

Los argumentos para **print** y **println** pueden ser de cualquier tipo primitivo o referenciado: **Object**, **String**, **char[]**, **int**, **long**, **float**, **double**, y **boolean**. En adición, hay una versión extra de **println** que no tiene argumentos y lo que hace es escribir un carácter ‘\n’, lo que se traduce en un avance a la línea siguiente.

Como ejemplo, la siguiente aplicación utiliza **println** para escribir datos de varios tipos en la salida estándar.

```
public class TestTiposDatos
{
    // Tipos de datos
    public static void main(String[] args)
    {
        String sCadena = "Lenguaje Java";
        char[] cMatrizCars = { 'a', 'b', 'c' }; // matriz de caracteres
        int dato_int = 4;
        long dato_long = Long.MIN_VALUE;      // mínimo valor long
        float dato_float = Float.MAX_VALUE;   // máximo valor float
        double dato_double = Math.PI;         // 3.1415926
        boolean dato_boolean = true;

        System.out.println(sCadena);
        System.out.println(cMatrizCars);
        System.out.println(dato_int);
        System.out.println(dato_long);
        System.out.println(dato_float);
        System.out.println(dato_double);
        System.out.println(dato_boolean);
    }
}
```

Los resultados que produce la aplicación anterior son los siguientes:

```
Lenguaje Java
abc
4
-9223372036854775808
3.4028235E38
3.141592653589793
true
```

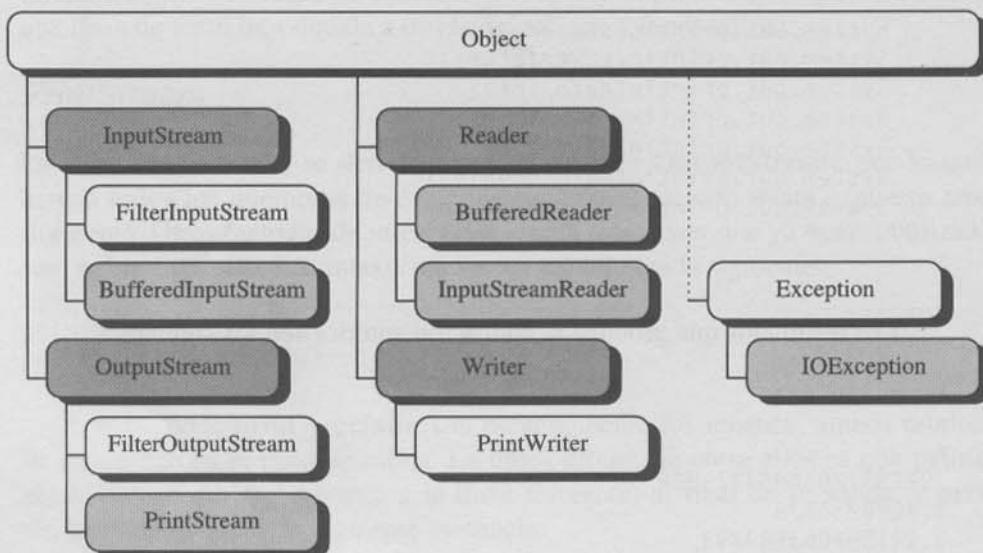
Observe que se puede imprimir un objeto; el primer método **println** imprime un objeto **String**. Cuando se utiliza **print** o **println** para imprimir un objeto, el dato impreso depende del tipo del objeto. En el ejemplo se puede observar que la impresión de un objeto **String** hace que se imprima la cadena de caracteres que almacena. Sin embargo, la impresión de un objeto **Class** daría lugar a que se imprimiera una cadena de caracteres correspondiente al nombre de la clase del objeto que estuviera siendo inspeccionado.

La clase análoga a **PrintStream** es **PrintWriter**, clase derivada de **Writer**, pero los métodos proporcionados por ambas son prácticamente los mismos, por lo que no comentaremos esta última. Una diferencia entre ambas es que cuando se ejecuta un método de **PrintStream**, el *buffer* de salida es vaciado automáticamente (los datos se muestran), no sucediendo lo mismo con **PrintWriter**; en este caso habría que forzar el vaciado del *buffer* de salida invocando a su método **flush**. Por ejemplo:

```
PrintWriter flujoS = new PrintWriter(System.out);
int dato_int = 4;
flujoS.println(dato_int); flujoS.flush();
```

Trabajar con tipos de datos primitivos

Hagamos un breve recorrido por la jerarquía de clases vista desde un esquema gráfico para ver dónde se sitúan las que hemos comentado:



En la figura anterior se pueden observar en color gris las clases comentadas en este capítulo; las coloreadas en gris más oscuro son clases abstractas (una línea discontinua indica que esa clase no se deriva directamente de **Object**; esto es, entre **Object** y la clase hay otras clases que no tienen interés para el tema que estamos tratando).

Después de analizar la jerarquía de clases para, entre otras cosas, llegar a ver la procedencia de los flujos **in** y **out**, se deduce que para leer del flujo **in** sólo se dispone de métodos que proporcionan un carácter, o bien una matriz de caracteres; para leer una cadena de caracteres del flujo **in** y almacenarla en un objeto **String** lo tenemos que hacer desde un flujo de la clase **BufferedReader**; y para escribir en el flujo **out** tenemos los métodos proporcionados por la clase **PrintStream**, o bien **PrintWriter**, que permiten escribir cualquier valor de cualquier tipo primitivo o referenciado.

Evidentemente, cualquier operación aritmética requiere de valores numéricos; pero, según lo expuesto, en el mejor de los casos sólo se puede obtener una cadena de bytes. El código siguiente pertenece a la aplicación *LeerUnaCadena* realizado anteriormente:

```
flujoS.print("Introduzca un texto: ");
sdato = flujoE.readLine(); // leer una línea de texto
```

El código anterior permite leer del flujo **in** una cadena de caracteres que será almacenada en el objeto **sdato** de tipo **String**. Por ejemplo, si cuando se ejecute el método **readLine** se teclea el dato **456**, estos dígitos serán almacenados en **sdato** como una cadena de caracteres. Ahora bien, para que esa cadena de tres caracteres pueda ser utilizada en una expresión aritmética, tiene que adquirir la categoría de valor numérico, lo que implica convertirla a un valor de alguno de los tipos primitivos. Esto puede hacerse utilizando los métodos proporcionados por las clases que encapsulan los tipos primitivos.

Clases que encapsulan los tipos primitivos

El paquete **java.lang** proporciona las clases **Byte**, **Character**, **Short**, **Integer**, **Long**, **Float**, **Double** y **Boolean**, que encapsulan cada uno de los tipos primitivos, proporcionando así una funcionalidad añadida para manipularlos. Analicemos, por ejemplo, la clase **Integer**.

Un objeto de la clase **Integer** contiene un atributo de tipo **int**, que es el objetivo de la clase. Además, proporciona otros atributos y varios métodos útiles para tratar con un entero; por ejemplo, para convertir un **int** en un **String** o un **String** en un **int**. Algunos de ellos son los siguientes:

<i>Atributo</i>	<i>Descripción</i>
MIN_VALUE	Valor más pequeño de tipo int .
MAX_VALUE	Valor más grande de tipo int .
<i>Método</i>	<i>Descripción</i>
doubleValue()	Devuelve el objeto Integer como un valor double .
floatValue()	Devuelve el objeto Integer como un valor float .
intValue()	Devuelve el objeto Integer como un valor int .
longValue()	Devuelve el objeto Integer como un valor long .
parseInt(String)	Convierte una cadena a un valor int .
toString(int)	Convierte un valor int en una cadena (objeto String).
valueOf(String)	Crea un objeto Integer a partir de una cadena.

El resto de las clases tienen métodos análogos. No obstante, es necesario resaltar que las clases **Float** y **Double** no tienen un método **parse...**. En cambio, incluyen otros atributos que pueden observarse en la tabla siguiente. Por ejemplo, para la clase **Float** (ídem para la clase **Double**):

<i>Atributo</i>	<i>Descripción</i>
MIN_VALUE	Valor más pequeño de tipo float .
MAX_VALUE	Valor más grande de tipo float .
NaN	No es un Número; de tipo float .
NEGATIVE_INFINITY	Valor infinito negativo de tipo float .
POSITIVE_INFINITY	Valor infinito positivo de tipo float .

De acuerdo con lo expuesto, para obtener, por ejemplo, un entero a partir de una cadena de caracteres proporcionada por **readLine** habrá que ejecutar los siguientes pasos:

1. Definir un flujo de entrada de la clase **BufferedReader**.
2. Leer la cadena de caracteres.
3. Convertir el objeto **String** en un entero.

El siguiente código corresponde a los puntos enunciados:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader flujoE = new BufferedReader(isr);
```

```

String sdato; // variable para almacenar una cadena
int dato_int; // variable para almacenar un entero
try
{
    sdato = flujoE.readLine(); // leer una cadena de caracteres
    dato_int = Integer.parseInt(sdato); // convertir a entero
}
catch (IOException ignorada) { }

```

En el ejemplo anterior se observa que una vez leída la cadena *sdato*, que se supone es una cadena válida para ser convertida en un entero, se invoca al método estático **parseInt** para convertir el objeto **String** en un dato de tipo **int**.

Análogamente, para convertir una cadena de bytes que representa un número con punto decimal, en un valor de tipo **float**, el código sería el siguiente:

```

sdato = flujoE.readLine();           // leer una cadena de caracteres
Float f = new Float(sdato);         // crear un objeto Float
float dato_float = f.floatValue();   // obtener el valor float

```

En el ejemplo anterior se observa que al no disponer la clase **Float** de un método análogo a **parseInt**, se ha tenido que recurrir a crear un objeto **Float** a partir de la cadena de caracteres, para después obtener el valor **float** que encapsula dicho objeto.

Según lo expuesto, podemos escribir un método *dato* que lea una cadena de caracteres desde el teclado, la almacene en un objeto **String** y devuelva como resultado dicho objeto.

```

String dato()
{
    String sdato = "";
    try
    {
        // Definir un flujo de caracteres de entrada: flujoE
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader flujoE = new BufferedReader(isr);
        // Leer. La entrada finaliza al pulsar la tecla Entrar
        sdato = flujoE.readLine();
    }
    catch(IOException e)
    {
        System.err.println("Error: " + e.getMessage());
    }
    return sdato; // devolver el dato tecleado
}

```

Partiendo de la cadena devuelta por el método anterior, podemos escribir también, por ejemplo, un método *datoInt* que la convierta en un número entero y devuelva este valor como resultado:

```
int datoInt()
{
    String sdato = dato();           // invoca al método dato
    return Integer.parseInt(sdato); // convierte sdato en un int
}
```

Si el valor devuelto por el método *dato* no fuera válido para convertirlo en un número entero, el método **parseInt** lanzaría una excepción de tipo **NumberFormatException** que podríamos manejar. Para no complicar el tema que estamos exponiendo, y puesto que dedicaremos un capítulo posterior a explicar las excepciones, si se lanza una excepción del tipo descrito, el método simplemente devolverá un valor significativo (una constante miembro de la clase). Según esto podríamos modificar el método anterior así:

```
int datoInt()
{
    try
    {
        return Integer.parseInt(dato());
    }
    catch(NumberFormatException e)
    {
        return Integer.MIN_VALUE; // valor más pequeño de tipo int
    }
}
```

Observe que el argumento del método **parseInt** es la cadena de caracteres devuelta por el método *dato*. Si ocurre un error, por ejemplo, porque se introduce una cadena que no es convertible a un número entero, el sistema lanzará una excepción de tipo **NumberFormatException** que será atrapada por el bloque **catch** lo que dará lugar a que el método *datoInt* devuelva el valor *MIN_VALUE*, definido como una constante de la clase.

Análogamente, podemos escribir otros métodos para convertir una cadena válida, devuelta por el método *dato*, en otros tipos de datos primitivos. Agrupemos todos estos métodos en una clase denominada *Leer*.

Clase Leer

El objetivo es escribir una clase *Leer* que incluya como miembros, además de los métodos que hemos venido implementando anteriormente, otros métodos, de ma-

nera que todos juntos proporcionen una interfaz que cualquier programa puede utilizar para obtener del teclado datos de cualquier tipo primitivo. El código que define esta clase se muestra a continuación.

```
import java.io.*;  
  
public class Leer  
{  
    public static String dato()  
    {  
        String sdato = "";  
        try  
        {  
            // Definir un flujo de caracteres de entrada: flujoE  
            InputStreamReader isr = new InputStreamReader(System.in);  
            BufferedReader flujoE = new BufferedReader(isr);  
            // Leer. La entrada finaliza al pulsar la tecla Entrar  
            sdato = flujoE.readLine();  
        }  
        catch(IOException e)  
        {  
            System.err.println("Error: " + e.getMessage());  
        }  
        return sdato; // devolver el dato tecleado  
    }  
  
    public static short datoShort()  
    {  
        try  
        {  
            return Short.parseShort(dato());  
        }  
        catch(NumberFormatException e)  
        {  
            return Short.MIN_VALUE; // valor más pequeño  
        }  
    }  
  
    public static int datoInt()  
    {  
        try  
        {  
            return Integer.parseInt(dato());  
        }  
        catch(NumberFormatException e)  
        {  
            return Integer.MIN_VALUE; // valor más pequeño  
        }  
    }  
}
```

```

public static long datoLong()
{
    try
    {
        return Long.parseLong(dato());
    }
    catch(NumberFormatException e)
    {
        return Long.MIN_VALUE; // valor más pequeño
    }
}

public static float datoFloat()
{
    try
    {
        Float f = new Float(dato());
        return f.floatValue();
    }
    catch(NumberFormatException e)
    {
        return Float.NaN; // No es un Número; valor float.
    }
}

public static double datoDouble()
{
    try
    {
        Double d = new Double(dato());
        return d.doubleValue();
    }
    catch(NumberFormatException e)
    {
        return Double.NaN; // No es un Número; valor double.
    }
}
}

```

En la clase *Leer*, se puede observar que todos los métodos, además de públicos, se han declarado **static** con el fin de que puedan ser invocados allí donde se necesiten, sin necesidad de que exista un objeto de la clase. Recuerde que la sintaxis para invocar a un método de una clase es:

nombreClase.nombreMétodo

Una vez escrita la clase *Leer*, podemos utilizarla como soporte para otras aplicaciones. Como ejemplo, vamos a escribir una aplicación que lea un dato de cada uno de los tipos contemplados en *Leer* y muestre después los valores leídos.

Recuerde que para que la clase aplicación que vamos a escribir pueda utilizar la clase *Leer*, deben estar ambas en la misma carpeta de trabajo.

```
// Utiliza la clase Leer que debe de estar almacenada
// en la misma carpeta

public class LeerDatos
{
    public static void main(String[] args)
    {
        short dato_short = 0;
        int dato_int = 0;
        long dato_long = 0;
        float dato_float = 0;
        double dato_double = 0;

        System.out.print("Dato short: ");
        dato_short = Leer.datoShort();
        System.out.print("Dato int: ");
        dato_int = Leer.datoInt();
        System.out.print("Dato long: ");
        dato_long = Leer.datoLong();
        System.out.print("Dato float: ");
        dato_float = Leer.datoFloat();
        System.out.print("Dato double: ");
        dato_double = Leer.datoDouble();

        System.out.println(dato_short);
        System.out.println(dato_int);
        System.out.println(dato_long);
        System.out.println(dato_float);
        System.out.println(dato_double);
    }
}
```

Después del trabajo realizado, ya tenemos una forma de leer datos numéricos introducidos a través del teclado. Esto nos permitirá escribir diversas aplicaciones que requieren de este proceso. Además, sabemos también cómo convertir números a cadenas de caracteres y viceversa.

¿DÓNDE SE UBICAN LAS CLASES QUE DAN SOPORTE?

Para que Java pueda utilizar una clase debe conocer dónde está almacenada en el sistema de ficheros. De otra forma, cuando se compile el programa se obtendrá un error indicando que esa clase no existe. Java utiliza dos elementos para localizar las clases: el nombre del paquete y las rutas especificadas por la variable *CLASSPATH*.

Variable CLASSPATH

Cuando en el código fuente de un programa se hace referencia a una clase que no pertenece a un paquete que se pueda importar, como ocurre con la clase *Leer*, Java busca por ella en el directorio actual si la variable *CLASSPATH* no ha sido establecida. En otro caso busca en las rutas especificadas por esta variable.

Si recuerda, en el capítulo 1, al explicar cómo se compilaba y ejecutaba un programa, se dijo que había que establecer la variable de entorno *PATH*. Pues bien, para establecer la variable *CLASSPATH* proceda de forma análoga. Por ejemplo, si almacenamos las clases que vayan a ser compartidas por otros programas, como es el caso de la clase *Leer*, en la carpeta *jdk1.3\misClases*, asigne a la variable *CLASSPATH* esta ruta:

```
CLASSPATH=c:.\jdk1.3\misClases
```

El ejemplo anterior indica a Java que busque las clases a las que haga referencia un determinado programa, además de en los paquetes importados, en la carpeta actual de trabajo (.) o en la carpeta *jdk1.3\misClases*.

CARÁCTER FIN DE FICHERO

Desde el punto de vista de un usuario de una aplicación, un dispositivo de entrada o de salida estándar es tratado por el lenguaje Java como si de un fichero de datos en el disco se tratara. Un fichero de datos no es más que una colección de información. Los datos que introducimos por el teclado son una colección de información y los datos que visualizamos en el monitor son también una colección de información.



Todo fichero tiene un principio y un final. ¿Cómo sabe un programa que está leyendo datos de un fichero, que se ha llegado al final del mismo y por lo tanto no hay más datos? Por una marca de fin de fichero. En el caso de un fichero grabado en un disco esa marca estará escrita al final del mismo. En el caso del teclado la información procede de lo que nosotros tecleamos, por lo tanto si nuestro programa requiere detectar la marca de fin de fichero, tendremos que teclearla cuando demos por finalizada la introducción de información. Esto se hace pulsando las teclas *Ctrl+D* en UNIX o *Ctrl+Z* en una aplicación de consola en Windows.

Ya que un fichero o un dispositivo siempre es manejado a través de un flujo, hablar del final del flujo es sinónimo de hablar del fichero. Por eso, de ahora en adelante nos referiremos al flujo en lugar de al fichero o dispositivo vinculado.

Recuerde que cuando el método **read** intenta leer y se encuentra con el final del flujo, retorna la constante **-1**. Análogamente, cuando el método **readLine** intenta leer del flujo y se encuentra con el final del mismo, retorna la constante **null**. Para aclarar lo expuesto, el siguiente ejemplo solicita del teclado un dato *precio*. Entonces, si al mensaje “Precio:” respondemos escribiendo una cantidad, la variable *precio* almacenará ese valor, pero si respondemos pulsando las teclas **Ctrl+Z** (carácter fin de fichero), deberá almacenar el valor *NaN* de tipo **float**.

```
import java.io.*;
public class Test
{
    public static void main(String[] args)
    {
        // Definir un flujo de caracteres de entrada: flujoE
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader flujoE = new BufferedReader(isr);
        // Definir una referencia al flujo estándar de salida: flujoS
        PrintStream flujoS = System.out;

        String sdato;
        float precio = 0.0F;
        try
        {
            flujoS.print("Precio: ");
            sdato = flujoE.readLine();
            precio = (sdato != null)
                ? (new Float(sdato)).floatValue()
                : Float.NaN;
        }
        catch (IOException ignorada){ }
        flujoS.println(precio);
        flujoS.println("Continua la aplicación");
    }
}
```

Cuando ejecute esta aplicación puede proceder de cualquiera de las dos formas siguientes:

1. Introduciendo un dato válido:

```
Precio: 123.45
123.45
Continua la aplicación
```

2. Pulsando las teclas *Ctrl+Z* (marca de final del flujo):

```
Precio: (se pulsan las teclas Ctrl+Z)
NaN
Continua la aplicación
```

Una aclaración. La aplicación anterior utiliza el operador ternario (?:) para verificar si se llegó al final del flujo, lo que sucederá cuando se pulsen las teclas *Ctrl+Z*. La expresión booleana *sdato != null* será **true** si se introdujo un dato válido, en cuyo caso se asignará a *precio* el resultado de la expresión (*new Float(sdato).floatValue()*); y será **false** si se pulsaron las teclas *Ctrl+Z*, en cuyo caso se asignará a *precio* el valor *NaN*.

La expresión (*new Float(sdato).floatValue()*) es equivalente a:

```
Float f = new Float(sdato);
precio = f.floatValue();
```

En capítulos posteriores utilizaremos *Ctrl+Z* como condición para finalizar la entrada de un número de datos, en principio indeterminado.

CARACTERES \r\n

Cuando se están introduciendo datos a través del teclado y pulsamos la tecla *Entrar* se introducen también los caracteres \r\n, correspondientes a los caracteres ASCII CR LF (CR es el ASCII 13 y LF es el ASCII 10). Mientras que en la salida \n produce un CR+LF, en la entrada se corresponde con un LF; esto es, una expresión Java como '\n' == 10 daría como resultado **true**.

Por ejemplo, suponiendo definidos los flujos *flujoE* y *flujoS* igual que en el ejemplo anterior, el código siguiente lee un carácter:

```
char opción;
try
{
    flujoS.print("Opción (a, b o c): ");
    opción = (char)flujoE.read();
}
catch (IOException ignorada){ }
```

Cuando se ejecute el método **read** de la aplicación anterior, si tecleamos la opción *b* y pulsamos la tecla *Entrar*:

```
b[Entrar]
```

antes de la lectura, el *buffer* de entrada contendrá la siguiente información:

b	\r	\n									
---	----	----	--	--	--	--	--	--	--	--	--

y después de la lectura:

\r	\n									
----	----	--	--	--	--	--	--	--	--	--

ya que **read** lee un solo carácter. Estos caracteres sobrantes pueden ocasionarnos problemas si a continuación se ejecuta otra sentencia de entrada que admite datos que sean caracteres. Por ejemplo:

```
char opción;
String sdato;
try
{
    flujoS.print("Opción (a, b o c): ");
    opción = (char)flujoE.read();

    flujoS.print("Precio: ");
    sdato = flujoE.readLine();
    flujoS.println("Continua la aplicación");
}
catch (IOException ignorada){}
```

Si ejecutamos esta aplicación y tecleamos, por ejemplo, como opción *b* seguida de la pulsación de la tecla *Entrar*, se producirá el siguiente resultado:

```
Opción (a, b o c): b
Precio: Continua la aplicación
```

A la vista del resultado, se observa que cuando se ejecutó **readLine** no se detuvo la ejecución de la aplicación para introducir el dato solicitado ¿Por qué? Porque los caracteres sobrantes \r y \n son válidos para el método **readLine**. Recuerde que este método permite leer una cadena de caracteres hasta encontrar '\r' (CR), '\n' (LF) o '\r\n' (CRLF); estos caracteres son leídos pero no almacenados. Por este motivo es por lo que este método no necesita esperar a que introduzcamos un carácter para la variable *sdato*. Recuerde también, que cuando explicamos anteriormente las clases **BufferedInputStream** y **BufferedReader** dijimos que un *buffer* permitía, entre otras cosas, introducir los datos por anticipado. En este caso, nuestra intención no era ésa, pero la forma en la que hemos introducido un dato nos han conducido a ello.

La solución al problema planteado es limpiar los caracteres indeseables del *buffer* de entrada. Hay dos formas sencillas de hacer esto. Una es utilizar el propio método **readLine** para hacer una lectura “en falso”, con la única intención de extraer todos los caracteres que haya, y otra es utilizar los métodos **skip** y **available**.

El método **skip** permite saltar *n* caracteres en el flujo de entrada para que no estén presentes en la próxima operación de lectura; y el método **available** devuelve el número de caracteres que hay disponibles en el flujo de entrada. Por ejemplo:

```
char opción;
int ncars;
String sdato;
try
{
    flujoS.print("Opción (a, b o c): ");
    opción = (char)flujoE.read();
    ncars = flujoE.available(); // caracteres disponibles
    flujoE.skip(ncars); // saltar los caracteres CR LF

    flujoS.print("Precio: ");
    sdato = flujoE.readLine();
    flujoS.println("Continua la aplicación");
}
catch (IOException ignorada){}
```

Un *buffer* se limpia automáticamente cuando está lleno, cuando se cierra el flujo, o bien cuando el programa finaliza normalmente.

MÉTODOS MATEMÁTICOS

La biblioteca de clases de Java incluye una clase llamada **Math** en su paquete **java.lang**, la cual define un conjunto de operaciones matemáticas de uso común que pueden ser utilizadas por cualquier programa.

La clase **Math** contiene métodos para ejecutar operaciones numéricas elementales tales como raíz cuadrada, exponencial, logaritmo, y funciones trigonométricas. Por ejemplo:

```
double raíz_cuadrada, n = 345.0;
raíz_cuadrada = Math.sqrt(n);
System.out.println("La raíz cuadrada de " + n + " es " + raíz_cuadrada);
```

La tabla siguiente resume los miembros de la clase **Math**. Todos los miembros de esta clase son **static** para que puedan ser invocados sin necesidad de definir un objeto de la clase.

<i>Método</i>	<i>Descripción</i>
static double E	Valor del número <i>e</i> (base del logaritmo neperiano o natural).

double PI	Valor del número π .
<i>tipo abs(tipo a)</i>	Valor absoluto de <i>a</i> . El <i>tipo</i> , igual en todos los casos, puede ser: double , float , int o long .
double ceil(double a)	Valor double sin decimales más pequeño que es mayor o igual que <i>a</i> .
double floor(double a)	Valor double sin decimales más grande que es menor o igual que <i>a</i> .
<i>tipo max(tipo a, tipo b)</i>	Valor mayor de <i>a</i> y <i>b</i> . El <i>tipo</i> , igual en todos los casos, puede ser: double , float , int o long .
<i>tipo min(tipo a, tipo b)</i>	Valor menor de <i>a</i> y <i>b</i> . El <i>tipo</i> , igual en todos los casos, puede ser: double , float , int o long .
double random()	Valor aleatorio mayor o igual que <i>0.0</i> y menor que <i>1.0</i> .
double rint(double a)	Valor double sin decimales más cercano a <i>a</i> (redondeo de <i>a</i>).
long round(double a)	Valor long más cercano a <i>a</i> .
int round(float a)	Valor int más cercano a <i>a</i> .
double sqrt(double a)	Raíz cuadrada de <i>a</i> (<i>a</i> no puede ser negativo).
double exp(double a)	Valor de e^a .
double log(double a)	Logaritmo neperiano (natural) de <i>a</i> .
double pow(double a, double b)	Valor de a^b .
double IEEERemainder(double f1, double f2)	Resto de una división entre números reales: $c=f1/f2$, siendo <i>c</i> el valor <u>entero</u> más cercano al valor real de $f1/f2$; por lo tanto, el resto puede ser positivo o negativo.
double acos(double a)	Arco, de <i>0.0</i> a π , cuyo coseno es <i>a</i> .
double asin(double a)	Arco, de $-\pi/2$ a $\pi/2$, cuyo seno es <i>a</i> .
double atan(double a)	Arco, de $-\pi/2$ a $\pi/2$, cuya tangente es <i>a</i> .
double atan2(double a, double b)	Convierte las coordenadas rectangulares (<i>b</i> , <i>a</i>) a polares: (r, θ) .
double sin(double a)	Seno de <i>a</i> radianes.
double cos(double a)	Coseno de <i>a</i> radianes.
double tan(double a)	Tangente de <i>a</i> radianes.
double toDegrees(double rads)	Convertir un ángulo en radianes a grados.
double toRadians(double grados)	Convertir un ángulo en grados a radianes.

EJERCICIOS RESUELTOS

- Realizar una aplicación que dé como resultado los intereses producidos y el capital total acumulado de una cantidad c , invertida a un interés r durante t días.

La fórmula utilizada para el cálculo de los intereses es:

$$I = \frac{c * r * t}{360 * 100}$$

siendo:

I = Total de intereses producidos.

c = Capital.

r = Tasa de interés nominal en tanto por ciento.

t = Período de cálculo en días.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double c, intereses, capital;
float r;
int t;
```

- A continuación leemos los datos c , r y t .

```
System.out.print("Capital invertido: ");
c = Leer.datoDouble();
System.out.print("\nA un % anual del: ");
r = Leer.datoFloat();
System.out.print("\nDurante cuántos días: ");
t = Leer.datoInt();
```

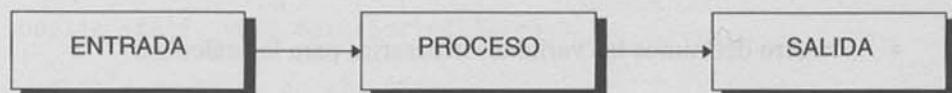
- Conocidos los datos, realizamos los cálculos. Nos piden los intereses producidos y el capital acumulado. Los intereses producidos los obtenemos aplicando directamente la fórmula. El capital acumulado es el capital inicial más los intereses producidos.

```
intereses = c * r * t / (360 * 100);
capital = c + intereses;
```

- Finalmente, escribimos el resultado.

```
System.out.println("Intereses producidos... " + intereses);
System.out.println("Capital acumulado..... " + capital);
```

Observe que el desarrollo de una aplicación, en general, consta de tres bloques colocados en el siguiente orden:



La aplicación completa se muestra a continuación. Observar que se ha utilizado para la entrada de datos los métodos de la clase *Leer* implementada anteriormente en este mismo capítulo.

```

// La clase Leer debe estar en alguna carpeta de las especificadas
// por la variable de entorno CLASSPATH.
public class CIntereses
{
    public static void main(String[] args)
    {
        double c, intereses, capital;
        float r;
        int t;
        System.out.print("Capital invertido: ");
        c = Leer.datoDouble();
        System.out.print("\nA un % anual del: ");
        r = Leer.datoFloat();
        System.out.print("\nDurante cuántos días: ");
        t = Leer.datoInt();

        intereses = c * r * t / (360 * 100);
        capital = c + intereses;

        System.out.println("Intereses producidos... " + intereses);
        System.out.println("Capital acumulado..... " + capital);
    }
}
  
```

2. Realizar una aplicación que dé como resultado las soluciones reales x_1 y x_2 de una ecuación de segundo grado, de la forma:

$$ax^2 + bx + c = 0$$

Las soluciones de una ecuación de segundo grado vienen dadas por la fórmula:

$$x_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Las soluciones son reales sólo si $b^2 - 4ac$ es mayor o igual que 0. Con lo aprendido hasta ahora, la solución de este problema puede desarrollarse de la forma siguiente:

- Primero definimos las variables necesarias para los cálculos:

```
double a, b, c, d, x1, x2;
```

- A continuación leemos los coeficientes a , b y c de la ecuación:

```
System.out.print("Coeficiente a: "); a = Leer.datoDouble();
System.out.print("Coeficiente b: "); b = Leer.datoDouble();
System.out.print("Coeficiente c: "); c = Leer.datoDouble();
```

- Nos piden calcular las raíces reales. Para que existan raíces reales tiene que cumplirse que $b^2 - 4ac \geq 0$; si no, las raíces son complejas conjugadas. Entonces, si hay raíces reales las calculamos; en otro caso, salimos de la aplicación.

Para salir de una aplicación, en general para salir de un proceso sin hacer nada más, Java proporciona la sentencia **return**.

```
d = b * b - 4 * a * c;
if (d < 0)
{
    // Si d es menor que 0
    System.out.println("Las raíces son complejas.");
    return; // salir
}
// Si d es mayor o igual que 0
System.out.println("Las raíces reales son:");
```

- Si hay raíces reales las calculamos aplicando la fórmula.

```
d = Math.sqrt(d);
x1 = (-b + d) / (2 * a);
x2 = (-b - d) / (2 * a);
```

El método **sqrt** calcula la raíz cuadrada de su argumento. En el ejemplo, se calcula la raíz cuadrada de d y se almacena el resultado de nuevo en d .

- Por último escribimos los resultados obtenidos.

```
System.out.println("x1 = " + x1 + ", x2 = " + x2);
```

La aplicación completa se muestra a continuación:

```

// La clase Leer debe estar en alguna carpeta de las especificadas
// por la variable de entorno CLASSPATH.
public class CEquacion
{
    public static void main(String[] args)
    {
        double a, b, c, d, x1, x2;

        System.out.print("Coeficiente a: "); a = Leer.datoDouble();
        System.out.print("Coeficiente b: "); b = Leer.datoDouble();
        System.out.print("Coeficiente c: "); c = Leer.datoDouble();

        d = b * b - 4 * a * c;
        if (d < 0)
        {
            // Si d es menor que 0
            System.out.println("Las raíces son complejas.");
            return; // salir
        }
        // Si d es mayor o igual que 0
        System.out.println("Las raíces reales son:");
        d = Math.sqrt(d);
        x1 = (-b + d) / (2 * a);
        x2 = (-b - d) / (2 * a);
        System.out.println("x1 = " + x1 + ", x2 = " + x2);
    }
}

```

EJERCICIOS PROPUESTOS

1. Realizar una aplicación que calcule el volumen de una esfera, que viene dado por la fórmula:

$$V = \frac{4}{3} \pi r^3$$

2. Realizar una aplicación que pregunte el nombre y el año de nacimiento y dé como resultado:

Hola nombre, en el año 2030 tendrás n años

3. Realizar una aplicación que evalúe el polinomio

$$p = 3x^5 - 5x^3 + 2x - 7$$

y visualizar el resultado con el siguiente formato:

Para x = valor, $3x^5 - 5x^3 + 2x - 7 =$ resultado

4. Realizar la misma aplicación anterior, pero empleando ahora coeficientes variables *a*, *b* y *c*.
5. Ejecute la siguiente aplicación, explique lo que ocurre y realice las modificaciones que sean necesarias para su correcto funcionamiento.

```
import java.io.*;

public class Test
{
    public static void main(String[] args)
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader flujoE = new BufferedReader(isr);
        PrintStream flujoS = System.out;
        char car = 0;
        try
        {
            flujoS.print("Carácter: ");
            car = (char)flujoE.read();
            flujoS.println(car);
            flujoS.print("Carácter: ");
            car = (char)flujoE.read();
            flujoS.println(car);
        }
        catch(IOException ignorada) {}
    }
}
```

6. Indique qué resultado da la siguiente aplicación. A continuación ejecute la aplicación y compare los resultados.

```
import java.io.*;

public class Test
{
    public static void main(String[] args)
    {
        PrintStream flujoS = System.out;

        char car1 = 'A', car2 = 65, car3 = 0;

        car3 = (char)(car1 + 'a' - 'A');
        flujoS.println(car3 + " " + (int)car3);
        car3 = (char)(car2 + 32);
        flujoS.println(car3 + " " + (int)car3);
    }
}
```

CAPÍTULO 6

© F.J.Ceballos/RA-MA

SENTENCIAS DE CONTROL

Cada método de las aplicaciones que hemos hecho hasta ahora, era un conjunto de sentencias que se ejecutaban en el orden en el que se habían escrito, entendiendo por sentencia una secuencia de expresiones que especifica una o varias operaciones. Pero esto no es siempre así; seguro que en algún momento nos ha surgido la necesidad de ejecutar unas sentencias u otras en función de unos criterios especificados por nosotros. Por ejemplo, en el capítulo anterior, cuando calculábamos las raíces de una ecuación de segundo grado, vimos que en función del valor del discriminante las raíces podían ser reales o complejas. En un caso como éste, surge la necesidad de que sea el propio programa el que tome la decisión, en función del valor del discriminante, de si lo que tiene que calcular son dos raíces reales o dos raíces complejas conjugadas.

Así mismo, en más de una ocasión necesitaremos ejecutar un conjunto de sentencias un número determinado de veces, o bien hasta que se cumpla una condición impuesta por nosotros. Por ejemplo, en el capítulo anterior hemos visto cómo leer un carácter de la entrada estándar. Pero si lo que queremos es leer, no un carácter sino todos los que escribamos por el teclado hasta detectar la marca de fin de fichero, tendremos que utilizar una sentencia repetitiva.

En este capítulo aprenderá a escribir código para que un programa tome decisiones y para que sea capaz de ejecutar bloques de sentencias repetidas veces.

SENTENCIA if

La sentencia **if** permite a un programa tomar una decisión para ejecutar una acción u otra, basándose en el resultado verdadero o falso de una expresión. La sintaxis para utilizar esta sentencia es la siguiente:

```
if (condición)
    sentencia 1;
[else
    sentencia 2];
```

donde *condición* es una expresión booleana, y *sentencia 1* y *sentencia 2* representan a una sentencia simple o compuesta. Cada sentencia simple debe finalizar con un punto y coma.

Una sentencia **if** se ejecuta de la forma siguiente:

1. Se evalúa la condición.
2. Si el resultado de la evaluación de la condición es verdadero (**true**) se ejecutará lo indicado por la *sentencia 1*.
3. Si el resultado de la evaluación de la condición es falso (**false**), se ejecutará lo indicado por la *sentencia 2*, si la cláusula **else** se ha especificado.
4. Si el resultado de la evaluación de la condición es falso, y la cláusula **else** se ha omitido, la *sentencia 1* se ignora.
5. En cualquier caso, la ejecución continúa en la siguiente sentencia ejecutable que haya a continuación de la sentencia **if**.

A continuación se exponen algunos ejemplos para que vea de una forma sencilla cómo se utiliza la sentencia **if**.

```
if (x != 0)
    b = a / x;
    b = b + 1;
```

En este ejemplo, la condición viene impuesta por la expresión $x \neq 0$. Entonces $b = a / x$, que sustituye a la *sentencia 1* del formato general, se ejecutará si la expresión es verdadera (x distinta de 0) y no se ejecutará si la expresión es falsa (x igual a 0). En cualquier caso, se continúa la ejecución en la línea siguiente, $b = b + 1$. Veamos otro ejemplo:

```
if (a < b) c = c + 1;
// siguiente línea del programa
```

En este otro ejemplo, la condición viene impuesta por una expresión $a < b$. Si al evaluar la condición se cumple que a es menor que b , entonces se ejecuta la sentencia $c = c + 1$. En otro caso, esto es, si a es mayor o igual que b , se continúa en la línea siguiente, ignorándose la sentencia $c = c + 1$.

En el ejemplo siguiente, la condición viene impuesta por la expresión $a \neq 0 \& b \neq 0$. Si al evaluar la condición se cumple que a y b son distintas de cero, entonces se ejecuta la sentencia $x = i$. En otro caso, la sentencia $x = i$ se ignora, continuando la ejecución en la línea siguiente.

```
if (a != 0 && b != 0)
    x = i;
// siguiente línea del programa
```

En el ejemplo siguiente, si se cumple que a es igual a $b * 5$, se ejecutan las sentencias $x = 4$ y $a = a + x$. En otro caso, se ejecuta la sentencia $b = 0$. En ambos casos, la ejecución continúa en la siguiente línea de programa.

```
if (a == b * 5)
{
    x = 4;
    a = a + x;
}
else
    b = 0;
// siguiente línea del programa
```

Un error típico es escribir, en lugar de la condición del ejemplo anterior, la siguiente:

```
if (a = b * 5)
// ...
```

En este caso, suponiendo por ejemplo que a es de tipo **int**, el compilador mostrará un mensaje de error indicando que no puede convertir un **int** a **boolean**, porque la sentencia anterior es equivalente a escribir:

```
a = b * 5;
if (a)
// ...
```

donde se observa que a no puede dar un resultado **boolean**. Sí sería correcto lo siguiente:

```
a = b * 5;
if (a != 0)
// ...
```

que equivale a:

```
if ((a = b * 5) != 0)
// ...
```

En este otro ejemplo que se muestra a continuación, la sentencia **return** se ejecutará solamente cuando *car* sea igual al carácter '*s*'.

```
if (car == 's')
    return;
```

ANIDAMIENTO DE SENTENCIAS if

Observando el formato general de la sentencia **if** cabe una pregunta: ¿cómo *sentencia 1* o *sentencia 2* se puede escribir otra sentencia **if**? La respuesta es sí. Esto es, las sentencias **if ... else** pueden estar anidadas. Por ejemplo:

```
if (condición 1)
{
    if (condición 2)
        sentencia 1;
}
else
    sentencia 2;
```

Al evaluarse las condiciones anteriores, pueden presentarse los casos que se indican en la tabla siguiente:

condición 1	condición 2	se ejecuta: sentencia 1	sentencia 2
F	F	no	sí
F	V	no	sí
V	F	no	no
V	V	sí	no

(V = verdadero, F = falso, no = no se ejecuta, sí = sí se ejecuta)

En el ejemplo anterior las llaves definen perfectamente que la cláusula **else** está emparejada con el primer **if**. ¿Qué sucede si quitamos las llaves?

```
if (condición 1)
    if (condición 2)
        sentencia 1;
else
    sentencia 2;
```

Ahora podríamos dudar de a qué **if** pertenece la cláusula **else**. Cuando en el código de un programa aparecen sentencias **if ... else** anidadas, la regla para diferenciar cada una de estas sentencias es que “cada **else** se corresponde con el **if** más próximo que no haya sido emparejado”. Según esto la cláusula **else** está em-

parejada con el segundo **if**. Entonces, al evaluarse ahora las *condiciones 1* y *2*, pueden presentarse los casos que se indican en la tabla siguiente:

<i>condición 1</i>	<i>condición 2</i>	<i>se ejecuta: sentencia 1</i>	<i>sentencia 2</i>
F	F	no	no
F	V	no	no
V	F	no	sí
V	V	sí	no

(V = verdadero, F = falso, no = no se ejecuta, sí = sí se ejecuta)

Como ejemplo se puede observar el siguiente segmento de programa que escribe un mensaje indicando cómo es un número *a* con respecto a otro *b* (mayor, menor o igual):

```
if (a > b)
    flujoS.println(a + " es mayor que " + b);
else if (a < b)
    flujoS.println(a + " es menor que " + b);
else
    flujoS.println(a + " es igual a " + b);
// siguiente línea del programa
```

Es importante observar que una vez que se ejecuta una acción como resultado de haber evaluado las condiciones impuestas, la ejecución del programa continúa en la siguiente línea a la estructura a que dan lugar las sentencias **if ... else** anidadas. En el ejemplo anterior si se cumple que *a* es mayor que *b*, se escribe el mensaje correspondiente y se continúa en la siguiente línea del programa.

Así mismo, si en el ejemplo siguiente ocurre que *a* no es igual a 0, la ejecución continúa en la siguiente línea del programa.

```
if (a == 0)
    if (b != 0)
        s = s + b;
    else
        s = s + a;
// siguiente línea del programa
```

Si en lugar de la solución anterior, lo que deseamos es que se ejecute $s = s + a$ cuando *a* no es igual a 0, entonces tendremos que incluir entre llaves el segundo **if** sin la cláusula **else**; esto es:

```
if (a == 0)
{
```

```

if (b != 0)
    s = s + b;
}
else
    s = s + a;
// siguiente linea del programa

```

Como ejercicio sobre la teoría expuesta, vamos a realizar una aplicación que dé como resultado el menor de tres números *a*, *b* y *c*. La forma de proceder es comparar cada número con los otros dos una sola vez. La simple lectura del código que se muestra a continuación es suficiente para entender el proceso seguido.

```

// La clase Leer debe estar en alguna carpeta de las especificadas
// por la variable de entorno CLASSPATH.
//
public class CMenor
{
    // Menor de tres números a, b y c

    public static void main(String[] args)
    {
        float a, b, c, menor;

        // Leer los valores de a, b y c
        System.out.print("a : "); a = Leer.datoFloat();
        System.out.print("b : "); b = Leer.datoFloat();
        System.out.print("c : "); c = Leer.datoFloat();
        // Obtener el menor
        if (a < b)
            if (a < c)
                menor = a;
            else
                menor = c;
        else
            if (b < c)
                menor = b;
            else
                menor = c;
        System.out.println("Menor = " + menor);
    }
}

```

ESTRUCTURA else if

La estructura presentada a continuación, aparece con bastante frecuencia y es por lo que se le da un tratamiento por separado. Esta estructura es consecuencia de las sentencias **if** anidadas. Su formato general es:

```

if (condición 1)
    sentencia 1;
else if (condición 2)
    sentencia 2;
else if (condición 3)
    sentencia 3;
.
.
.
else
    sentencia n;

```

La evaluación de esta estructura sucede así: si se cumple la *condición 1*, se ejecuta la *sentencia 1* y si no se cumple se examinan secuencialmente las condiciones siguientes hasta el último **else**, ejecutándose la sentencia correspondiente al primer **else if**, cuya *condición* sea cierta. Si todas las condiciones son falsas, se ejecuta la *sentencia n* correspondiente al último **else**. En cualquier caso, se continúa en la primera sentencia ejecutable que haya a continuación de la estructura. Las *sentencias 1, 2, ..., n* pueden ser sentencias simples o compuestas.

Por ejemplo, al efectuar una compra en un cierto almacén, si adquirimos más de 100 unidades de un mismo artículo, nos hacen un descuento de un 40 %; entre 25 y 100 un 20 %; entre 10 y 24 un 10 %; y no hay descuento para una adquisición de menos de 10 unidades. Se pide calcular el importe a pagar. La solución se presentará de la siguiente forma:

Código artículo.....	111
Cantidad comprada.....	100
Precio unitario.....	100
 Descuento.....	20.0%
Total.....	8000.0

En la solución presentada como ejemplo, se puede observar que como la cantidad comprada está entre 25 y 100, el descuento aplicado es de un 20%.

La solución de este problema puede ser de la forma siguiente:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```

int ar, cc;
float pu, desc;

```

- A continuación leemos los datos *ar, cc y pu*.

```
System.out.print("Código artículo..... ");
```

```

ar = Leer.datoInt();
System.out.print("Cantidad comprada..... ");
cc = Leer.datoInt();
System.out.print("Precio unitario..... ");
pu = Leer.datoFloat();

```

- Conocidos los datos, realizamos los cálculos y escribimos el resultado.

```

if (cc > 100)
    desc = 40F;      // descuento 40%
else if (cc >= 25)
    desc = 20F;      // descuento 20%
else if (cc >= 10)
    desc = 10F;      // descuento 10%
else
    desc = 0.0F;      // descuento 0%
System.out.println("Descuento..... " + desc + "%");
System.out.println("Total..... " +
                    cc * pu * (1 - desc / 100));

```

Se puede observar que las condiciones se han establecido según los descuentos de mayor a menor. Como ejercicio, piense o pruebe que ocurriría si establece las condiciones según los descuentos de menor a mayor. La aplicación completa se muestra a continuación.

```

// La clase Leer debe estar en alguna carpeta de las especificadas
// por la variable de entorno CLASSPATH.
//
public class CDescuento
{
    public static void main(String[] args)
    {
        int ar, cc;
        float pu, desc;

        System.out.print("Código artículo..... ");
        ar = Leer.datoInt();
        System.out.print("Cantidad comprada..... ");
        cc = Leer.datoInt();
        System.out.print("Precio unitario..... ");
        pu = Leer.datoFloat();
        System.out.println();

        if (cc > 100)
            desc = 40F;      // descuento 40%
        else if (cc >= 25)
            desc = 20F;      // descuento 20%
        else if (cc >= 10)
            desc = 10F;      // descuento 10%
    }
}

```

```

        else
            desc = 0.0F;      // descuento 0%
        System.out.println("Descuento..... " + desc + "%");
        System.out.println("Total..... " +
                           cc * pu * (1 - desc / 100));
    }
}

```

SENTENCIA switch

La sentencia **switch** permite ejecutar una de varias acciones, en función del valor de una expresión. Es una sentencia especial para decisiones múltiples. La sintaxis para utilizar esta sentencia es:

```

switch (expresión)
{
    case expresión-constante 1:
        [sentencia 1;]
    [case expresión-constante 2:]
        [sentencia 2;]
    [case expresión-constante 3:]
        [sentencia 3;]

    .
    .

    [default:]
        [sentencia n;]
}

```

donde *expresión* es una expresión entera de tipo **char**, **byte**, **short** o **int** y *expresión-constante* es una constante también entera y de los mismos tipos. Tanto la *expresión* como las *expresiones constantes* son convertidas implícitamente a **int**. Por último, *sentencia* es una sentencia simple o compuesta. En el caso de tratarse de una sentencia compuesta, no hace falta incluir las sentencias simples entre {}.

La sentencia **switch** evalúa la expresión entre paréntesis y compara su valor con las constantes de cada **case**. La ejecución de las sentencias del bloque de la sentencia **switch**, comienza en el **case** cuya constante coincida con el valor de la expresión y continúa hasta el final del bloque o hasta una sentencia que transfiera el control fuera del bloque de **switch**; por ejemplo, **break**. La sentencia **switch** puede incluir cualquier número de cláusulas **case**.

Si no existe una constante igual al valor de la expresión, entonces se ejecutan las sentencias que están a continuación de **default**, si esta cláusula ha sido especificada. La cláusula **default** puede colocarse en cualquier parte del bloque y no necesariamente al final.

En una sentencia **switch** es posible hacer declaraciones en el bloque de cada **case**, igual que en cualquier otro bloque, pero no al principio del bloque **switch**, antes del primer **case**. Por ejemplo:

```
switch (m)
{
    int n = 0, k = 2; // declaración no permitida
    case 7:
        int i = 0;      // declaración permitida
        while ( i < m )
        {
            n += (k + i) * 3;
            i++;
        }
        break;
    case 13:
        // ...
        break;
    // ...
}
```

El error que se ha presentado en el ejemplo anterior puede solucionarse así:

```
int n = 0, k = 2;
switch (m)
{
    // ...
}
```

Para ilustrar la sentencia **switch**, vamos a realizar un programa que lea una fecha representada por dos enteros, *mes* y *año*, y dé como resultado los días correspondientes al *mes*. Esto es:

```
Introducir mes (##) y año (####): 5 2002
El mes 5 del año 2002 tiene 31 días
```

Hay que tener en cuenta que febrero puede tener 28 días, o bien 29 si el año es bisiesto. Un año es bisiesto cuando es múltiplo de 4 y no de 100 o cuando es múltiplo de 400. Por ejemplo, el año 2000 por las dos primeras condiciones no sería bisiesto, pero sí lo es porque es múltiplo de 400; el año 2100 no es bisiesto porque aunque sea múltiplo de 4, también lo es de 100 y no es múltiplo de 400.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int días = 0, mes = 0, año = 0;
```

- A continuación leemos los datos *mes* y *año*.

```
System.out.print("Mes (##): "); mes = Leer.datoInt();
System.out.print("Año (####): "); año = Leer.datoInt();
```

- Despues comparamos el *mes* con las constantes 1, 2, ..., 12. Si *mes* es 1, 3, 5, 7, 8, 10 ó 12 asignamos a *días* el valor 31. Si *mes* es 4, 6, 9 u 11 asignamos a *días* el valor 30. Si *mes* es 2, verificaremos si el *año* es bisiesto, en cuyo caso asignamos a *días* el valor 29 y si no es bisiesto, asignamos a *días* el valor 28. Si *mes* no es ningun valor de los anteriores enviaremos un mensaje al usuario indicándole que el mes no es válido. Todo este proceso lo realizaremos con una sentencia **switch**.

```
switch (mes)
{
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        días = 31;
        break;
    case 4: case 6: case 9: case 11:
        días = 30;
        break;
    case 2:
        // ¿Es el año bisiesto?
        if ((año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0))
            días = 29;
        else
            días = 28;
        break;
    default:
        System.out.println("\nEl mes no es válido");
        break;
}
```

Cuando una constante coincide con el valor de *mes*, se ejecutan las sentencias especificadas a continuación de la misma, siguiendo la ejecución del programa por los bloques de las siguientes cláusulas **case**, a no ser que se tome una acción explícita para abandonar el bloque de la sentencia **switch**. Ésta es precisamente la función de la sentencia **break** al final de cada bloque **case**.

- Por ultimo si el *mes* es válido, escribimos el resultado solicitado.

```
if (mes >= 1 && mes <= 12)
    System.out.println("\nEl mes " + mes + " del año " + año +
        " tiene " + días + " días");
```

El programa completo se muestra a continuación:

```

// La clase Leer debe estar en alguna carpeta de las especificadas
// por la variable de entorno CLASSPATH.
//
public class CDiasMes
{
    // Días correspondientes a un mes de un año dado

    public static void main(String[] args)
    {
        int días = 0, mes = 0, año = 0;

        System.out.print("Mes (##): "); mes = Leer.datoInt();
        System.out.print("Año (####): "); año = Leer.datoInt();

        switch (mes)
        {
            case 1:      // enero
            case 3:      // marzo
            case 5:      // mayo
            case 7:      // julio
            case 8:      // agosto
            case 10:     // octubre
            case 12:     // diciembre
                días = 31;
                break;
            case 4:      // abril
            case 6:      // junio
            case 9:      // septiembre
            case 11:     // noviembre
                días = 30;
                break;
            case 2:      // febrero
                // ¿Es el año bisiesto?
                if ((año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0))
                    días = 29;
                else
                    días = 28;
                break;
            default:
                System.out.println("\nEl mes no es válido");
                break;
        }
        if (mes >= 1 && mes <= 12)
            System.out.println("\nEl mes " + mes + " del año " + año +
                               " tiene " + días + " días");
    }
}

```

El que las cláusulas **case** estén una a continuación de otra o una debajo de otra no es más que una cuestión de estilo, ya que Java interpreta cada carácter

nueva línea como un espacio en blanco; esto es, el código al que llega el compilador es el mismo en cualquier caso.

La sentencia **break** que se ha puesto a continuación de la cláusula **default** no es necesaria; simplemente obedece a un buen estilo de programación. Así, cuando tengamos que añadir otro caso ya tenemos puesto **break**, con lo que hemos eliminado una posible fuente de errores.

SENTENCIA while

La sentencia **while** ejecuta una sentencia, simple o compuesta, cero o más veces, dependiendo del valor de una expresión booleana. Su sintaxis es:

```
while (condición)
    sentencia;
```

donde *condición* es cualquier expresión booleana y *sentencia* es una sentencia simple o compuesta.

La ejecución de la sentencia **while** sucede así:

1. Se evalúa la *condición*.
2. Si el resultado de la evaluación es **false** (falso), la sentencia no se ejecuta y se pasa el control a la siguiente sentencia en el programa.
3. Si el resultado de la evaluación es **true** (verdadero), se ejecuta la sentencia y el proceso descrito se repite desde el punto 1.

Por ejemplo, el siguiente código, que podrá ser incluido en cualquier aplicación, solicita obligatoriamente una de las dos respuestas posibles: *s/n* (sí o no).

```
char car = '\0';
try
{
    System.out.print("\nDesea continuar s/n (sí o no) ");
    car = (char)System.in.read();
    // Saltar los caracteres disponibles en el flujo de entrada
    System.in.skip(System.in.available());
    while (car != 's' && car != 'n')
    {
        System.out.print("\nDesea continuar s/n (sí o no) ");
        car = (char)System.in.read();
        System.in.skip(System.in.available());
    }
}
catch (IOException ignorada) {}
```

Observe que antes de ejecutarse la sentencia **while** se visualiza el mensaje “Desea continuar s/n (sí o no)” y se inicia la condición; esto es, se asigna un carácter a la variable *car* que interviene en la condición de la sentencia **while**.

La sentencia **while** se interpreta de la forma siguiente: mientras el valor de *car* no sea igual ni al carácter ‘s’ ni al carácter ‘n’, visualizar el mensaje “Desea continuar s/n (sí o no)” y leer otro carácter. Esto obliga al usuario a escribir el carácter ‘s’ o ‘n’ en minúsculas.

El ejemplo expuesto, puede escribirse de forma más simplificada así:

```
char car = '\0';
try
{
    System.out.print("\nDesea continuar s/n (sí o no) ");
    while ((car = (char)System.in.read()) != 's' && car != 'n')
    {
        // Saltar los caracteres disponibles en el flujo de entrada
        System.in.skip(System.in.available());
        System.out.print("\nDesea continuar s/n (sí o no) ");
    }
}
catch(IOException ignorada) {}
```

La diferencia de este ejemplo con respecto al anterior es que ahora la *condición* incluye la lectura de la variable *car*, que se ejecuta primero por estar entre paréntesis. A continuación se compara *car* con los caracteres ‘s’ y ‘n’.

El siguiente ejemplo, que visualiza el código ASCII de cada uno de los caracteres de una cadena de texto introducida por el teclado, da lugar a un bucle infinito, porque la condición es siempre cierta (valor **true**). Para salir del bucle infinito tiene que pulsar las teclas *Ctrl+C*.

```
import java.io.*;
public class CAscii
{
    // Código ASCII de cada uno de los caracteres de un texto
    public static void main(String[] args)
    {
        char car = 0; // car = carácter nulo (\0)

        try
        {
            System.out.print("Introduzca una cadena de texto: ");
            while (true) // condición siempre cierta
            {
```

```

        car = (char)System.in.read(); // leer el siguiente carácter
        if (car != '\r' && car != '\n')
            System.out.println("El código ASCII de " + car +
                " es " + (int)car);
        // Si no hay datos disponibles, solicitarlos
        if (System.in.available() == 0)
            System.out.print("Introduzca texto: ");
    }
}
catch(IOException ignorada) {}
}

```

A continuación ejecutamos la aplicación. Introducimos, por ejemplo, el carácter 'a' y observamos los siguientes resultados:

Introduzca una cadena de texto: a[Entrar]
El código ASCII de a es 97
Introduzca una cadena de texto:

Este resultado demuestra que cuando escribimos ‘a’ y pulsamos la tecla *Entrar* para validar la entrada, sólo se visualiza el código ASCII de ese carácter; los caracteres *\r* y *\n* introducidos al pulsar *Entrar* son ignorados porque así se ha programado. Cuando se han leído todos los caracteres del flujo de entrada, se solicitan nuevos datos. Lógicamente, habrá comprendido que aunque se lea carácter a carácter se puede escribir, hasta pulsar *Entrar*, un texto cualquiera. Por ejemplo:

```
Introduzca una cadena de texto: hola[Entrar]
El código ASCII de h es 104
El código ASCII de o es 111
El código ASCII de l es 108
El código ASCII de a es 97
Introduzca una cadena de texto:
```

El resultado obtenido permite observar que el bucle **while** se está ejecutando sin pausa mientras hay caracteres en el flujo de entrada. Cuando dicho flujo queda vacío y se ejecuta el método **read** de nuevo, la ejecución se detiene a la espera de nuevos datos.

Modifiquemos ahora el ejemplo anterior con el objetivo de eliminar el bucle infinito. Esto se puede hacer incluyendo en el **while** una condición de terminación; por ejemplo, leer datos hasta alcanzar la marca de fin de fichero. Recuerde que para el flujo estándar de entrada, esta marca se produce cuando se pulsan las teclas *Ctrl+D* en UNIX, o bien *Ctrl+Z* en aplicaciones Windows de consola, y que cuando **read** lee una marca de fin de fichero, devuelve el valor -1.

```

import java.io.*;

public class CAscii
{
    // Código ASCII de cada uno de los caracteres de un texto
    public static void main(String[] args)
    {
        final char eof = (char)-1;
        char car = 0; // car = carácter nulo (\0)
        try
        {
            System.out.println("Introduzca una cadena de texto.");
            System.out.println("Para terminar pulse Ctrl+z\n");
            while ((car = (char)System.in.read()) != eof)
            {
                if (car != '\r' && car != '\n')
                    System.out.println("El código ASCII de " + car +
                        " es " + (int)car);
            }
        }
        catch(IOException ignorada) {}
    }
}

```

Una solución posible de esta aplicación es la siguiente:

Introduzca una cadena de texto.
Para terminar pulse Ctrl+z

```

hola[Entrar]
El código ASCII de h es 104
El código ASCII de o es 111
El código ASCII de l es 108
El código ASCII de a es 97
adiós[Entrar]
El código ASCII de a es 97
El código ASCII de d es 100
El código ASCII de i es 105
El código ASCII de ó es 162
El código ASCII de s es 115
[Ctrl] [z]

```

Bucles anidados

Cuando se incluye una sentencia **while** dentro de otra sentencia **while**, en general una sentencia **while**, **do**, o **for** dentro de otra de ellas, estamos en el caso de bucles anidados. Por ejemplo:

```

public static void main(String[] args)
{
    int i = 1, j = 1;
    while ( i <= 3 ) // mientras i sea menor o igual que 3
    {
        System.out.print("Para i = " + i + ": ");
        while ( j <= 4 ) // mientras j sea menor o igual que 4
        {
            System.out.print("j = " + j + ", ");
            j++; // aumentar j en una unidad
        }
        System.out.println(); // avanzar a una nueva línea
        i++; // aumentar i en una unidad
        j = 1; // iniciar j de nuevo a 1
    }
}

```

Al ejecutar este método se obtiene el siguiente resultado:

```

Para i = 1: j = 1, j = 2, j = 3, j = 4,
Para i = 2: j = 1, j = 2, j = 3, j = 4,
Para i = 3: j = 1, j = 2, j = 3, j = 4,

```

Este resultado demuestra que el bucle exterior se ejecute tres veces, y por cada una de éstas, el bucle interior se ejecuta a su vez cuatro veces. Es así como se ejecutan los bucles anidados: por cada iteración del bucle externo, el interno se ejecuta hasta finalizar todas sus iteraciones.

Observe también que cada vez que finaliza la ejecución de la sentencia **while** interior, avanzamos a una nueva línea, incrementamos el valor de *i* en una unidad e iniciamos de nuevo *j* al valor 1.

Como aplicación de lo expuesto, vamos a realizar un programa que imprima los números *z*, comprendidos entre 1 y 50, que cumplan la expresión:

$$z^2 = x^2 + y^2$$

donde *z*, *x* e *y* son números enteros positivos. El resultado se presentará de la forma siguiente:

<i>Z</i>	<i>X</i>	<i>Y</i>
5	3	4
13	5	12
10	6	8
...
50	30	40

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int x = 1, y = 1, z = 0;
```

- A continuación escribimos la cabecera de la solución.

```
System.out.println("Z\t" + "X\t" + "Y");
System.out.println("_____");
```

- Después, para $x = 1$, e $y = 1, 2, 3, \dots$, para $x = 2$, e $y = 2, 3, 4, \dots$, para $x = 3$, e $y = 3, 4, \dots$, hasta $x = 50$, calculamos la $\sqrt{x^2 + y^2}$; llamamos a este valor z (observe que y es igual o mayor que x para evitar que se repitan pares de valores como $x=3, y=4$ y $x=4, y=3$). Si z es exacto, escribimos z, x e y . Esto es, para los valores descritos de x e y , hacemos los cálculos:

```
z = (int)Math.sqrt(x * x + y * y); // z es una variable entera
if (z * z == x * x + y * y)      // ¿la raíz cuadrada fue exacta?
System.out.println(z + "\t" + x + "\t" + y);
```

Además, siempre que obtengamos un valor z mayor que 50 lo desecharemos y continuaremos con un nuevo valor de x y los correspondientes valores de y .

El programa completo se muestra a continuación:

```
public class CPitagoras
{
    // Teorema de Pitágoras
    public static void main(String[] args)
    {
        int x = 1, y = 1, z = 0;
        System.out.println("Z\t" + "X\t" + "Y");
        System.out.println("_____");

        while (x <= 50)
        {
            // Calcular z. Como z es un entero, almacena
            // la parte entera de la raíz cuadrada
            z = (int)Math.sqrt(x * x + y * y);
            while (y <= 50 && z <= 50)
            {
                // Si la raíz cuadrada anterior fue exacta,
                // escribir z, x e y
                if (z * z == x * x + y * y)
                    System.out.println(z + "\t" + x + "\t" + y);
            }
        }
    }
}
```

```
    y = y + 1;
    z = (int)Math.sqrt(x * x + y * y);
}
x = x + 1; y = x;
```

SENTENCIA do ... while

La sentencia **do ... while** ejecuta una sentencia, simple o compuesta, una o más veces dependiendo del valor de una expresión. Su sintaxis es la siguiente:

do *sentencia;*
while (condición);

donde *condición* es cualquier expresión booleana y *sentencia* es una sentencia simple o compuesta. Observe que la estructura **do ... while** finaliza con un punto y coma.

La ejecución de una sentencia **do ... while** sucede de la siguiente forma:

1. Se ejecuta el bloque (sentencia simple o compuesta) de **do**.
 2. Se evalúa la expresión correspondiente a la *condición* de finalización del bucle.
 3. Si el resultado de la evaluación es **false** (falso), se pasa el control a la siguiente sentencia en el programa.
 4. Si el resultado de la evaluación es **true** (verdadero), el proceso descrito se repite desde el punto 1.

Por ejemplo, el siguiente código obliga al usuario a introducir un valor positivo:

```
double n;
do // ejecutar las sentencias siguientes
{
    System.out.print("Número: ");
    n = Leer.datoDouble();
}
while ( n < 0 ); // mientras n sea menor que 0
```

Cuando se utiliza una estructura **do ... while** el bloque de sentencias se ejecuta al menos una vez, porque la condición se evalúa al final. En cambio, cuando se

ejecuta una estructura **while** puede suceder que el bloque de sentencias no se ejecute, lo que ocurrirá siempre que la condición sea inicialmente falsa.

Como ejercicio, vamos a realizar un programa que calcule la raíz cuadrada de un número n por el método de Newton. Este método se enuncia así: sea r_i la raíz cuadrada aproximada de n . La siguiente raíz aproximada r_{i+1} se calcula en función de la anterior así:

$$r_{i+1} = \frac{\frac{n}{r_i} + r_i}{2}$$

El proceso descrito se repite hasta que la diferencia en valor absoluto de las dos últimas aproximaciones calculadas, sea tan pequeña como nosotros queramos (teniendo en cuenta los límites establecidos por tipo de datos utilizado). Según esto, la última aproximación será una raíz válida, cuando se cumpla que:

$$\text{abs}(r_i - r_{i+1}) \leq \varepsilon$$

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double n;           // número
double aprox;      // aproximación a la raíz cuadrada
double antaprox;   // anterior aproximación a la raíz cuadrada
double epsilon;    // coeficiente de error
```

- A continuación leemos los datos n , $aprox$ y $epsilon$.

```
System.out.print("Número: ");
n = Leer.datoDouble();
System.out.print("Raíz cuadrada aproximada: ");
aprox = Leer.datoDouble();
System.out.print("Coeficiente de error: ");
epsilon = Leer.datoDouble();
```

- Despues, se aplica la fórmula de Newton.

```
do
{
    antaprox = aprox;
    aprox = (n/antaprox + antaprox) / 2;
}
while (Math.abs(aprox - antaprox) >= epsilon);
```

Al aplicar la fórmula por primera vez, la variable *antaprox* contiene el valor aproximado a la raíz cuadrada que hemos introducido a través del teclado. Para sucesivas veces, *antaprox* contendrá la última aproximación calculada.

- Cuando la condición especificada en la estructura **do ... while** mostrada anteriormente sea falsa, el proceso habrá terminado. Sólo queda imprimir el resultado.

```
System.out.println("La raíz cuadrada de " + n + " es " + aprox);
```

El programa completo se muestra a continuación. Para no permitir la entrada de números negativos, se ha utilizado una estructura **do ... while** que preguntará por el valor solicitado mientras el introducido sea negativo.

```
// Leer.class debe estar en la carpeta especificada por CLASSPATH
public class CRaizCuadrada
{
    // Raíz cuadrada. Método de Newton.

    public static void main(String[] args)
    {
        double n;          // número
        double aprox;      // aproximación a la raíz cuadrada
        double antaprox;  // anterior aproximación a la raíz cuadrada
        double epsilon;    // coeficiente de error

        do
        {
            System.out.print("Número: ");
            n = Leer.datoDouble();
        }
        while ( n <= 0 );

        do
        {
            System.out.print("Raíz cuadrada aproximada: ");
            aprox = Leer.datoDouble();
        }
        while ( aprox <= 0 );

        do
        {
            System.out.print("Coeficiente de error: ");
            epsilon = Leer.datoDouble();
        }
        while ( epsilon <= 0 );
    }
}
```

```

do
    antaprox = aprox;
    aprox = (n/antaprox + antaprox) / 2;
}
while (Math.abs(aprox - antaprox) >= epsilon);
System.out.println("La raíz cuadrada de " + n + " es " + aprox);
}
}

```

Si ejecuta este programa para un valor de *n* igual a 10, obtendrá la siguiente solución:

```

Número: 10
Raíz cuadrada aproximada: 1
Coeficiente de error: 1e-4
La raíz cuadrada de 10.0 es 3.16

```

SENTENCIA for

La sentencia **for** permite ejecutar una sentencia simple o compuesta, repetidamente un número de veces conocido. Su sintaxis es la siguiente:

```

for ([v1=e1 [, v2=e2]...];[condición];[progresión-condición])
    sentencia;

```

- *v1, v2, ...*, representan variables de control que serán iniciadas con los valores de las expresiones *e1, e2, ...*;
- *condición* es una expresión booleana que si se omite, se supone verdadera;
- *progresión-condición* es una o más expresiones separadas por comas cuyos valores evolucionan en el sentido de que se cumpla la condición para finalizar la ejecución de la sentencia **for**;
- *sentencia* es una sentencia simple o compuesta.

La ejecución de la sentencia **for** sucede de la siguiente forma:

1. Se inician las variables *v1, v2, ...*
2. Se evalúa la condición:
 - a) Si el resultado es **true** (verdadero), se ejecuta el bloque de sentencias, se evalúa la expresión que da lugar a la progresión de la condición y se vuelve al punto 2.
 - b) Si el resultado es **false** (falso), la ejecución de la sentencia **for** se da por finalizada y se pasa el control a la siguiente sentencia del programa.

Por ejemplo, la siguiente sentencia **for** imprime los números del 1 al 100. Literalmente dice: desde *i* igual a 1, mientras *i* sea menor o igual que 100, incrementado la *i* de uno en uno, escribir el valor de *i*.

```
int i;
for (i = 1; i <= 100; i++)
    System.out.print(i + " ");
```

El siguiente ejemplo imprime los múltiplos de 7 que hay entre 7 y 112. Se puede observar que, en este caso, la variable se ha declarado e iniciado en la propia sentencia **for** (esto no se puede hacer en una sentencia **while**; las variables que intervienen en la condición de una sentencia **while** deben haber sido declaradas e iniciadas antes de que se procese la condición por primera vez).

```
for (int k = 7; k <= 112; k += 7)
    System.out.print(k + " ");
```

En el siguiente ejemplo se puede observar la utilización de la coma como separador de las variables de control y de las expresiones que hacen que evolucionen los valores que intervienen en la condición de finalización.

```
int f, c;
for (f = 3, c = 6; f + c < 40; f++, c += 2)
    System.out.println("f = " + f + "\tc = " + c);
```

Este otro ejemplo que ve a continuación, imprime los valores desde 1 hasta 10 con incrementos de 0.5.

```
for (float i = 1; i <= 10; i += 0.5)
    System.out.print(i + " ");
```

El siguiente ejemplo imprime las letras del abecedario en orden inverso.

```
char car;
for (car = 'z'; car >= 'a'; car--)
    System.out.print(car + " ");
```

El ejemplo siguiente indica cómo realizar un bucle infinito. Para salir de un bucle infinito tiene que pulsar las teclas *Ctrl+C*.

```
for (;;)
{
    sentencias;
```

sentencias; contiene una serie de comandos que se ejecutan de forma continua. La ejecución de este bucle se detiene solo cuando se pulsa la combinación de teclas *Ctrl+C*.

Como aplicación de la sentencia **for** vamos a imprimir un tablero de ajedrez en el que las casillas blancas se simbolizarán con una B y las negras con una N. Así mismo, el programa deberá marcar con * las casillas a las que se puede mover un alfil desde una posición dada. La solución será similar a la siguiente:

Posición del alfil:

```
fila 3
columna 4
```

```
B * B N B * B N
N B * B * B N B
B N B * B N B N
N B * B * B N B
B * B N B * B N
* B N B N B * B
B N B N B N B *
N B N B N B N B
```

Desarrollo del programa:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int falfil, calfil; // posición inicial del alfil
int fila, columna; // posición actual del alfil
```

- Leer la fila y la columna en la que se coloca el alfil.

```
System.out.print(" fila "); falfil = Leer.datToInt();
System.out.print(" columna "); calfil = Leer.datToInt();
```

- Partiendo de la fila 1, columna 1 y recorriendo el tablero por filas,

```
for (fila = 1; fila <= 8; fila++)
{
    for (columna = 1; columna <= 8; columna++)
    {
        // Pintar el tablero de ajedrez
    }
    System.out.println(); // cambiar de fila
}
```

imprimir un *, una B o una N dependiendo de las condiciones especificadas a continuación:

- ◊ Imprimir un * si se cumple, que la suma o diferencia de la fila y columna actuales, coincide con la suma o diferencia de la fila y columna donde se coloca el alfil.

- ◊ Imprimir una B si se cumple que la fila más columna actuales es par.
- ◊ Imprimir una N si se cumple que la fila más columna actuales es impar.

```
// Pintar el tablero de ajedrez
if ((fila + columna == falfil + calfil) ||
    (fila - columna == falfil - calfil))
    System.out.print("* ");
else if ((fila + columna) % 2 == 0)
    System.out.print("B ");
else
    System.out.print("N ");
```

El programa completo se muestra a continuación.

```
// Leer.class debe estar en la carpeta especificada por CLASSPATH
//
public class CAjedrez
{
    // Imprimir un tablero de ajedrez.
    public static void main(String[] args)
    {
        int falfil, calfil; // posición inicial del alfil
        int fila, columna; // posición actual del alfil

        System.out.println("Posición del alfil:");
        System.out.print("    fila      "); falfil = Leer.datToInt();
        System.out.print("    columna   "); calfil = Leer.datToInt();
        System.out.println(); // dejar una línea en blanco

        // Pintar el tablero de ajedrez
        for (fila = 1; fila <= 8; fila++)
        {
            for (columna = 1; columna <= 8; columna++)
            {
                if ((fila + columna == falfil + calfil) ||
                    (fila - columna == falfil - calfil))
                    System.out.print("* ");
                else if ((fila + columna) % 2 == 0)
                    System.out.print("B ");
                else
                    System.out.print("N ");
            }
            System.out.println(); // cambiar de fila
        }
    }
}
```

SENTENCIA break

Anteriormente vimos que la sentencia **break** finaliza la ejecución de una sentencia **switch**. Pues bien, cuando se utiliza **break** en el bloque correspondiente a una sentencia **while**, **do**, o **for**, hace lo mismo: finaliza la ejecución del bucle.

Cuando las sentencias **switch**, **while**, **do**, o **for** estén anidadas, la sentencia **break** solamente finaliza la ejecución del bucle donde esté incluida.

Por ejemplo, el bucle interno de la aplicación *CPitagoras* desarrollada anteriormente, podría escribirse también así:

```
while (y <= 50)
{
    // Si la raíz cuadrada anterior fue exacta,
    // escribir z, x e y
    if (z * z == x * x + y * y)
        System.out.println(z + "\t" + x + "\t" + y);
    y = y + 1;
    z = (int) Math.sqrt(x * x + y * y);
    if (z > 50) break; // salir del bucle
}
```

SENTENCIA continue

La sentencia **continue** obliga a ejecutar la siguiente iteración del bucle **while**, **do**, o **for**, en el que está contenida. Su sintaxis es:

`continue;`

Como ejemplo, vea la siguiente aplicación que imprime todos los números entre 1 y 100 que son múltiplos de 5.

```
public class Test
{
    public static void main(String[] args)
    {
        for (int n = 0; n <= 100; n++)
        {
            // Si n no es múltiplo de 5, siguiente iteración
            if (n % 5 != 0) continue;
            // Imprime el siguiente múltiplo de 5
            System.out.println(n + " ");
        }
    }
}
```

Ejecute este programa y observe que cada vez que se ejecuta la sentencia **continue**, se inicia la ejecución del bloque de sentencias de **for** para un nuevo valor de *n*.

ETIQUETAS

Con las sentencias **break** y **continue** se puede también utilizar una etiqueta para indicar dónde se debe reanudar la ejecución (quiero advertir que el uso de etiquetas es una mala práctica en programación, por lo que debe reducirse a casos excepcionales). Según lo explicado anteriormente, cuando se utiliza **break** en bucles anidados, permite finalizar la ejecución del bucle donde está incluida, continuando la ejecución en el bucle exterior más cercano; y **continue**, inicia una nueva iteración del bucle donde está incluida. Pues bien, utilizando una etiqueta con **break** o con **continue** se puede reanudar la ejecución en un bucle más externo. La etiqueta, finalizada con dos puntos, debe escribirse justo antes de la sentencia **while**, **do**, o **for**. Por ejemplo:

```

salir:
for (x = 1; x <= 5; x++)
{
    for (y = 1; y <= 5; y++)
    {
        for (z = 1; z <= 5; z++)
        {
            if ((x * y + z) % 11 == 0)
            {
                System.out.println(x + "*" + y + "+" + z +
                                     " es múltiplo de 11");
                break salir;
            }
        }
    }
}
System.out.println("Continúa la ejecución");

```

Si ejecuta una aplicación que contenga el código anterior, obtendrá el siguiente resultado:

```

2*3+5 es múltiplo de 11
Continúa la ejecución

```

La solución visualizada demuestra que cuando la condición $(x*y+z)\%11 == 0$ se cumple, **break** interrumpe la ejecución de los tres bucles, continuando la ejecución en la sentencia siguiente al bucle más externo.

Si en el código anterior se sustituye **break** por **continue**, la solución sería esta otra:

```
2*3+5 es múltiplo de 11
3*2+5 es múltiplo de 11
4*2+3 es múltiplo de 11
5*2+1 es múltiplo de 11
Continúa la ejecución
```

Los resultados mostrados indican que ahora, cada vez que se cumple la condición, **continue** hace que se reanude la ejecución para la siguiente iteración del bucle más externo (para el siguiente valor de *x*).

SENTENCIAS **try ... catch**

En el capítulo anterior expusimos que cuando durante la ejecución de un programa ocurre un error que impide su continuación, Java lanza una excepción que hace que se visualice un mensaje acerca de lo ocurrido y se detenga la ejecución. Cuando esto ocurra, si no deseamos que la ejecución del programa se detenga, habrá que utilizar **try** para poner en alerta a la aplicación acerca del código que puede lanzar una excepción y utilizar **catch** para capturar y manejar cada excepción que se lance. Por ejemplo, si ejecuta la aplicación *Test* que se muestra un poco más adelante, lanzará la excepción del tipo **ArithmaticException** que se indica a continuación:

```
Exception in thread "main" java.lang.ArithmaticException: / by zero
at Test.main(Test.java:9)
```

La información dada por el mensaje anterior, además del tipo de excepción, especifica que ha ocurrido una división por cero en la línea 9 del método **main** de la clase *Test*.

```
public class Test
{
    public static void main(String[] args)
    {
        int dato1 = 0, dato2 = 0, dato3;

        System.out.println("Se inicia la aplicación");
        dato1++;
        dato3 = dato1 / dato2;
        dato2++;
        // Otras sentencias
        System.out.println(dato1 + " " + dato2 + " " + dato3);
    }
}
```

Modifiquemos la aplicación con la intención de capturar la excepción lanzada. El resultado puede ser el siguiente:

```
public class Test
{
    public static void main(String[] args)
    {
        int dato1 = 0, dato2 = 0, dato3 = 0;

        System.out.println("Se inicia la aplicación");
        try
        {
            dato1++;
            dato3 = dato1 / dato2;
            dato2++;
            // Otras sentencias
        }
        catch(ArithmeticException e)
        {
            // Manejar una excepción de tipo ArithmeticException
            System.out.println("Error: " + e.getMessage());
            dato3 = dato1;
        }
        System.out.println(dato1 + " " + dato2 + " " + dato3);
    }
}
```

Ahora, si la sentencia `dato3 = dato1 / dato2` da lugar a una división por cero, Java detendrá temporalmente la ejecución de la aplicación y lanzará una excepción de tipo **ArithmeticException** que será capturada por la sentencia **catch**. La ejecución de la aplicación se reanudará a partir de la primera sentencia perteneciente al bloque **catch** y continuará hasta el final de la aplicación. Se puede observar que la opción que se ha tomado ante la excepción lanzada ha sido suponer una división entre 1; esto es: `dato3 = dato1`. El resultado cuando finalice la aplicación será:

```
Se inicia la aplicación
Error: / by zero
1 0 1
```

EJERCICIOS RESUELTOS

- Realizar un programa que calcule las raíces de la ecuación:

$$ax^2 + bx + c = 0$$

teniendo en cuenta los siguientes casos:

1. Si a es igual a 0 y b es igual a 0, imprimiremos un mensaje diciendo que la ecuación es degenerada.
2. Si a es igual a 0 y b no es igual a 0, existe una raíz única con valor $-c/b$.
3. En los demás casos, utilizaremos la fórmula siguiente:

$$x_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

La expresión $d = b^2 - 4ac$ se denomina discriminante.

- Si d es mayor o igual que 0 entonces hay dos raíces reales.
- Si d es menor que 0 entonces hay dos raíces complejas de la forma:

$$x + yj, \quad x - yj$$

Indicar con literales apropiados los datos a introducir, así como los resultados obtenidos.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double a, b, c; // coeficientes de la ecuación
double d; // discriminante
double re, im; // parte real e imaginaria de la raíz
```

- A continuación leemos los datos a , b y c .

```
System.out.print("a = "); a = Leer.datoDouble();
System.out.print("b = "); b = Leer.datoDouble();
System.out.print("c = "); c = Leer.datoDouble();
```

- Leídos los coeficientes, pasamos a calcular las raíces.

```
if (a == 0 && b == 0)
    System.out.println("La ecuación es degenerada");
else if (a == 0)
    System.out.println("La única raíz es: " + -c/b);
else
    // Evaluar la fórmula. Cálculo de d, re e im
```

```

if (d >= 0)
{
    // Imprimir las raíces reales
}
else
{
    // Imprimir las raíces complejas conjugadas
}
}

```

- Cálculo de $\frac{-b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}$
-

```

re = -b / (2 * a);
d = b * b - 4 * a * c;
im = Math.sqrt(Math.abs(d)) / (2 * a);

```

- Imprimir las raíces reales.

```

System.out.println("Raíces reales:");
System.out.println((re+im) + ", " + (re-im));

```

- Imprimir las raíces complejas conjugadas.

```

System.out.println("Raíces complejas:");
System.out.println(re + " + " + Math.abs(im) + " j");
System.out.println(re + " - " + Math.abs(im) + " j");

```

El programa completo se muestra a continuación.

```

// Leer.class debe estar en la carpeta especificada por CLASSPATH
//
public class CEcuacion2Grado
{
    // Calcular las raíces de una ecuación de 2º grado
    public static void main(String[] args)
    {
        double a, b, c; // coeficientes de la ecuación
        double d; // discriminante
        double re, im; // parte real e imaginaria de la raíz

        System.out.println("Coeficientes a, b y c de la ecuación:");
        System.out.print("a = "); a = Leer.datoDouble();
        System.out.print("b = "); b = Leer.datoDouble();
        System.out.print("c = "); c = Leer.datoDouble();
    }
}

```

```

        System.out.println();

        if (a == 0 && b == 0)
            System.out.println("La ecuación es degenerada");
        else if (a == 0)
            System.out.println("La única raíz es: " + -c/b);
        else
        {
            re = -b / (2 * a);
            d = b * b - 4 * a * c;
            im = Math.sqrt(Math.abs(d)) / (2 * a);
            if (d >= 0)
            {
                System.out.println("Raíces reales:");
                System.out.println((re+im) + ", " + (re-im));
            }
            else
            {
                System.out.println("Raíces complejas:");
                System.out.println(re + " + " + Math.abs(im) + " j");
                System.out.println(re + " - " + Math.abs(im) + " j");
            }
        }
    }
}

```

2. Escribir un programa para que lea un texto y dé como resultado el número de palabras con al menos cuatro vocales diferentes. Suponemos que una palabra está separada de otra por uno o más espacios (' '), tabuladores (\t) o caracteres '\n'. La entrada de datos finalizará cuando se detecte la marca de fin de fichero. La ejecución será de la forma siguiente:

Introducir texto. Para finalizar pulsar Ctrl+z.

En la Universidad hay muchos

estudiantes de Telecomunicación

[Ctrl] [z]

Número de palabras con 4 vocales distintas: 3

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en el programa.

```

int np = 0; // número de palabras con 4 vocales distintas
int a = 0, e = 0, i = 0, o = 0, u = 0;
char car;
final char eof = (char)-1;

```

- A continuación leemos el texto carácter a carácter.

```

System.out.println("Introducir texto. " +
                    "Para finalizar pulsar Ctrl+z.\n");
while ((car = (char)System.in.read()) != eof)
{
    /*
        Si el carácter leído es una 'a' hacer a = 1
        Si el carácter leído es una 'e' hacer e = 1
        Si el carácter leído es una 'i' hacer i = 1
        Si el carácter leído es una 'o' hacer o = 1
        Si el carácter leído es una 'u' hacer u = 1
        Si el carácter leído es un espacio en blanco,
        un \t o un \n, acabamos de leer una palabra. Entonces,
        si a+e+i+o+u >= 4, incrementar el contador de palabras
        de cuatro vocales diferentes y poner a, e, i, o y u de
        nuevo a cero.
    */
}
// fin del while

```

- Si la marca de fin de fichero está justamente a continuación de la última palabra (no se pulsó Entrar después de la última palabra), entonces se sale del bucle **while** sin verificar si esta palabra tenía o no cuatro vocales diferentes. Por eso este proceso hay que repetirlo fuera del **while**.

```
if ((a + e + i + o + u) >= 4) np++;
```

- Finalmente, escribimos el resultado.

```
System.out.println("\n\nNúmero de palabras con " +
                    "4 vocales distintas: " + np);
```

El programa completo se muestra a continuación.

```

import java.io.*;
// Leer.class debe estar en la carpeta especificada por CLASSPATH
//
public class CPalabras
{
    // Contar el número de palabras en un texto
    // con 4 o más vocales diferentes
    public static void main(String[] args)
    {
        int np = 0; // número de palabras con 4 vocales distintas
        int a = 0, e = 0, i = 0, o = 0, u = 0;
        char car;
        final char eof = (char)-1;
    }
}

```

```

try
{
    System.out.println("Introducir texto. " +
                       "Para finalizar pulsar Ctrl+z.\n");
    while ((car = (char)System.in.read()) != eof)
    {
        switch (car)
        {
            case 'A': case 'a': case 'á':
                a = 1;
                break;
            case 'E': case 'e': case 'é':
                e = 1;
                break;
            case 'I': case 'i': case 'í':
                i = 1;
                break;
            case 'O': case 'o': case 'ó':
                o = 1;
                break;
            case 'U': case 'u': case 'ú':
                u = 1;
                break;
            default:
                if (car == ' ')
                {
                    if ((a + e + i + o + u) >= 4) np++;
                    a = e = i = o = u = 0;
                }
                if (car == '\n')
                {
                    if ((a + e + i + o + u) >= 4) np++;
                    a = e = i = o = u = 0;
                }
        } // fin del switch
    } // fin del while
    if ((a + e + i + o + u) >= 4) np++;
    System.out.println("\n\nNúmero de palabras con " +
                      "4 vocales distintas: " + np);
}
catch(IOException ignorada) {}
}

```

3. Escribir un programa para que lea un texto y dé como resultado el número de caracteres, palabras y líneas del mismo. Suponemos que una palabra está separada de otra por uno o más espacios (' '), caracteres *tab* (\t) o caracteres '\n'. La ejecución será de la forma siguiente:

Introducir texto. Pulse [Entrar] después de cada línea.
 Para finalizar pulsar Ctrl+z.

Este programa cuenta los caracteres, las palabras y las líneas de un documento.

[Ctrl] [z]
 80 13 2

El programa completo se muestra a continuación. Como ejercicio analice paso a paso el código del programa y justifique la solución presentada como ejemplo anteriormente.

```
import java.io.*;
// Leer.class debe estar en la carpeta especificada por CLASSPATH
//
public class CContarPalabras
{
    // Contar caracteres, palabras y líneas en un texto
    public static void main(String[] args)
    {
        final char eof = (char)-1;
        char car;
        boolean palabra = false;
        int ncaracteres = 0, npalabras = 0, nlineas = 0;

        try
        {
            System.out.println("Introducir texto. " +
                "Pulse [Entrar] después de cada línea.");
            System.out.println("Para finalizar pulsar Ctrl+z.\n");

            while ((car = (char)System.in.read()) != eof)
            {
                // [Entrar] = CRLF = \r\n
                if (car == '\r') continue; // le sigue un \n
                ncaracteres++; // contador de caracteres

                // Eliminar blancos, tabuladores y finales de línea
                // entre palabras
                if (car == ' ' || car == '\n' || car == '\t')
                    palabra = false;
                else if (!palabra) // comienza una palabra
                {
                    npalabras++; // contador de palabras
                    palabra = true;
                }
                if (car == '\n') // finaliza una línea
                    nlineas++; // contador de líneas
            }
            System.out.println();
        }
    }
}
```

```
        System.out.println(ncaracteres + " " + npalabras + " " +
                           nlineas);
    }
    catch(IOException ignorada) {}
}
```

4. Realizar un programa que a través de un menú permita realizar las operaciones de *sumar, restar, multiplicar, dividir y salir*. Las operaciones constarán solamente de dos operandos. El menú será visualizado por un método sin argumentos, que devolverá como resultado la opción elegida. La ejecución será de la forma siguiente:

1. sumar
 2. restar
 3. multiplicar
 4. dividir
 5. salir

Seleccione la operación deseada: 3

Dato 1: 2.5

Dato 1: 2.

Resultado = 25.0

Pulse [Entrar] para continuar

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables y los prototipos de las funciones que van a intervenir en el programa.

```
double dato1 = 0, dato2 = 0, resultado = 0;  
int operación = 0;
```

- A continuación presentamos el menú en la pantalla para poder elegir la operación a realizar.

operación = menú():

El método *menú* será definido como un método **static** de la clase aplicación para que se pueda invocar sin tener que definir un objeto de esa clase. La definición de este método puede ser así:

```
static int menú()
{
    int op;
    do
    {
        System.out.println("\t1. sumar");
    }
```

```

        System.out.println("\t2. restar");
        System.out.println("\t3. multiplicar");
        System.out.println("\t4. dividir");
        System.out.println("\t5. salir");
        System.out.print("\nSeleccione la operación deseada: ");
        op = Leer.datoInt();
    }
    while (op < 1 || op > 5);
    return op;
}

```

- Si la operación elegida no ha sido *salir*, leemos los operandos *dato1* y *dato2*.

```

if (operación != 5)
{
    // Leer datos
    System.out.print("Dato 1: "); dato1 = Leer.datoDouble();
    System.out.print("Dato 2: "); dato2 = Leer.datoDouble();

    // Realizar la operación
}
else
    break; // salir

```

- A continuación, realizamos la operación elegida con los datos leídos e imprimimos el resultado.

```

switch (operación)
{
    case 1:
        resultado = dato1 + dato2;
        break;

    case 2:
        resultado = dato1 - dato2;
        break;
    case 3:
        resultado = dato1 * dato2;
        break;
    case 4:
        resultado = dato1 / dato2;
        break;
}
// Escribir el resultado
System.out.println("Resultado = " + resultado);
// Hacer una pausa
System.out.println("Pulse [Entrar] para continuar");
System.in.read();

```

- Las operaciones descritas formarán parte de un bucle infinito formado por una sentencia **while** con el fin de poder encadenar distintas operaciones.

```
while (true)
{
    // sentencias
}
```

El programa completo se muestra a continuación.

```
import java.io.*;
// Leer.class debe estar en la carpeta especificada por CLASSPATH
//
public class CCalculadora
{
    // Simulación de una calculadora
    static int menú()
    {
        int op;
        do
        {
            System.out.println("\t1. sumar");
            System.out.println("\t2. restar");
            System.out.println("\t3. multiplicar");
            System.out.println("\t4. dividir");
            System.out.println("\t5. salir");
            System.out.print("\nSeleccione la operación deseada: ");
            op = Leer.datToInt();
        }
        while (op < 1 || op > 5);

        return op;
    }

    public static void main(String[] args)
    {
        double dato1 = 0, dato2 = 0, resultado = 0;
        int operación = 0;

        try
        {
            while (true)
            {
                operación = menú();
                if (operación != 5)
                {
                    // Leer datos
                    System.out.print("Dato 1: "); dato1 = Leer.datToInt();
                    System.out.print("Dato 2: "); dato2 = Leer.datToInt();
                    if (operación == 1)
                        resultado = dato1 + dato2;
                    else if (operación == 2)
                        resultado = dato1 - dato2;
                    else if (operación == 3)
                        resultado = dato1 * dato2;
                    else if (operación == 4)
                        resultado = dato1 / dato2;
                    System.out.println("El resultado es: " + resultado);
                }
            }
        }
    }
}
```

```

// Limpiar el buffer del flujo de entrada
System.in.skip(System.in.available());
// Realizar la operación
switch (operación)
{
    case 1:
        resultado = dato1 + dato2;
        break;
    case 2:
        resultado = dato1 - dato2;
        break;
    case 3:
        resultado = dato1 * dato2;
        break;
    case 4:
        resultado = dato1 / dato2;
        break;
}
// Escribir el resultado
System.out.println("Resultado = " + resultado);
// Hacer una pausa
System.out.println("Pulse [Entrar] para continuar");
System.in.read();
// Limpiar el buffer del flujo de entrada
System.in.skip(System.in.available());
}
else
    break;
}
catch(IOException ignorada) {}

```

EJERCICIOS PROPUESTOS

- Realizar un programa que calcule e imprima la suma de los múltiplos de 5 comprendidos entre dos valores a y b . El programa no permitirá introducir valores negativos para a y b , y verificará que a es menor que b . Si a es mayor que b , intercambiará estos valores.
 - Realizar un programa que permita evaluar la serie:

$$\sum_{a=0}^b \frac{1}{x+ay}$$

3. Si quiere averiguar su número de Tarot, sume los números de su fecha de nacimiento y a continuación redúzcalos a un único dígito; por ejemplo si su fecha de nacimiento fuera 17 de Octubre de 1970, los cálculos a realizar serían:

$$17 + 10 + 1970 = 1997 \Rightarrow 1 + 9 + 9 + 7 = 26 \Rightarrow 2 + 6 = 8$$

lo que quiere decir que su número de Tarot es el 8.

Realizar un programa que pida una fecha, de la forma:

día del mes de año

donde *día*, *mes* y *año* son enteros, y dé como resultado el número de Tarot. El programa verificará si la fecha es correcta, esto es, los valores están dentro de los rangos permitidos.

4. Realizar un programa que genere la siguiente secuencia de dígitos:

El número de filas estará comprendido entre 11 y 20 y el resultado aparecerá centrado en la pantalla como se indica en la figura.

- Realizar un programa para jugar con el ordenador a acertar números. El ordenador piensa un número y nosotros debemos de acertar cuál es, en un número de intentos determinado. Por cada intento sin éxito el ordenador nos irá indicando si el número especificado es mayor o menor que el pensado por él. El número pensado por el ordenador se puede obtener multiplicando por una constante el valor devuelto por el método **random** de la clase **Math**, y los números pensados por nosotros los introduciremos por el teclado.
 - Un centro numérico es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números, cuyas sumas son iguales. El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1, 2, 3, 4, 5)

y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595. Escribir un programa que calcule los centros numéricos entre 1 y n .

7. Realizar un programa que solicite un texto (suponer que los caracteres que forman el texto son sólo letras, espacios en blanco, comas y el punto como final del texto) y a continuación lo escriba modificado de forma que, a la A le corresponda la K, a la B la L, ..., a la O la Y, a la P la Z, a la Q la A, ... y a la Z la J, e igual para las letras minúsculas. Suponga que la entrada no excede de una línea y que finaliza con un punto.

Al realizar este programa tenga en cuenta que el tipo **char** es un tipo entero, por lo tanto las afirmaciones en los ejemplos siguientes son correctas:

- ‘A’ es menor que ‘a’; es equivalente a decir que 65 es menor que 97, porque el valor ASCII de ‘A’ es 65 y el de ‘a’ es 97.
- ‘A’ + 3 es igual a ‘D’; es equivalente a decir que 65 + 3 es igual a 68, y este valor es el código ASCII del carácter ‘D’.

INTRODUCCIÓN A LAS MATRICES

CAPÍTULO 7

© F.J.Ceballos/RA-MA

MATRICES

Hasta ahora sólo hemos tenido que trabajar con algunas variables en cada uno de los programas que hemos realizado. Sin embargo, en más de una ocasión tendremos que manipular conjuntos más grandes de valores. Por ejemplo, para calcular la temperatura media del mes de agosto necesitaremos conocer los 31 valores correspondientes a la temperatura media de cada día. En este caso, podríamos utilizar una variable para introducir los 31 valores, uno cada vez, y acumular la suma en otra variable. Pero ¿qué ocurriría con los valores que vayamos introduciendo? que cuando tecleemos el segundo valor, el primero se perderá; cuando tecleemos el tercero, el segundo se perderá, y así sucesivamente. Cuando hayamos introducido todos los valores podremos calcular la media, pero las temperaturas correspondientes a cada día se habrán perdido. ¿Qué podríamos hacer para almacenar todos esos valores? Pues, podríamos utilizar 31 variables diferentes; pero ¿qué pasaría si fueran 100 o más valores los que tuviéramos que registrar? Además de ser muy laborioso el definir cada una de las variables, el código se vería enormemente incrementado.

En este capítulo, aprenderá a registrar conjuntos de valores, todos del mismo tipo, en unas estructuras de datos llamadas *matrices*. Así mismo, aprenderá a registrar cadenas de caracteres, que no son más que conjuntos de caracteres, o bien, si lo prefiere, matrices de caracteres.

Si las matrices son la forma de registrar conjuntos de valores, todos del mismo tipo (**int**, **float**, **double**, **char**, **String**, etc.), ¿qué haremos para almacenar un conjunto de valores relacionados entre sí, pero de diferentes tipos? Por ejemplo, almacenar los datos relativos a una persona como su *nombre*, *dirección*, *teléfono*, etc. Ya hemos visto que esto se hace definiendo una clase; en este caso, podría ser la clase de objetos *persona*. Posteriormente podremos crear también matrices de objetos, cuestión que aprenderemos más adelante.

INTRODUCCIÓN A LAS MATRICES

Una matriz es una estructura homogénea, compuesta por varios elementos, todos del mismo tipo y almacenados consecutivamente en memoria. Cada elemento puede ser accedido directamente por el nombre de la variable matriz seguido de uno o más subíndices encerrados entre corchetes.



En general, la representación de las matrices se hace mediante variables suscritas o de subíndices y pueden tener una o varias dimensiones (subíndices). A las matrices de una dimensión se les llama también listas y a los de dos dimensiones, tablas.

Desde un punto de vista matemático, en más de una ocasión necesitaremos utilizar variables subindexadas tales como:

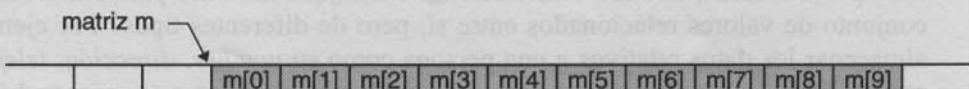
$$v = [a_0, a_1, a_2, \dots, a_i, \dots, a_n]$$

en el caso de un subíndice, o bien

$$m = \begin{pmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0j} & \dots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{i0} & a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \end{pmatrix}$$

si se utilizan dos subíndices. Esta misma representación se puede utilizar desde un lenguaje de programación recurriendo a las matrices que acabamos de definir y que a continuación se estudian.

Por ejemplo, supongamos que tenemos una matriz unidimensional de enteros llamada *m*, la cual contiene 10 elementos. Estos elementos se identificarán de la siguiente forma:



Observe que los subíndices son enteros consecutivos, y que el primer subíndice vale 0. Un subíndice puede ser cualquier expresión entera positiva.

Así mismo, una matriz de dos dimensiones se representa mediante una variable con dos subíndices (filas, columnas); una matriz de tres dimensiones se representa mediante una variable con tres subíndices etc. El número máximo de dimensiones o el número máximo de elementos, dentro de los límites establecidos por el compilador, para una matriz depende de la memoria disponible.

Entonces, las matrices según su dimensión se clasifican en unidimensionales y multidimensionales; y según su contenido, en numéricas, de caracteres y de referencias a objetos.

En Java, cada elemento de una matriz unidimensional es de un tipo primitivo, o bien una referencia a un objeto; y cada elemento de una matriz multidimensional es, a su vez, una referencia a otra matriz. A continuación se estudia todo esto detalladamente.

MATRICES NUMÉRICAS UNIDIMENSIONALES

Igual que sucede con otras variables, antes de utilizar una matriz hay que declararla. La declaración de una matriz especifica el nombre de la matriz y el tipo de elementos de la misma.

Para crear y utilizar una matriz hay que realizar tres operaciones: declararla, crearla e iniciarla.

Declarar una matriz

La declaración de una matriz de una dimensión, se hace indistintamente de una de las dos formas siguientes:

```
tipo[] nombre;
tipo nombre[];
```

donde *tipo* indica el tipo de los elementos de la matriz, que pueden ser de cualquier tipo primitivo o referenciado; y *nombre* es un identificador que nombra a la matriz. Los corchetes modifican la definición normal del identificador para que sea interpretado por el compilador como una matriz.

Las siguientes líneas de código son ejemplos de declaraciones de matrices:

```
int[] m;
float[] temperatura;
COrdenador[] ordenador; // COrdenador es una clase de objetos
```

La primera línea declara una matriz de elementos de tipo **int**; la segunda, una matriz de elementos de tipo **float**; y la tercera una matriz de objetos **COrdenador**.

Notar que las declaraciones no especifican el tamaño de la matriz. El tamaño será especificado cuando se cree la matriz, operación que se hará durante la ejecución del programa.

Según se ha podido observar, los corchetes se pueden colocar también después del nombre de la matriz. Por lo tanto, las declaraciones anteriores pueden escribirse también así:

```
int m[];  
float temperatura[];  
COrdenador ordenador[]; // COrdenador es una clase de objetos
```

Crear una matriz

Después de haber declarado una matriz, el siguiente paso es crearla o construirla. Crear una matriz significa reservar la cantidad de memoria necesaria para contener todos sus elementos y asignar al nombre de la matriz una referencia a ese bloque. Esto puede expresarse genéricamente así:

```
nombre = new tipo[tamaño];
```

donde *nombre* es el nombre de la matriz previamente declarada; *tipo* es el tipo de los elementos de la matriz; y *tamaño* es una expresión entera positiva menor o igual que la precisión de un **int**, que especifica el número de elementos.

El hecho de utilizar el operador **new** significa que *Java implementa las matrices como objetos*, por lo tanto serán tratadas como cualquier otro objeto.

Las siguientes líneas de código crean las matrices declaradas en el ejemplo anterior:

```
m = new int[10];  
temperatura = new float[31];  
ordenador = new COrdenador[25];
```

La primera línea crea una matriz identificada por *m* con 10 elementos de tipo **int**; es decir, puede almacenar 10 valores enteros; el primer elemento es *m[0]* (se lee: *m* sub-cero), el segundo *m[1]*, ..., y el último *m[9]*. La segunda crea una matriz *temperatura* de 31 elementos de tipo **float**. Y la tercera crea una matriz *ordenador* de 25 elementos, cada uno de los cuales puede referenciar a un objeto **COrdenador**. Una matriz de objetos es una matriz de referencias a dichos objetos.

Es bastante común declarar y crear la matriz en una misma línea. Esto puede hacerse así:

```
tipo[] nombre = new tipo[tamaño];
tipo nombre[] = new tipo[tamaño];
```

Las siguientes líneas de código declaran y crean las matrices expuestas en los ejemplos anteriores:

```
int[] m = new int[10];
float[] temperatura = new float[31];
COrdenador[] ordenador = new COrdenador[25];
```

Cuando se crea una matriz, el tamaño de la misma puede ser también especificado durante la ejecución a través de una variable a la que se asignará como valor el número de elementos requeridos. Por ejemplo, la última línea de código del ejemplo siguiente crea una matriz con el número de elementos especificados por la variable *nElementos*:

```
int nElementos;
System.out.print("Número de elementos de la matriz: ");
nElementos = Leer.datoInt();
int[] m = new int[nElementos];
```

Iniciar una matriz

Una matriz es un objeto; por lo tanto, cuando es creada, sus elementos son automáticamente iniciados, igual que sucedía con las variables miembro de una clase. Si la matriz es numérica, sus elementos son iniciados a 0 y si no es numérica, a un valor análogo al 0; por ejemplo, los caracteres son iniciados al valor '\u0000', un elemento booleano a **false** y las referencias a objetos, a **null**.

Si deseamos iniciar una matriz con otros valores diferentes a los predeterminados, podemos hacerlo de la siguiente forma:

```
float[] temperatura = {10.2F, 12.3F, 3.4F, 14.5F, 15.6F, 16.7F};
```

El ejemplo anterior crea una matriz *temperatura* de tipo **float** con tantos elementos como valores se hayan especificado entre llaves.

Acceder a los elementos de una matriz

Para acceder al valor de un elemento de una matriz se utiliza el nombre de la matriz, seguido de un subíndice entre corchetes. Esto es, un elemento de una matriz

no es más que una variable subindicada; por lo tanto, se puede utilizar exactamente igual que cualquier otra variable. Por ejemplo, en las operaciones que se muestran a continuación intervienen elementos de una matriz:

```
int[] m = new int[100];
int k = 0, a = 0;
// ...
a = m[1] + m[99];
k = 50;
m[k]++;
m[k+1] = m[k];
```

Observe que para referenciar un elemento de una matriz se puede emplear como subíndice una constante, una variable o una expresión de tipo entero. El subíndice especifica la posición del elemento dentro de la matriz. La primera posición es la 0.

Si se intenta acceder a un elemento con un subíndice menor que cero o mayor que el número de elementos de la matriz menos uno, Java lanzará una excepción de tipo **ArrayIndexOutOfBoundsException**, indicando que el subíndice está fuera de los límites establecidos cuando se creó la matriz. Por ejemplo, cuando se ejecute la última línea de código del ejemplo siguiente Java lanzará una excepción, puesto que intenta asignar el valor del elemento de subíndice 99 al elemento de subíndice 100, que está fuera del rango 0 a 99 válido.

```
int[] m = new int[100];
int k = 0, a = 0;
// ...
k = 99;
m[k+1] = m[k];
```

¿Cómo podemos asegurarnos de no exceder accidentalmente el final de una matriz? Verificando la longitud de la matriz mediante la variable estática **length**, que puede ser accedida por cualquier matriz. Ésta es el único atributo soportada por las matrices. Por ejemplo:

```
int n = m.length; // número de elementos de la matriz n
```

Métodos de una matriz

La clase genérica “matriz” proporciona un conjunto de métodos que ha heredado de la clase **Object** del paquete **java.lang**. Entre ellos cabe ahora destacar **equals** (*boolean equals(Object obj)*) y **clone** (*Object clone()*). El primero permite verificar si dos referencias se refieren a un mismo objeto, y el segundo permite duplicar un objeto (vea en el capítulo siguiente “La clase **Object**”).

Por ejemplo, el código expuesto a continuación crea una matriz *m2* que es una copia de otra matriz existente *m1*. Después pregunta si *m1* es igual a *m2*; el resultado será **false** puesto que *m1* y *m2* se refieren a matrices diferentes.

```
int[] m1 = {10, 20, 30, 40, 50};
int[] m2 = (int[])m1.clone(); // m2 es una copia de m1
if (m1.equals(m2)) // equivale a: if (m1 == m2)
    System.out.println("m1 y m2 se refieren a la misma matriz");
else
    System.out.println("m1 y m2 se refieren a matrices diferentes");
```

Trabajar con matrices unidimensionales

Para practicar la teoría expuesta hasta ahora, vamos a realizar un programa que asigne datos a una matriz unidimensional *m* de *nElementos* elementos y, a continuación, como comprobación del trabajo realizado, escriba el contenido de dicha matriz. La solución será similar a la siguiente:

```
Número de elementos de la matriz: 10
Introducir los valores de la matriz.
m[0]= 1
m[1]= 2
m[2]= 3
...
1 2 3 ...
Fin del proceso.
```

Para ello, en primer lugar definimos la variable *nElementos* para fijar el número de elementos de la matriz, creamos la matriz *m* con ese número de elementos y definimos el subíndice *i* para acceder a los elementos de dicha matriz.

```
int nElementos;
nElementos = Leer.datoInt();
int[] m = new int[nElementos]; // crear la matriz m
int i = 0; // subíndice
```

El paso siguiente es asignar un valor desde el teclado a cada elemento de la matriz.

```
for (i = 0; i < nElementos; i++)
{
    System.out.print("m[" + i + "] = ");
    m[i] = Leer.datoInt();
}
```

Una vez leída la matriz la visualizamos para comprobar el trabajo realizado.

```
for (i = 0; i < nElementos; i++)
    System.out.print(m[i] + " ");
```

El programa completo se muestra a continuación:

```
// Leer.class debe estar en la carpeta especificada por CLASSPATH
public class CMatrizUnidimensional
{
    // Creación de una matriz unidimensional
    public static void main(String[] args)
    {
        int nElementos;

        System.out.print("Número de elementos de la matriz: ");
        nElementos = Leer.datoInt();
        int[] m = new int[nElementos]; // crear la matriz m
        int i = 0; // subíndice

        System.out.println("Introducir los valores de la matriz.");
        for (i = 0; i < nElementos; i++)
        {
            System.out.print("m[" + i + "] = ");
            m[i] = Leer.datoInt();
        }

        // Visualizar los elementos de la matriz
        System.out.println();
        for (i = 0; i < nElementos; i++)
            System.out.print(m[i] + " ");
        System.out.println("\n\nFin del proceso.");
    }
}
```

El ejercicio anterior nos enseña cómo leer una matriz y cómo escribirla. El paso siguiente es aprender a trabajar con los valores almacenados en la matriz. Por ejemplo, pensemos en un programa que lea la nota media obtenida por cada alumno de un determinado curso, las almacene en una matriz y dé como resultado la nota media del curso.

Igual que hicimos en el programa anterior, en primer lugar crearemos una matriz *nota* con un número determinado de elementos solicitado a través del teclado. No se permitirá que este valor sea negativo. En este caso interesa que la matriz sea de tipo **float** para que sus elementos puedan almacenar un valor con decimales. También definiremos un índice *i* para acceder a los elementos de la matriz, y una variable *suma* para almacenar la suma total de todas las notas.

```

int nAlumnos; // número de alumnos
do
{
    System.out.print("Número de alumnos: ");
    nAlumnos = Leer.datoInt();
}
while (nAlumnos < 1);
float[] nota = new float[nAlumnos]; // crear la matriz nota
int i = 0; // subíndice
float suma = 0F; // suma total de las notas medias

```

El paso siguiente será almacenar en la matriz las notas introducidas a través del teclado.

```

for (i = 0; i < nota.length; i++)
{
    System.out.print("Nota media del alumno " + (i+1) + ": ");
    nota[i] = Leer.datoFloat();
}

```

Finalmente se suman todas la notas y se visualiza la nota media. La suma se almacenará en la variable *suma*. Una variable utilizada de esta forma recibe el nombre de acumulador. Es importante que observe que inicialmente su valor es cero.

```

for (i = 0; i < nota.length; i++)
    suma += nota[i];
System.out.println("\n\nNota media del curso: " + suma / nAlumnos);

```

El programa completo se muestra a continuación.

```

// Leer.class debe estar en la carpeta especificada por CLASSPATH
public class CMatrizUnidimensional
{
    // Trabajar con una matriz unidimensional
    public static void main(String[] args)
    {
        int nAlumnos; // número de alumnos (valor no negativo)
        do
        {
            System.out.print("Número de alumnos: ");
            nAlumnos = Leer.datoInt();
        }
        while (nAlumnos < 1);

        float[] nota = new float[nAlumnos]; // crear la matriz nota
        int i = 0; // subíndice
        float suma = 0F; // suma total de las notas medias
        System.out.println("Introducir las notas medias del curso.");
    }
}

```

```

for (i = 0; i < nota.length; i++)
{
    System.out.print("Nota media del alumno " + (i+1) + ": ");
    nota[i] = Leer.datoFloat();
}

// Sumar las notas medias
for (i = 0; i < nota.length; i++)
    suma += nota[i];

// Visualizar la nota media del curso
System.out.println("\n\nNota media del curso: " + suma / nAlumnos);
}
}

```

Los dos bucles **for** de la aplicación anterior podrían reducirse a uno como se indica a continuación. No se ha hecho por motivos didácticos.

```

for (i = 0; i < nota.length; i++)
{
    System.out.print("Nota media del alumno " + (i+1) + ": ");
    nota[i] = Leer.datoFloat();
    suma += nota[i];
}

```

Matrices asociativas

Cuando el índice de una matriz se corresponde con un dato, se dice que la matriz es asociativa (por ejemplo, una matriz *díasMes[13]* que almacene en el elemento de índice 1 los días del mes 1, en el de índice 2 los días del mes 2 y así sucesivamente; ignoramos el elemento de índice 0). En estos casos, la solución del problema resultará más fácil si utilizamos esa coincidencia. Por ejemplo, vamos a realizar un programa que cuente el número de veces que aparece cada una de las letras de un texto introducido por el teclado y a continuación imprima el resultado. Para hacer el ejemplo sencillo, vamos a suponer que el texto sólo contiene letras minúsculas del alfabeto inglés (no hay ni letras acentuadas, ni la *ll*, ni la *ñ*). La solución podría ser de la forma siguiente:

Introducir un texto.

Para finalizar pulsar [Ctrl][z]

las matrices mas utilizadas son las unidimensionales
y las bidimensionales.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
9	1	1	3	5	0	0	0	9	0	0	6	4	6	3	0	0	1	11	2	2	0	0	0	1	1

Antes de empezar el problema, vamos a analizar algunas de las operaciones que después utilizaremos en el programa. Por ejemplo, la expresión:

'z' - 'a' + 1

da como resultado 26. Recuerde que cada carácter tiene asociado un valor entero (código ASCII) que es el que utiliza la máquina internamente para manipularlo. Así por ejemplo la 'z' tiene asociado el entero 122, la 'a' el 97, etc. Según esto, la evaluación de la expresión anterior es: $122 - 97 + 1 = 26$.

Por la misma razón, si realizamos las declaraciones,

```
int[] c = new int[256]; // la tabla ASCII tiene 256 caracteres
char car = 'a';           // car tiene asignado el entero 97
```

la siguiente sentencia asigna a $c[97]$ el valor 10,

```
c['a'] = 10;
```

y esta otra sentencia que se muestra a continuación realiza la misma operación, lógicamente, suponiendo que *car* tiene asignado el carácter 'a'.

```
c[car] = 10;
```

Entonces, si leemos un carácter (de la 'a' a la 'z'),

```
car = (char)System.in.read();
```

y a continuación realizamos la operación,

```
c[car]++;
```

¿qué elemento de la matriz *c* se ha incrementado? La respuesta es el de subíndice igual al código correspondiente al carácter leído. Hemos hecho coincidir el carácter leído con el subíndice de la matriz. Así cada vez que leamos una 'a' se incrementará el contador $c[97]$ o lo que es lo mismo $c['a']$; tenemos entonces un contador de 'a'. Análogamente diremos para el resto de los caracteres.

Pero ¿qué pasa con los elementos $c[0]$ a $c[96]$? Según hemos planteado el problema inicial quedarían sin utilizar (el enunciado decía: con qué frecuencia aparecen los caracteres de la 'a' a la 'z'). Esto, aunque no presenta ningún problema, se puede evitar así:

```
c[car - 'a']++;
```

Para *car* igual a ‘*a*’ se trataría del elemento *c[0]* y para *car* igual a ‘*z*’ se trataría del elemento *c[25]*. De esta forma podemos definir una matriz de enteros justamente con un número de elementos igual al número de caracteres de la ‘*a*’ a la ‘*z*’ (26 caracteres según la tabla ASCII). El primer elemento será el contador de ‘*a*’, el segundo el de ‘*b*’, y así sucesivamente.

Un contador es una variable que inicialmente vale cero (suponiendo que la cuenta empieza desde uno) y que después se incrementa en una unidad cada vez que ocurre el suceso que se desea contar.

El programa completo se muestra a continuación.

```
import java.io.*;

// Leer.class debe estar en la carpeta especificada por CLASSPATH
public class CMatrizAsociativa
{
    // Frecuencia con la que aparecen las letras en un texto.
    public static void main(String[] args)
    {
        // Crear la matriz c con 'z'-'a'+1 elementos.
        // Java inicia los elementos de la matriz a cero.
        int[] c = new int['z'-'a'+1];

        char car; // subíndice
        final char eof = (char)-1;

        // Entrada de datos y cálculo de la tabla de frecuencias
        System.out.println("Introducir un texto.");
        System.out.println("Para finalizar pulsar [Ctrl][z]\n");
        try
        {
            // Leer el siguiente carácter del texto y contabilizarlo
            while ((car = (char)System.in.read()) != eof)
            {
                // Si el carácter leído está entre la 'a' y la 'z'
                // incrementar el contador correspondiente
                if (car >= 'a' && car <= 'z')
                    c[car - 'a']++;
            }
        }
        catch (IOException ignorada) {}
        // Mostrar la tabla de frecuencias
        System.out.println("\n");
        // Visualizar una cabecera "a b c ... "
        for (car = 'a'; car <= 'z'; car++)
            System.out.print(" " + car);
        System.out.println("\n -----" + "-----");
    }
}
```

```

// Visualizar la frecuencia con la que han aparecido los caracteres
for (car = 'a'; car <= 'z'; car++)
    System.out.print(" " + c[car - 'a']);
System.out.println();
}
}

```

CADERAS DE CARACTERES

Las cadenas de caracteres en Java son objetos de la clase **String**. Cuando expusimos los literales en el capítulo 3 vimos que cada vez que en un programa se utiliza un literal de caracteres, Java crea de forma automática un objeto **String** con el valor del literal. Por ejemplo, la línea de código siguiente visualiza el literal “Fin del proceso.”, para lo cual, Java previamente lo convierte en un objeto **String**:

```
System.out.println("Fin del proceso.");
```

Básicamente, una cadena de caracteres se almacena como una matriz unidimensional de elementos de tipo **char**:

```
char[] cadena = new char[10];
```

Igual que sucedía con las matrices numéricas, una matriz unidimensional de caracteres puede ser iniciada en el momento de su definición. Por ejemplo:

```
char[] cadena = {'a', 'b', 'c', 'd'};
```

Este ejemplo define *cadena* como una matriz de caracteres con cuatro elementos (*cadena[0]* a *cadena[3]*) y asigna al primer elemento el carácter ‘a’, al segundo el carácter ‘b’, al tercero el carácter ‘c’ y al cuarto el carácter ‘d’.

Puesto que cada carácter es un entero, el ejemplo anterior podría escribirse también así:

```
char[] cadena = {97, 98, 99, 100};
```

Cada carácter tiene asociado un entero entre 0 y 65535 (código Unicode). Por ejemplo, a la ‘a’ le corresponde el valor 97, a la ‘b’ el valor 98, etc. (recuerde que los primeros 128 códigos Unicode coinciden con los primeros 128 códigos ASCII y ANSI; capítulo 3, tipo **char**).

Si se crea una matriz de caracteres y se le asigna un número de caracteres menor que su tamaño, el resto de los elementos quedan con el valor ‘\0’ con el que fueron iniciados. Por ejemplo:

```
char[] cadena = new char[10];
cadena[0] = 'a'; cadena[1] = 'b'; cadena[2] = 'c'; cadena[3] = 'd';
System.out.println(cadena);
```

La llamada a **println** permite visualizar la cadena. Se visualizan todos los caracteres hasta finalizar la matriz, incluidos los nulos ('\0').

Como ya se expuso al hablar de las matrices numéricas, un intento de acceder a un valor de un elemento con un subíndice fuera de los límites establecidos al crear la matriz, daría lugar a que Java lanzara una excepción durante la ejecución.

Leer y escribir una cadena de caracteres

En el capítulo 5, cuando se expusieron los flujos de entrada, vimos que una forma de leer un carácter del flujo **in** era utilizando el método **read**. Entonces, leer una cadena de caracteres supondrá ejecutar repetidas veces la ejecución de **read** y almacenar cada carácter leído en la siguiente posición libre de una matriz de caracteres. Por ejemplo:

```
char[] cadena = new char[40]; // matriz de 40 caracteres
int i = 0, car;
try
{
    System.out.print("Introducir un texto: ");
    while ((car = System.in.read()) != '\r' && i < cadena.length)
    {
        cadena[i] = (char)car;
        i++;
    }
    System.out.println("Texto introducido: " + cadena);
    System.out.println("Longitud del texto: " + i);
    System.out.println("Dimensión de la matriz: " + cadena.length);
}
catch(IOException ignorada) {}
```

El ejemplo anterior define la variable *cadena* como una matriz de caracteres de longitud 40. Después establece un bucle para leer los caracteres que se tecleen hasta que se pulse la tecla *Entrar*. Cada carácter leído se almacena en la siguiente posición libre de la matriz *cadena*. Finalmente se escribe el contenido de *cadena*, el número de caracteres almacenados, y la dimensión de la matriz. Se puede observar que el valor dado por el atributo **length** no es el número de caracteres almacenado en *cadena*, sino la dimensión de la matriz.

Observe que el bucle utilizado para leer los caracteres tecleados, podría haberse escrito también así:

```
while ((car = System.in.read()) != '\r' && i < cadena.length)
    cadena[i++] = (char)car;
```

En el capítulo 5 vimos también otra forma de leer una cadena de caracteres. Consiste en leer una línea de texto de un flujo de la clase **BufferedReader**, conectado al flujo **in**, utilizando el método **readLine**, y almacenarla en un objeto **String**. El método **readLine** lee hasta encontrar el carácter '**\r**', '**\n**' o los caracteres '**\r\n**' introducidos al pulsar la tecla *Entrar*; estos caracteres son leídos pero no almacenados, simplemente son interpretados como delimitadores. Por ejemplo:

```
// Definir un flujo de caracteres de entrada: flujoE
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader flujoE = new BufferedReader(isr);

// Definir una referencia al flujo estándar de salida: flujoS
PrintStream flujoS = System.out;

String cadena; // variable para almacenar una línea de texto
try
{
    flujoS.print("Introduzca un texto: ");
    cadena = flujoE.readLine(); // leer una línea de texto
    flujoS.println(cadena); // escribir la línea leída
}
catch (IOException ignorada) { }
```

El ejemplo anterior define en primer lugar un flujo de entrada, *flujoE*, del cual se podrán leer líneas de texto. Después, define una referencia, *flujoS*, al flujo de salida estándar; esto permitirá utilizar la referencia *flujoS* en lugar de **System.out**. Finalmente lee una línea de texto introducida a través del teclado. Con esa información, el método **readLine** crea un objeto y devuelve una referencia al mismo que es almacenada en *cadena*. Finalmente, la llamada a **println** permite visualizar el objeto **String**.

Comparando el método **read** con el método **readLine**, se puede observar que este último proporciona una forma más cómoda de leer cadenas de caracteres de un flujo y además, devuelve un objeto **String** cuyos métodos, como veremos a continuación, hacen muy fácil la manipulación de cadenas.

Una matriz de caracteres también puede ser convertida en un objeto **String**, según se muestra a continuación. Por ejemplo:

```
char[] cadena = new char[40]; // matriz de 40 caracteres
// ...
String scadena = new String(cadena);
```

Trabajar con cadenas de caracteres

El siguiente ejemplo lee una cadena de caracteres y a continuación visualiza el símbolo y el valor ASCII de cada uno de los caracteres de la cadena. La solución será de la forma:

```
Escriba una cadena de caracteres:  
Hola ¿qué tal?  
Carácter = 'H', código ASCII = 72  
Carácter = 'o', código ASCII = 111  
...
```

El problema consiste en definir una cadena de caracteres, *cadena*, y asignarle datos desde el teclado utilizando el método **read**. Una vez leída la cadena, se accede a cada uno de sus elementos (no olvide que son elementos de una matriz) y por cada uno de ellos se visualiza su contenido y el valor ASCII correspondiente.

Observar que el método **println** visualiza un elemento de tipo **char** como un carácter; por lo tanto, para visualizar su valor ASCII es necesario convertirlo explícitamente a **int**. El programa completo se muestra a continuación.

```
import java.io.*;  
public class CValorAscii  
{  
    // Examinar una cadena de caracteres almacenada en una matriz  
    public static void main(String[] args)  
    {  
        char[] cadena = new char[80]; // matriz de caracteres  
        int car, i = 0; // un carácter y el subíndice para la matriz  
  
        try  
        {  
            System.out.println("Escriba una cadena de caracteres:");  
            while ((car = System.in.read()) != '\r' && i < cadena.length)  
                cadena[i++] = (char)car;  
            // Examinar la matriz de caracteres  
            i = 0;  
            do  
            {  
                System.out.println("Carácter = " + cadena[i] +  
                    ", código ASCII = " + (int)cadena[i]);  
                i++;  
            }  
            while (i < cadena.length && cadena[i] != '\0');  
        }  
        catch(IOException ignorada) {}  
    }  
}
```

Cuando un usuario ejecute este programa, se le solicitará que introduzca una cadena. Por ejemplo:

cadena →

H	o	I	a		¿	q	u	é		t	a	l	?	\0	\0	...
---	---	---	---	--	---	---	---	---	--	---	---	---	---	----	----	-----

Observar que el bucle utilizado para examinar la cadena, para i igual a 0 accede al primer elemento de la matriz, para i igual a 1 al segundo, y así hasta llegar al final de la matriz o hasta encontrar un carácter nulo ('\0') que indica el final de los caracteres tecleados.

En el siguiente ejemplo se trata de escribir un programa que lea una línea de la entrada estándar y la almacene en una matriz de caracteres. A continuación, utilizando un método, deseamos convertir los caracteres escritos en minúsculas, a mayúsculas.

Si observa la tabla ASCII en los apéndices de este libro, comprobará que los caracteres 'A', ..., 'Z', 'a', ..., 'z' están consecutivos y en orden ascendente de su código (valores 65 a 122). Entonces, pasar un carácter de minúsculas a mayúsculas supone restar al valor entero (código ASCII) asociado con el carácter, la diferencia entre los códigos de ese carácter en minúscula y el mismo en mayúscula. Por ejemplo, la diferencia 'a'-'A' es $97 - 65 = 32$, y es la misma que 'b'-'B', que 'c'-'C', etc. Como ayuda relacionada con lo expuesto, puede repasar los conceptos que se expusieron en el apartado "Matrices asociativas" expuesto anteriormente en este mismo capítulo.

El método que realice esta operación recibirá como parámetro la matriz de caracteres que contiene el texto a convertir. Si el método se llama *MinusculasMayusculas* y la matriz *cadena*, la llamada será así:

MinusculasMayusculas(cadena);

Como se puede observar en el código mostrado a continuación, el método recibirá una referencia a la cadena que se desea pasar a mayúsculas. A continuación, accederá al primer elemento de la matriz y comprobará si se trata de una minúscula, en cuyo caso cambiará el valor ASCII almacenado en dicho elemento por el valor ASCII correspondiente a la mayúscula. Esto es:

```
static void MinusculasMayusculas(char[] str)
{
    int i = 0, desp = 'a' - 'A';
    for (i = 0; i < str.length && str[i] != '\0'; i++)
        if (str[i] >= 'a' && str[i] <= 'z')
            str[i] = (char)(str[i] - desp);
}
```

Observe que cuando se llama al método *MinusculasMayusculas*, lo que en realidad se pasa es una referencia al comienzo de la matriz. Por lo tanto, el método llamado y el método que llama, trabajan sobre la misma matriz, con lo que los cambios realizados por uno u otro son visibles para ambos.

El programa completo se muestra a continuación.

```
import java.io.*;
public class CCadenas
{
    // Convertir una cadena a Mayúsculas
    static void MinusculasMayusculas(char[] str)
    {
        int i = 0, desp = 'a' - 'A';
        for (i = 0; i < str.length && str[i] != '\0'; i++)
            if (str[i] >= 'a' && str[i] <= 'z')
                str[i] = (char)(str[i] - desp);
    }

    public static void main(String[] args)
    {
        char[] cadena = new char[80]; // matriz de caracteres
        int car, i = 0; // un carácter y el subíndice para la matriz

        try
        {
            System.out.println("Escriba una cadena de caracteres:");
            while ((car = System.in.read()) != '\r' && i < cadena.length)
                cadena[i++] = (char)car;
            // Convertir minúsculas a mayúsculas
            MinusculasMayusculas(cadena); // llamar al método
            System.out.println(cadena);
        }
        catch(IOException ignorada) {}
    }
}
```

La solución que se ha dado al problema planteado no contempla los caracteres típicos de nuestra lengua como la *ñ* o las vocales acentuadas. Este trabajo queda como ejercicio para el lector.

La utilización de matrices de caracteres para la solución de problemas puede ser ampliamente sustituida por objetos de la clase **String**. La gran cantidad y variedad de métodos aportados por esta clase facilitarán enormemente el trabajo con cadenas de caracteres, puesto que, como ya sabemos, un objeto **String** encapsula una cadena de caracteres.

Clase String

La clase **String**, que pertenece al paquete **java.lang**, proporciona métodos para examinar caracteres individuales de una cadena de caracteres, comparar cadenas, buscar y extraer subcademas, copiar cadenas y convertir cadenas a mayúsculas o a minúsculas. A continuación veremos algunos de los métodos más comunes de la clase **String**. Pero antes sepá que un objeto **String** representa una cadena de caracteres no modificable. Por lo tanto, una operación como convertir a mayúsculas no modificará el objeto original sino que devolverá un nuevo objeto con la cadena resultante de esa operación.

Así mismo, el lenguaje Java proporciona el operador + para concatenar objetos **String**, así como soporte para convertir otros objetos a objetos **String**. Por ejemplo, en la siguiente línea de código, Java debe convertir las expresiones que aparecen entre paréntesis en objetos **String**, antes de realizar la concatenación.

```
System.out.println("Dimensión de la matriz: " + cadena.length);
```

La concatenación de objetos **String** está implementada a través de la clase **StringBuffer** y la conversión, a través del método **toString** heredado de la clase **Object**. Tanto la clase como el método citados serán estudiados a continuación.

Recuerde que para acceder desde un método de la clase aplicación o de cualquier otra clase a un miembro (atributo o método) de un objeto de otra clase diferente se utiliza la sintaxis *objeto.miembro*. La interpretación que se hace en programación orientada a objetos es que el objeto ha recibido un mensaje, el especificado por el nombre del método, y responde ejecutando ese método. Los métodos **static** son una excepción a la regla (puede obtener más información en el apartado “Miembro de un objeto o de una clase” del capítulo 4).

String(String valor)

En el capítulo 4 hicimos un breve comentario acerca de que toda clase tiene al menos un método predeterminado especial denominado igual que ella, que es necesario invocar para crear un objeto; se trata del *constructor* de la clase, del cual aprenderemos más en un capítulo posterior. Según esto, **String** es el constructor de la clase **String**. Anteriormente, trabajando con cadenas de caracteres, vimos cómo utilizar este constructor para crear un objeto **String** a partir de una matriz de caracteres. Pero en la mayoría de los casos lo utilizaremos para crear un objeto **String** a partir de un literal o a partir de otro **String**. Por ejemplo, cada una de las líneas siguientes crea un **String**. Dejamos para un próximo capítulo las diferencias que hay entre utilizar una u otra forma, puesto que no repercuten en el código que escribimos debido a que los **String** son objetos no modificables.

```
String str1 = "abc";           // crea un String "abc"
String str2 = new String("def"); // crea un String "def"
String str3 = new String(str1); // crea un nuevo String "abc"
```

String **toString()**

Este método devuelve el propio objeto **String** que recibe el mensaje **toString**. Por ejemplo, el siguiente código copia la referencia *str1* en *str2* (no crea un objeto nuevo referenciado por *str2*, a partir de *str1*). El resultado es que las dos variables, *str1* y *str2*, permiten acceder al mismo objeto **String**.

```
String str1 = "abc", str2;
str2 = str1.toString(); // equivale a str2 = str1
```

La misma operación puede ser realizada utilizando la expresión *str2 = str1* lo cual ya fue expuesto en el apartado “Referencias a objetos” del capítulo 4.

String **concat(String str)**

Este método devuelve como resultado un nuevo objeto **String** resultado de concatenar el **String** especificado a continuación del objeto **String** que recibe el mensaje **concat**. Por ejemplo, la primera línea de código que se muestra a continuación da como resultado “Ayer llovió” y la segunda “Ayer llovió mucho”.

```
System.out.println("Ayer".concat(" llovió"));
System.out.println("Ayer".concat(" llovió".concat(" mucho")));
```

Si alguno de los **String** tienen longitud 0, se concatena una cadena nula. Este otro ejemplo que se muestra a continuación construye un objeto “abcdef” resultado de concatenar *str1* y *str2*, y asigna a *str1* la referencia al nuevo objeto.

```
String str1 = "abc", str2 = "def";
str1 = str1.concat(str2);
```

int **compareTo(String otroString)**

Este método compara lexicográficamente el **String** especificado, con el objeto **String** que recibe el mensaje **compareTo** (el método **equals** realiza la misma operación). El resultado devuelto es un entero:

< 0 si el **String** que recibe el mensaje es menor que el *otroString*,
= 0 si el **String** que recibe el mensaje es igual que el *otroString* y
> 0 si el **String** que recibe el mensaje es mayor que el *otroString*.

En otras palabras, el método **compareTo** permite saber si una cadena está en orden alfabético antes (es menor) o después (es mayor) que otra y el proceso que sigue es el mismo que nosotros ejercitamos cuando lo hacemos mentalmente, comparar las cadenas carácter a carácter. La comparación se realiza sobre los valores Unicode de cada carácter. El siguiente ejemplo compara dos cadenas y escribe "abcde" porque esta cadena está antes por orden alfabético.

```
String str1 = "abcde", str2 = "abcdefg";
if (str1.compareTo(str2) < 0)
    System.out.println(str1);
```

El método **compareTo** diferencia las mayúsculas de las minúsculas. Las mayúsculas están antes por orden alfabético. Esto es así porque en la tabla Unicode las mayúsculas tienen asociado un valor entero menor que las minúsculas. El siguiente ejemplo no escribe nada porque "abc" no está antes por orden alfabético que "Abc".

```
String str1 = "abc", str2 = "Abc";
if (str1.compareTo(str2) < 0)
    System.out.println(str1);
```

Si en vez de utilizar el método **compareTo** se utiliza el método **compareToIgnoreCase** no se hace diferencia entre mayúsculas y minúsculas. El resultado de ejecutar el siguiente programa es que *str1* y *str2* son iguales.

```
public class Test
{
    public static void main(String[] args)
    {
        String str1 = "La provincia de Santander es muy bonita";
        String str2 = "La provincia de SANTANDER es muy bonita";

        String strtemp;
        int resultado;

        resultado = str1.compareToIgnoreCase(str2);

        if( resultado > 0 )
            strtemp = "mayor que ";
        else if( resultado < 0 )
            strtemp = "menor que ";
        else
            strtemp = "igual a ";
        System.out.println( str1 + " es " + strtemp + str2 );
    }
}
```

int length()

Este método devuelve la longitud o número de caracteres Unicode (tipo **char**) del objeto **String** que recibe el mensaje **length**.

El siguiente ejemplo escribe como resultado: Longitud: 39

```
String str1 = "La provincia de Santander es muy bonita";
System.out.println("Longitud: " + str1.length());
```

String toLowerCase()

Este método convierte a minúsculas las letras mayúsculas del objeto **String** que recibe el mensaje **toLowerCase**. El resultado es un nuevo objeto **String** en minúsculas.

String toUpperCase()

Este método convierte a mayúsculas las letras minúsculas del objeto **String** que recibe el mensaje **toUpperCase**. El resultado es un nuevo objeto **String** en mayúsculas.

El siguiente ejemplo almacena en *str1* la cadena *str2* en mayúsculas.

```
String str1, str2 = "Santander, tu eres novia del mar...";
str1 = str2.toUpperCase();
```

String trim()

Este método devuelve un objeto **String** resultado de eliminar los espacios en blanco que pueda haber al principio y al final del objeto **String** que recibe el mensaje **trim**.

boolean startsWith(String prefijo)

Este método devuelve un valor **true** si el *prefijo* especificado coincide con el principio del objeto **String** que recibe el mensaje **startsWith**.

boolean endsWith(String sufijo)

Este método devuelve un valor **true** si el *sufijo* especificado coincide con el final del objeto **String** que recibe el mensaje **endsWith**. Un poco más adelante se muestra un ejemplo.

String substring(int IndiceInicial, int IndiceFinal)

Este método retorna un nuevo **String** que encapsula una subcadena de la cadena almacenada por el objeto **String** que recibe el mensaje **substring**. La subcadena empieza en *IndiceInicial* y se extiende hasta *IndiceFinal - 1*, o hasta el final si *IndiceFinal* no se especifica.

El siguiente ejemplo, elimina los espacios en blanco que haya al principio y al final de *str1*, verifica si *str1* finaliza con “gh” y en caso afirmativo obtiene de *str1* una subcadena *str2* igual a *str1* menos el sufijo “gh”.

```
String str1 = " abcdefgh ", str2 = "";
str1 = str1.trim();
if (str1.endsWith("gh"))
    str2 = str1.substring(0, str1.length() - "gh".length());
```

char charAt(int índice)

Este método devuelve el carácter que está en la posición especificada en el objeto **String** que recibe el mensaje **charAt**. El índice del primer carácter es el 0. Por lo tanto, el parámetro *índice* tiene que estar entre los valores 0 y **length()** - 1, de lo contrario Java lanzará un excepción.

int indexOf(int car)

Este método devuelve el índice de la primera ocurrencia del carácter especificado por *car* en el objeto **String** que recibe el mensaje **indexOf**. Si *car* no existe el método **indexOf** devuelve el valor -1. Puede comenzar la búsqueda por el final en lugar de hacerlo por el principio utilizando el método **lastIndexOf**.

int indexOf(String str)

Este método devuelve el índice de la primera ocurrencia de la subcadena especificada por *str* en el objeto **String** que recibe el mensaje **indexOf**. Si *str* no existe **indexOf** devuelve -1. Puede comenzar la búsqueda por el final en lugar de hacerlo por el principio utilizando el método **lastIndexOf**.

String replace(char car, char nuevoCar)

Este método devuelve un nuevo **String** resultado de reemplazar todas las ocurrencias *car* por *nuevoCar* en el objeto **String** que recibe el mensaje **replace**. Si el carácter *car* no existiera, entonces se devuelve el objeto **String** original.

static String valueOf(tipo dato)

Este método devuelve un nuevo **String** creado a partir del dato pasado como argumento. Puesto que el método es **static** no necesita ser invocado para un objeto **String**. El argumento puede ser de los tipos **boolean**, **char**, **char[]**, **int**, **long**, **float**, **double** y **Object**.

```
double pi = Math.PI;
String str1 = String.valueOf(pi);
```

char[] toCharArray()

Este método devuelve una matriz de caracteres creada a partir del objeto **String** que recibe el mensaje **toCharArray**.

```
String str = "abcde";
char[] mcar = str.toCharArray();
```

byte[] getBytes()

Este método devuelve una matriz de bytes creada a partir del objeto **String** que recibe el mensaje **getBytes**.

Clase StringBuffer

Del estudio de la clase **String** sabemos que un objeto de esta clase no es modificable. Se puede observar y comprobar que los métodos que actúan sobre un objeto **String** con la intención de modificarlo, no lo modifican, sino que devuelven un objeto nuevo con las modificaciones solicitadas. En cambio, un objeto **StringBuffer** es un objeto modificable tanto en contenido como en tamaño.

Algunos de los métodos más interesantes que proporciona la clase **StringBuffer**, perteneciente al paquete **java.lang**, son los siguientes:

StringBuffer([arg])

Este método permite crear un objeto de la clase **StringBuffer**. El siguiente ejemplo muestra las tres formas posibles de invocar a este método:

```
StringBuffer strb1 = new StringBuffer();
StringBuffer strb2 = new StringBuffer(80);
StringBuffer strb3 = new StringBuffer("abcde");
System.out.println(strb1 + " " + strb1.length() + " " + strb1.capacity());
System.out.println(strb2 + " " + strb2.length() + " " + strb2.capacity());
```

```
System.out.println(strb3 + " " + strb3.length() + " " + strb3.capacity());
```

La ejecución de las líneas de código del ejemplo anterior, da lugar a los siguientes resultados:

```
0 16
0 80
abcde 5 21
```

A la vista de los resultados podemos deducir que cuando **StringBuffer** se invoca sin argumentos construye un objeto vacío con una capacidad inicial para 16 caracteres; cuando se invoca con un argumento entero, construye un objeto vacío con la capacidad especificada; y cuando se invoca con un **String** como argumento construye un objeto con la secuencia de caracteres proporcionada por el argumento y una capacidad igual al número de caracteres almacenados más 16.

int length()

Este método devuelve la longitud o número de caracteres Unicode (tipo **char**) del objeto **StringBuffer** que recibe el mensaje **length**. Esta longitud puede ser modificada por el método **setLength** cuando sea necesario.

int capacity()

Este método devuelve la capacidad en caracteres Unicode (tipo **char**) del objeto **StringBuffer** que recibe el mensaje **capacity**.

StringBuffer append(tipo x)

Este método permite añadir la cadena de caracteres resultante de convertir el argumento *x* en un objeto **String**, al final del objeto **StringBuffer** que recibe el mensaje **append**. El *tipo* del argumento *x* puede ser **boolean**, **char**, **char[]**, **int**, **long**, **float**, **double**, **String** y **Object**. La longitud del objeto **StringBuffer** se incrementa en la longitud correspondiente al **String** añadido.

StringBuffer insert(int índice, tipo x)

Este método permite insertar la cadena de caracteres resultante de convertir el argumento *x* en un objeto **String**, en el objeto **StringBuffer** que recibe el mensaje **insert**. Los caracteres serán añadidos a partir de la posición especificada por el argumento *índice*. El *tipo* del argumento *x* puede ser **boolean**, **char**, **char[]**, **int**, **long**, **float**, **double**, **String** y **Object**. La longitud del objeto **StringBuffer** se incrementa en la longitud correspondiente al **String** insertado.

El siguiente ejemplo crea un objeto **StringBuffer** con la cadena “Mes de del año”, a continuación inserta la cadena “Abril ” a partir de la posición 7, y finalmente añade al final, la cadena representativa del entero 2002. El resultado será “Mes de Abril del año 2002”.

```
StringBuffer strb = new StringBuffer("Mes de del año ");
strb.insert("Mes de ".length(), "Abril "); // "Mes de ".length()=7
strb.append(2002);
```

StringBuffer delete(int p1, int p2)

Este método elimina los caracteres que hay entre las posiciones *p1* y *p2 - 1* del objeto **StringBuffer** que recibe el mensaje **delete**. El valor *p2* debe ser mayor que *p1*. Si *p1* es igual que *p2*, no se efectuará ningún cambio y si es mayor Java lanzará una excepción.

Partiendo del ejemplo anterior, el siguiente ejemplo elimina la subcadena “Abril ” del objeto *strb* y añade en su misma posición la cadena “Mayo ”. El resultado será “Mes de Mayo del año 2002”.

```
StringBuffer strb = new StringBuffer("Mes de del año ");
strb.insert(7, "Abril ");
strb.append(2002);
strb.delete(7, 13);
strb.insert(7, "Mayo ");
```

StringBuffer replace(int p1, int p2, String str)

Este método reemplaza los caracteres que hay entre las posiciones *p1* y *p2 - 1* del objeto **StringBuffer** que recibe el mensaje **replace**, por los caracteres especificados por *str*. La longitud y la capacidad del objeto resultante serán ajustadas automáticamente al valor requerido. El valor *p2* debe ser mayor que *p1*. Si *p1* es igual que *p2*, la operación se convierte en una inserción, y si es mayor Java lanzará una excepción. Según lo expuesto, el ejemplo anterior, puede escribirse también así:

```
StringBuffer strb = new StringBuffer("Mes de del año ");
strb.insert(7, "Abril ");
strb.append(2002);
strb.replace(7, 13, "Mayo ");
```

StringBuffer reverse()

Este método reemplaza la cadena almacenada en el objeto **StringBuffer** que recibe el mensaje **reverse**, por la misma cadena pero invertida.

String substring(int IndiceInicial, int IndiceFinal)

Este método retorna un nuevo **String** que encapsula una subcadena de la cadena almacenada por el objeto **StringBuffer** que recibe el mensaje **substring**. La subcadena empieza en *IndiceInicial* y se extiende hasta *IndiceFinal - 1*, o hasta el final si *IndiceFinal* no se especifica.

char charAt(int índice)

Este método devuelve el carácter que está en la posición especificada en el objeto **StringBuffer** que recibe el mensaje **charAt**. El índice del primer carácter es el 0. Por lo tanto, el parámetro *índice* tiene que estar entre los valores 0 y **length()** - 1.

void setCharAt(int índice, char car)

Este método reemplaza el carácter que está en la posición especificada en el objeto **StringBuffer** que recibe el mensaje **setCharAt**, por el nuevo carácter especificado. El índice del primer carácter es el 0. Por lo tanto, el parámetro *índice* tiene que estar entre los valores 0 y **length()** - 1.

String toString()

Este método devuelve como resultado un nuevo **String** copia del objeto **StringBuffer** que recibe el mensaje **toString**.

El siguiente ejemplo copia la cadena almacenada en *strb* en *str*.

```
StringBuffer strb = new StringBuffer("abcde");
String str = strb.toString();
```

Clase StringTokenizer

Esta clase, perteneciente al paquete **java.util**, permite dividir una cadena de caracteres en una serie de elementos delimitados por unos determinados caracteres. De forma predeterminada los delimitadores son: el espacio en blanco, el tabulador horizontal (**\t**), el carácter nueva línea (**\n**), el retorno de carro (**\r**) y el avance de página (**\f**).

Un objeto **StringTokenizer** se construye a partir de un objeto **String**. Por ejemplo:

```
StringTokenizer cadena;
cadena = new StringTokenizer("uno, dos, tres y cuatro");
```

Para obtener los elementos de la cadena separados por los delimitadores, en este caso predeterminados, utilizaremos los métodos **hasMoreTokens** para saber si hay más elementos en la cadena, y **nextToken** para obtener el siguiente elemento. Por ejemplo:

```
while (cadena.hasMoreTokens())
    System.out.println(cadena.nextToken());
```

Cuando ejecutemos las cuatro líneas de código correspondientes a los dos ejemplos anteriores, el resultado que se mostrará será el siguiente:

```
uno,  
dos,  
tres  
y  
cuatro
```

También se pueden especificar los delimitadores en el instante de construir el objeto **StringTokenizer**. Por ejemplo, la siguiente línea de código especifica como delimitadores la coma y el espacio en blanco:

```
cadena = new StringTokenizer("uno, dos, tres y cuatro", ", ");
```

En este caso, el resultado que se obtendrá a partir del objeto *cadena* es el siguiente:

```
uno  
dos  
tres  
y  
cuatro
```

La diferencia con respecto a la versión anterior es que ahora no aparece la coma como parte integrante de los elementos, ya que se ha especificado como delimitador y los delimitadores no aparecen. Si queremos que los delimitadores aparezcan como un elemento más, basta especificar **true** como tercer argumento:

```
cadena = new StringTokenizer("uno, dos, tres y cuatro", ",", true);
```

Ahora el resultado será el siguiente (las líneas sombreadas corresponden a los delimitadores):

```
uno  
•  
dos  
•
```

tres

y

cuatro

Conversión de cadenas de caracteres a datos numéricos

Cuando una cadena de caracteres representa un número y se necesita realizar la conversión al valor numérico correspondiente, por ejemplo, para realizar una operación aritmética con él, hay que utilizar los métodos apropiados proporcionados por las clases del paquete **java.lang**: **Byte**, **Character**, **Short**, **Integer**, **Long**, **Float**, **Double** y **Boolean**. Para más detalles, recurra al capítulo 5, donde fueron expuestos los métodos aludidos. Por ejemplo:

```
String str1 = "1234";
int dato1 = Integer.parseInt(str1); // convertir a entero

String str2 = "12.34";
float dato2 = (new Float(str2)).floatValue(); // convertir a float
```

MATRICES DE REFERENCIAS A OBJETOS

Según lo estudiado a lo largo de este capítulo podemos decir que cada elemento de una matriz unidimensional es de un tipo primitivo, o bien una referencia a un objeto. Entonces ¿cómo procederíamos si necesitáramos almacenar las temperaturas medias de cada día durante los 12 meses de un año?, o bien ¿cómo procederíamos si necesitáramos almacenar la lista de nombres de los alumnos de una determinada clase? Razonando un poco, llegaremos a la conclusión de que utilizar matrices unidimensionales para resolver los problemas planteados supondrá posteriormente un difícil acceso a los datos almacenados; esto es, responder a las preguntas: ¿cuál es la temperatura media del 10 de mayo?, o bien ¿cuál es el nombre del alumno número 25 de la lista? será mucho más sencillo si los datos los almacenamos en forma de tabla; en el caso de las temperaturas, una tabla de 12 filas (tantas como meses) por 31 columnas (tantas como los días del mes más largo); y en el caso de los nombres, una tabla de tantas filas como alumnos, y tantas columnas como el número de caracteres del nombre más largo. Por lo tanto, una solución fácil para los problemas planteados exige el uso de matrices de dos dimensiones.

Una matriz multidimensional, como su nombre indica, es una matriz de dos o más dimensiones. Java no soporta matrices multidimensionales, pero se puede lograr la misma funcionalidad declarando matrices de matrices; las cuales, a su vez,

pueden también contener matrices, y así sucesivamente, hasta llegar a obtener el número de dimensiones deseadas.

Por ejemplo, en el caso de las temperaturas podríamos definir una matriz de 12 elementos para que cada uno de ellos almacenara una referencia a una matriz unidimensional de 31 elementos; y en el caso de los nombres podríamos definir una matriz de n elementos para que cada uno de ellos almacenara una referencia a una matriz unidimensional de m caracteres (un nombre). Las figuras mostradas en los siguientes apartados le ayudarán a comprender lo expuesto.

Matrices numéricas multidimensionales

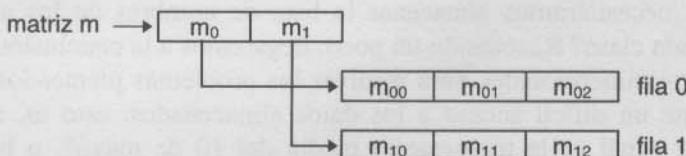
La definición de una matriz numérica de varias dimensiones se hace de la forma siguiente:

```
tipo[][]... nombre_matriz = new tipo[expr-1][expr-2]...;
```

donde *tipo* es un tipo primitivo entero o real. El número de elementos de una matriz multidimensional es el producto de las dimensiones indicadas por *expr-1*, *expr-2*, ... Por ejemplo, la línea de código siguiente crea una matriz de dos dimensiones con $2 \times 3 = 6$ elementos de tipo **int**:

```
int[][] m = new int[2][3];
```

A partir de la línea de código anterior, Java crea una matriz unidimensional *m* con 2 elementos *m[0]* y *m[1]* que son referencias a otras dos matrices unidimensionales de 3 elementos. Gráficamente podemos imaginarlo así:



Evidentemente, el tipo de los elementos de la matriz referenciada por *m* es **int[]** y el tipo de los elementos de las matrices referenciadas por *m[0]* y *m[1]* es **int**. Además, puede comprobar la existencia y la longitud de las matrices unidimensionales referenciadas por *m*, *m[0]* y *m[1]* utilizando el código siguiente:

```
int[][] m = new int[2][3];
System.out.println(m.length); // resultado: 2
System.out.println(m[0].length); // resultado: 3
System.out.println(m[1].length); // resultado: 3
```

Desde nuestro punto de vista, cuando se trate de matrices de dos dimensiones, es más fácil pensar en ellas como si de una tabla de f filas por c columnas se tratara. Por ejemplo:

matriz m	col 0	col 1	col 2
fila 0	m_{00}	m_{01}	m_{02}
fila 1	m_{10}	m_{11}	m_{12}

Para acceder a los elementos de la matriz m , puesto que se trata de una matriz de dos dimensiones, utilizaremos dos subíndices, el primero indicará la fila y el segundo la columna donde se localiza el elemento, según se puede observar en la figura anterior. Por ejemplo, la primera sentencia del ejemplo siguiente asigna el valor x al elemento que está en la fila 1, columna 2; y la segunda, asigna el valor de este elemento al elemento $m[0][1]$.

```
m[1][2] = x;
m[0][1] = m[1][2];
```

Como ejemplo de aplicación de matrices multidimensionales, vamos a realizar un programa que asigne datos a una matriz m de dos dimensiones y a continuación escriba las sumas correspondientes a las filas de la matriz. La ejecución del programa presentará el aspecto siguiente:

```
Número de filas de la matriz: 2
Número de columnas de la matriz: 2
Introducir los valores de la matriz.
m[0][0] = 2
m[0][1] = 5
m[1][0] = 3
m[1][1] = 6
Suma de la fila 0: 7.0
Suma de la fila 1: 9.0
```

Fin del proceso.

En primer lugar definimos las variables que almacenarán el número de filas y de columnas de la matriz, y a continuación leemos esos valores del teclado, deseando cualquier valor negativo.

```
int nfilas, ncols;      // filas y columnas de la matriz
do
{
    System.out.print("Número de filas de la matriz:   ");
    nfilas = Leer.datToInt();
}
while (nfilas < 1);    // no permitir un valor negativo
```

```

do
{
    System.out.print("Número de columnas de la matriz: ");
    ncols = Leer.datToInt();
}
while (ncols < 1); // no permitir un valor negativo

```

Después, creamos la matriz *m* con el número de filas y columnas especificado, definimos las variables *fila* y *col* que utilizaremos para manipular los subíndices correspondientes a la fila y a la columna, y la variable *sumafila* para almacenar la suma de los elementos de una fila:

```

float[][] m = new float[nfilas][ncols]; // crear la matriz m
int fila = 0, col = 0; // subíndices
float sumafila = 0; // suma de los elementos de una fila

```

El paso siguiente es asignar un valor desde el teclado a cada elemento de la matriz.

```

for (fila = 0; fila < nfilas; fila++)
{
    for (col = 0; col < ncols; col++)
    {
        System.out.print("m[" + fila + "][" + col + "] = ");
        m[fila][col] = Leer.datofloat();
    }
}

```

Una vez leída la matriz, calculamos la suma de cada fila y visualizamos los resultados para comprobar el trabajo realizado.

```

for (fila = 0; fila < nfilas; fila++)
{
    sumafila = 0;
    for (col = 0; col < ncols; col++)
        sumafila += m[fila][col];
    System.out.println("Suma de la fila " + fila + ": " + sumafila);
}

```

El programa completo se muestra a continuación.

```

// Leer.class debe estar en la carpeta especificada por CLASSPATH
public class CMatrizMultidimensional
{
    // Creación de una matriz multidimensional.
    // Suma de las filas de una matriz de dos dimensiones.
    public static void main(String[] args)
    {
        int nfilas, ncols; // filas y columnas de la matriz
    }
}

```

```

do
{
    System.out.print("Número de filas de la matriz:    ");
    nfilas = Leer.datoInt();
}
while (nfilas < 1); // no permitir un valor negativo
do
{
    System.out.print("Número de columnas de la matriz: ");
    ncols = Leer.datoInt();
}
while (ncols < 1); // no permitir un valor negativo

float[][] m = new float[nfilas][ncols]; // crear la matriz m
int fila = 0, col = 0; // subíndices
float sumafila = 0; // suma de los elementos de una fila

System.out.println("Introducir los valores de la matriz.");
for (fila = 0; fila < nfilas; fila++)
{
    for (col = 0; col < ncols; col++)
    {
        System.out.print("m[" + fila + "][" + col + "] = ");
        m[fila][col] = Leer.datoFloat();
    }
}

// Visualizar la suma de cada fila de la matriz
System.out.println();
for (fila = 0; fila < nfilas; fila++)
{
    sumafila = 0;
    for (col = 0; col < ncols; col++)
        sumafila += m[fila][col];

    System.out.println("Suma de la fila " + fila + ": " + sumafila);
}
System.out.println("\nFin del proceso.");
}
}

```

Seguramente habrá pensado que la suma de cada fila se podía haber hecho simultáneamente a la lectura tal como se indica a continuación.

```

for (fila = 0; fila < nfilas; fila++)
{
    sumafila = 0;
    for (col = 0; col < ncols; col++)
    {
        System.out.print("m[" + fila + "][" + col + "] = ");
    }
}

```

```

        m[fila][col] = Leer.datoFloat();
        sumafila += m[fila][col];
    }
    System.out.println("Suma de la fila " + fila + ": " + sumafila);
}

```

No obstante, esta forma de proceder presenta una diferencia a la hora de visualizar los resultados, y es que la suma de cada fila se presenta a continuación de haber leído los datos de la misma.

```

Número de filas de la matriz: 2
Número de columnas de la matriz: 2
Introducir los valores de la matriz.
m[0][0] = 2
m[0][1] = 5
Suma de la fila 0: 7.0
m[1][0] = 3
m[1][1] = 6
Suma de la fila 1: 9.0

```

Fin del proceso.

Con este último planteamiento, una solución para escribir los resultados al final sería almacenarlos en una matriz unidimensional y mostrar posteriormente la matriz. Este trabajo se deja como ejercicio para el lector.

Matrices de cadenas de caracteres

Las matrices de cadenas de caracteres son matrices multidimensionales, generalmente de dos dimensiones, en las que cada fila se corresponde con una cadena de caracteres. Entonces según lo estudiado, una fila puede ser un objeto matriz unidimensional, un objeto **String** o un objeto **StringBuffer**.

Haciendo un estudio análogo al realizado para las matrices numéricas multidimensionales, la definición de una matriz de cadenas de caracteres puede hacerse de la forma siguiente:

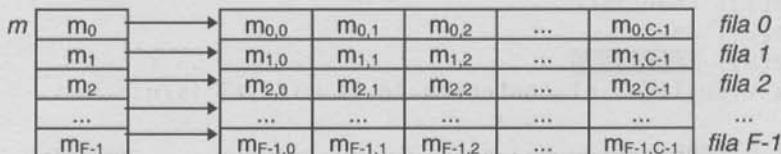
```
char[][] nombre_matriz = new char[filas][longitud_fila];
```

Por ejemplo, la línea de código siguiente crea una matriz de cadenas de caracteres de *F* filas por *C* caracteres máximo por cada fila.

```
char[][] m = new char[F][C];
```

A partir de la línea de código anterior, Java crea una matriz unidimensional *m* con los elementos, *m[0]*, *m[1]*, ..., *m[F-1]*, que son referencias a otras tantas ma-

trices unidimensionales de C elementos de tipo **char**. Gráficamente podemos imaginarlo así:



Evidentemente, el tipo de los elementos de la matriz referenciada por m es **char[]** y el tipo de los elementos de las matrices referenciadas por $m[0]$, $m[1]$, ..., es **char**. Desde nuestro punto de vista, es más fácil imaginarse una matriz de cadenas de caracteres como una lista. Por ejemplo, la matriz m del ejemplo anterior estará compuesta por las cadenas de caracteres $m[0]$, $m[1]$, $m[2]$, $m[3]$, etc.

m_0
m_1
m_2
m_3
...

Para acceder a los elementos de la matriz m , puesto que se trata de una matriz de cadenas de caracteres, utilizaremos sólo el primer subíndice, el que indica la fila. Sólo utilizaremos dos subíndices cuando sea necesario acceder a un carácter individual. Por ejemplo, la primera sentencia del ejemplo siguiente crea una matriz de cadenas de caracteres. La segunda asigna una cadena de caracteres a $m[0]$ desde el teclado; la cadena tendrá $nCarsPorFila$ caracteres como máximo y será almacenada a partir de la posición 0 de $m[0]$. Y la tercera sentencia, reemplaza el último carácter leído en $m[0]$ por '\0', puesto que **read** devuelve el número de caracteres leídos.

```
char[][] nombre = new char[nFilas][nCarsPorFila];
nCarsLeidos = flujoE.read(m[0], 0, nCarsPorFila);
nombre[0][nCarsLeidos-1] = '\0';
```

Es importante que asimile que $m[0]$, $m[1]$, etc. son cadenas de caracteres y que, por ejemplo, $m[1][3]$ es un carácter; el que está en la fila 1, columna 3.

Para ilustrar la forma de trabajar con cadenas de caracteres, vamos a realizar un programa que lea una lista de nombres y los almacene en una matriz. Una vez construida la matriz, visualizaremos su contenido.

La solución tendrá el aspecto siguiente:

Número de filas de la matriz: 10
Número de caracteres por fila: 40

Escriba los nombres que desea introducir.
 Puede finalizar pulsando las teclas [Ctrl][Z].
 Nombre[0]: Mª del Carmen
 Nombre[1]: Francisco
 Nombre[2]: Javier
 Nombre[3]: [Ctrl][Z]
 ¿Desea visualizar el contenido de la matriz? (s/n): S

Mª del Carmen
 Francisco
 Javier

La solución pasa por realizar los siguientes puntos:

1. Definir una matriz de cadenas, los índices y demás variables necesarias.
2. Establecer un bucle para leer las cadenas de caracteres utilizando el método **read**. La entrada de datos finalizará al introducir la marca de fin de fichero.
3. Preguntar al usuario del programa si quiere visualizar el contenido de la matriz.
4. Si la respuesta anterior es afirmativa, establecer un bucle para visualizar las cadenas de caracteres almacenadas en la matriz.

El programa completo se muestra a continuación.

```
import java.io.*;
// Utiliza Leer.class que está en CLASSPATH=c:\jdk1.3\misClases
public class CMatriz1Cadenas
{
    public static void main(String[] args)
    {
        try
        {
            // Definir un flujo de caracteres de entrada: flujoE
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader flujoE = new BufferedReader(isr);
            // Definir una referencia al flujo estándar de salida: flujoS
            PrintStream flujoS = System.out;
            int nFilas = 0, nCarsPorFila = 0;
            int fila = 0, nCarsLeidos = 0, eof = -1;
            do
            {
                System.out.print("Número de filas de la matriz: ");
                nFilas = Leer.datoInt();
            }
            while (nFilas < 1);           // no permitir un valor negativo
```

```

    do
    {
        System.out.print("Número de caracteres por fila: ");
        nCarsPorFila = Leer.datoInt();
    }
    while (nCarsPorFila < 1); // no permitir un valor negativo

    // Matriz de cadenas de caracteres
    char[][] nombre = new char[nFilas][nCarsPorFila];
    System.out.println("Escriba los nombres que desea introducir.");
    System.out.println("Puede finalizar pulsando las teclas [Ctrl][Z].");
    for (fila = 0; fila < nFilas; fila++)
    {
        flujoS.print("Nombre[" + fila + "]: ");
        nCarsLeidos = flujoE.read(nombre[fila], 0, nCarsPorFila);
        // Si se pulsó [Ctrl][Z], salir del bucle
        if (nCarsLeidos == eof) break;
        // Eliminar los caracteres CR LF
        nombre[fila][nCarsLeidos-1] = '\0';
        nombre[fila][nCarsLeidos-2] = '\0';
    }
    flujoS.print("\n\n");
    nFilas = fila; // número de filas leídas
    char respuesta;
    do
    {
        flujoS.print("¿Desea visualizar el contenido de la matriz? (s/n): ");
        respuesta = ((flujoE.readLine()).toLowerCase()).charAt(0);
    }
    while (respuesta != 's' && respuesta != 'n');
    if ( respuesta == 's' )
    {
        // Visualizar la lista de nombres
        flujoS.println();
        for (fila = 0; fila < nFilas; fila++)
            flujoS.println(nombre[fila]);
    }
}
catch (IOException ignorada) { }
}

```

El identificador *nombre* hace referencia a una matriz de caracteres de dos dimensiones. Una fila de esta matriz es una cadena de caracteres (una matriz de caracteres unidimensional) y la biblioteca de Java provee el método *read* para leer matrices unidimensionales de caracteres. Por eso, para leer una fila (una cadena de caracteres) utilizamos sólo un índice. Esto no es aplicable a las matrices numé-

ricas de dos dimensiones, ya que la biblioteca de Java no proporciona métodos para leer filas completas, lo cual es lógico.

Siguiendo con el análisis del programa anterior, la entrada de datos finalizará cuando se haya introducido la marca de fin de fichero, o bien cuando se hayan introducido la totalidad de los nombres.

Así mismo, una vez finalizada la entrada de datos, se lanza una pregunta acerca de si se desea visualizar el contenido de la matriz. En este caso la respuesta tecleada se obtiene con **readLine**. Como este método lee hasta el carácter '\n' inclusive, utilizamos el método **charAt** para obtener del **String** devuelto por **readLine**, el primer carácter leído, que deberá ser una 's' o bien una 'n'.

Observe la sentencia:

```
respuesta = ((flujoE.readLine()).toLowerCase()).charAt(0);
```

Es equivalente a:

```
String s = flujoE.readLine(); // leer una línea de texto  
s = s.toLowerCase(); // convertir el texto a mayúsculas  
respuesta = s.charAt(0); // obtener el primer carácter leido
```

¿De qué longitud son las cadenas de caracteres *nombre[0]*, *nombre[1]*, etc.? Independientemente del número de caracteres leídos para cada uno de los nombres solicitados, todas son de la misma longitud: *nCarsPorFila* caracteres (recuerde que las matrices de caracteres son iniciadas con nulos); para verificarlo puede recurrir al atributo **length** de las matrices. Evidentemente, esta forma de proceder supone un derroche de espacio de memoria, que se puede evitar haciendo que cadena fila de la matriz *nombre* tenga una longitud igual al número de caracteres del nombre que almacena. Apliquemos esta teoría al programa anterior.

El proceso que seguiremos para solucionar el problema planteado es el siguiente:

- Definimos la matriz de referencias a las matrices unidimensionales que serán las filas de una supuesta lista.

```
char[][] nombre = new char[nFilas][];
```

No asignamos memoria para cada una de las cadenas porque no conocemos su longitud (*nombre[0] = null*, *nombre[1] = null*, etc.). Por lo tanto, este proceso lo desarrollaremos paralelamente a la lectura de cada una de ellas.

- Leemos las cadenas de caracteres. Para poder leer una cadena, necesitamos definir una matriz de caracteres que vamos a denominar *unNombre*. Ésta será una matriz unidimensional de longitud 81 caracteres, por ejemplo.

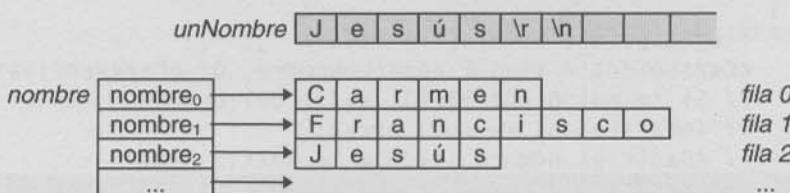
```
char[] unNombre = new char[81];
```

Una vez leída la cadena, conoceremos cuántos caracteres se han leído; entonces, reservamos memoria (**new**) para almacenar ese número de caracteres, almacenamos la referencia al bloque de memoria reservado en el siguiente elemento vacío de la matriz de referencias *nombre* y copiamos *unNombre* en el nuevo bloque asignado (fila de la matriz *nombre*). Este proceso lo repetiremos para cada uno de los nombres que leamos.

```
for (fila = 0; fila < nFilas; fila++)
{
    flujoS.print("Nombre[" + fila + "]: ");
    nCarsLeidos = flujoE.read(unNombre, 0, nCarsPorFila);
    // Si se pulsó [Ctrl][Z], salir del bucle
    if (nCarsLeidos == eof) break;

    // Añadir el nombre leído a la matriz nombre
    nombre[fila] = new char[nCarsLeidos-2]; // menos CR LF
    for (int i = 0; i < nCarsLeidos-2; i++)
        nombre[fila][i] = unNombre[i]; // copiar
}
```

Gráficamente puede imaginarse el proceso descrito de acuerdo a la siguiente estructura de datos:



La sentencia *nombre[fila] = new char[nCarsLeidos-2]* asigna para cada valor de *fila* un espacio de memoria de *nCarsLeidos-2* caracteres (en la figura: *fila 0*, *fila 1*, *fila 2*, etc.), para copiar la cadena leída a través de *unNombre*. Recuerde que el método **read** devuelve el número de caracteres leídos.

- Una vez leída la matriz la visualizamos si la respuesta a la petición de realizar este proceso es afirmativa.

El programa completo se muestra a continuación.

```

import java.io.*;
// Utiliza Leer.class que está en CLASSPATH=c:\jdk1.3\misClases
public class CMatriz2Cadenas
{
    public static void main(String[] args)
    {
        try
        {
            // Definir un flujo de caracteres de entrada: flujoE
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader flujoE = new BufferedReader(isr);

            // Definir una referencia al flujo estándar de salida: flujoS
            PrintStream flujoS = System.out;

            int nFilas = 0, nCarsPorFila = 81;
            int fila = 0, nCarsLeidos = 0, eof = -1;
            char[] unNombre = new char[nCarsPorFila];
            do
            {
                System.out.print("Número de filas de la matriz: ");
                nFilas = Leer.datoInt();
            }
            while (nFilas < 1); // no permitir un valor negativo

            // Matriz de cadenas de caracteres
            char[][] nombre = new char[nFilas][];
            System.out.println("Escriba los nombres que desea introducir.");
            System.out.println("Puede finalizar pulsando las teclas [Ctrl][Z].");
            for (fila = 0; fila < nFilas; fila++)
            {
                flujoS.print("Nombre[" + fila + "]: ");
                nCarsLeidos = flujoE.read(unNombre, 0, nCarsPorFila);
                // Si se pulsó [Ctrl][Z], salir del bucle
                if (nCarsLeidos == eof) break;
                // Añadir el nombre leído a la matriz nombre
                nombre[fila] = new char[nCarsLeidos-2]; // menos CR LF
                for (int i = 0; i < nCarsLeidos-2; i++)
                    nombre[fila][i] = unNombre[i]; // copiar
                flujoS.print("\n\n");
                nFilas = fila; // número de filas leídas
                // ...
                // continúa igual que en la versión anterior
            }
            catch (IOException ignorada) { }
        }
    }
}

```

Matrices de objetos String

En el apartado anterior hemos aprendido a manipular cadenas de caracteres a nivel de carácter. Pero Java proporciona las clases **String** y **StringBuffer** para hacer de las cadenas de caracteres objetos con sus atributos particulares, los cuales podrán ser accedidos por los métodos de sus clases. Desde este nivel de abstracción muchos de los problemas que se han presentado anteriormente y que hemos tenido que resolver, ahora simplemente no aparecerán con lo que la implementación del programa resultará más sencilla.

Para comprobar lo expuesto, vamos a resolver el programa anterior pero utilizando una matriz de objetos **String**. El proceso que seguiremos es el siguiente:

- Definimos la matriz de objetos **String**:

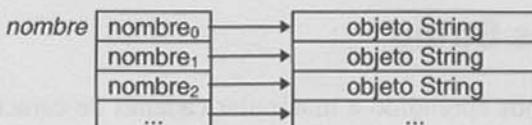
```
String[] nombre = new String[nFilas];
```

Cada elemento de esta matriz será iniciado por Java con el valor **null**, indicando así que la matriz inicialmente no referencia a ningún objeto **String**; esto es, la matriz está vacía.

- Leemos las cadenas de caracteres. Para poder leer una cadena, utilizaremos el método **readLine**. Recuerde que este método permite leer una cadena de caracteres hasta encontrar un carácter '**\n**', '**\r\n**' o '**\r\n\r\n**' (estos caracteres son leídos pero no almacenados) y devuelve una referencia a un objeto **String** que almacena la información leída; referencia que asignaremos al siguiente elemento vacío de la matriz *nombre*. Este proceso lo repetiremos para cada uno de los nombres que leamos. Recuerde también que si el método **readLine** intenta leer del flujo y se encuentra con el final del mismo, retornará la constante **null**.

```
for (fila = 0; fila < nFilas; fila++)
{
    flujoS.print("Nombre[" + fila + "]: ");
    nombre[fila] = flujoE.readLine();
    // Si se pulsó [Ctrl][Z], salir del bucle
    if (nombre[fila] == null) break;
}
```

Gráficamente puede imaginarse el proceso descrito de acuerdo a la siguiente estructura de datos, aunque para trabajar resulte más fácil pensar en una matriz unidimensional cuyos elementos *nombre[0]*, *nombre[1]*, etc. son cadenas de caracteres.



- Una vez leídos todos los nombres deseados los visualizamos si la respuesta a la petición de realizar este proceso es afirmativa.

El programa completo se muestra a continuación.

```

import java.io.*;
// Utiliza Leer.class que está en CLASSPATH=c:\jdk1.3\misClases

public class CMatriz3Cadenas
{
    public static void main(String[] args)
    {
        try
        {
            // Definir un flujo de caracteres de entrada: flujoE
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader flujoE = new BufferedReader(isr);

            // Definir una referencia al flujo estándar de salida: flujoS
            PrintStream flujoS = System.out;

            int nFilas = , fila = 0;
            do
            {
                System.out.print("Número de filas de la matriz: ");
                nFilas = Leer.datoInt();
            }
            while (nFilas < 1);      // no permitir un valor negativo

            // Matriz de cadenas de caracteres
            String[] nombre = new String[nFilas];

            System.out.println("Escriba los nombres que desea introducir.");
            System.out.println("Puede finalizar pulsando las teclas [Ctrl][Z].");
            for (fila = 0; fila < nFilas; fila++)
            {
                flujoS.print("Nombre[" + fila + "]: ");
                nombre[fila] = flujoE.readLine();
                // Si se pulsó [Ctrl][Z], salir del bucle
                if (nombre[fila] == null) break;
            }
            flujoS.print("\n\n");
            nFilas = fila; // número de filas leídas
            // ...
        }
    }
}
  
```

```
        // continúa igual que en la versión anterior  
    }  
    catch (IOException ignorada) { }  
}
```

Si en lugar de utilizar objetos **String** utilizamos objetos **StringBuffer**, las modificaciones son mínimas. Puede verlas en el código mostrado a continuación:

```
StringBuffer[] nombre = new StringBuffer[nFilas];
String sNombre;
for (fila = 0; fila < nFilas; fila++)
{
    flujoS.print("Nombre[" + fila + "]: ");
    // Si se pulsó [Ctrl][Z], salir del bucle
    if ((sNombre = flujoE.readLine()) == null) break;
    nombre[fila] = new StringBuffer(sNombre);
}
```

EJERCICIOS RESUELTOS

- Realizar un programa que lea una lista de valores introducida por el teclado. A continuación, y sobre la lista, buscar los valores máximo y mínimo, y escribirlos.

La solución de este problema puede ser de la siguiente forma:

- Definimos la matriz que va a contener la lista de valores y el resto de las variables necesarias en el programa.

```
int nElementos; // número de elementos (valor no negativo)

do
{
    System.out.print("Número de valores que desea introducir: ");
    nElementos = Leer.datoInt();
}
while (nElementos < 1);

float[] dato = new float[nElementos]; // crear la matriz dato
int i = 0; // subíndice
float max, min; // valor máximo y valor mínimo
```

- A continuación leemos los valores que forman la lista. La entrada de datos finalizará cuando se tecleen todos los valores, o bien cuando se teclee un valor no numérico; por ejemplo, pulsar simplemente la tecla *Entrar*. Hagamos un breve repaso de la clase *Leer* que implementamos en el capítulo 5. Los métodos

dos de esta clase devuelven el número entero o decimal introducido a través del teclado. Ahora bien, cuando el valor tecleado no se corresponde con un número, los métodos implementados para leer un entero devuelven el valor **MIN_VALUE** (valor mínimo) y los implementados para leer un decimal, devuelven el valor **NaN** (no es un número).

```

for (i = 0; i < dato.length; i++)
{
    System.out.print("dato[" + i + "] = ");
    dato[i] = Leer.datoFloat();
    if (Float.isNaN(dato[i])) break; // salir del bucle
}

nElementos = i; // número de valores leídos

```

El código anterior establece un bucle para leer datos hasta completar la matriz. Si por cualquier circunstancia se decide terminar la entrada de datos antes que se complete la matriz, pulsando, por ejemplo, la tecla *Entrar*, el método *datoFloat* devolverá el valor **NaN**. Para detectar si esto ha ocurrido debemos utilizar el método *isNaN* de la clase *Float*. Se trata de un método *static* que devuelve **true** si su argumento se corresponde con un dato no numérico, y **false** en otro caso.

- Una vez leída la lista de valores, calculamos el máximo y el mínimo. Para ello suponemos inicialmente que el primer valor es el máximo y el mínimo (como si todos los valores fueran iguales). Después comparamos cada uno de estos dos valores con los restantes de la lista. El valor de la lista comparado pasará a ser el nuevo mayor si es más grande que el mayor actual y pasará a ser el nuevo menor si es más pequeño que el menor actual.

```

max = min = dato[0];
for (i = 0; i < nElementos; i++)
{
    if (dato[i] > max)
        max = dato[i];
    if (dato[i] < min)
        min = dato[i];
}

```

- Finalmente, escribimos el resultado.

```

System.out.println("\nValor máximo: " + max);
System.out.println("Valor mínimo: " + min);

```

El programa completo se muestra a continuación.

```

abes //Leer.class debe estar en la carpeta especificada por CLASSPATH
public class CValoresMaxMin
{
    // Obtener el máximo y el mínimo de un conjunto de valores
    public static void main(String[] args)
    {
        int nElementos; // número de elementos (valor no negativo)

        do
        {
            System.out.print("Número de valores que desea introducir: ");
            nElementos = Leer.datoInt();
        }
        while (nElementos < 1);

        float[] dato = new float[nElementos]; // crear la matriz dato
        int i = 0; // subíndice
        float max, min; // valor máximo y valor mínimo
        System.out.println("Introducir los valores.\n\n" +
                           "Para finalizar pulse [Entrar]");
        for (i = 0; i < dato.length; i++)
        {
            System.out.print("dato[" + i + "] = ");
            dato[i] = Leer.datoFloat();
            if (Float.isNaN(dato[i])) break;
        }
        nElementos = i; // número de valores leídos
        // Obtener los valores máximo y mínimo
        if (nElementos > 0)
        {
            max = min = dato[0];
            for (i = 0; i < nElementos; i++)
            {
                if (dato[i] > max) max = dato[i];
                if (dato[i] < min) min = dato[i];
            }
            // Escribir los resultados
            System.out.println("\nValor máximo: " + max);
            System.out.println("Valor mínimo: " + min);
        }
        else
            System.out.println("\nNo hay datos.");
    }
}

```

2. Escribir un programa que dé como resultado la frecuencia con la que aparece cada una de las parejas de letras adyacentes de un texto introducido por el teclado. No se hará diferencia entre mayúsculas y minúsculas. El resultado se presentará en forma de tabla, de la manera siguiente:

	a	b	c	d	e	f	...	z
a	0	4	0	2	1	0	...	1
b	8	0	0	0	3	1	...	0
c
d
e
f
.
.
z

Por ejemplo, la tabla anterior dice que la pareja de letras *ab* ha aparecido 4 veces. La tabla resultante contempla todas las parejas posibles de letras, desde la *aa* hasta la *zz*.

Las parejas de letras adyacentes de “*hola que tal*” son: *ho*, *ol*, *la*, *a blanco* no se contabiliza por estar el carácter espacio en blanco fuera del rango ‘a’ - ‘z’, *blanco q* no se contabiliza por la misma razón, *qu*, etc.

Para realizar este problema, en función de lo expuesto necesitamos una matriz de enteros de dos dimensiones. Cada elemento actuará como contador de la pareja de letras correspondiente. Por lo tanto, todos los elementos de la matriz deben valer inicialmente cero.

```
int[][] tabla = new int['z'-'a'+1]['z'-'a'+1];
```

Para que la solución sea fácil, aplicaremos el concepto de matrices asociativas visto anteriormente en este mismo capítulo; es decir, la pareja de letras a contabilizar serán los índices del elemento de la matriz que actúa como contador de dicha pareja. Observe la tabla anterior y vea que el contador de la pareja *aa* es el elemento (0,0) de la supuesta matriz. Esto supone restar una constante de valor ‘a’ a los valores de los índices (*carant*, *car*) utilizados para acceder a un elemento. La variable *carant* contendrá el primer carácter de la pareja y *car* el otro carácter.

```
if ((carant>='a' && carant<='z') && (car>='a' && car<='z'))
    tabla[carant - 'a'][car - 'a']++;
```

El problema completo se muestra a continuación.

```

import java.io.*;
// Leer.class debe estar en la carpeta especificada por CLASSPATH
public class CFrecuencia
{
    // Tabla de frecuencias de letras adyacentes en un texto.
    public static void main(String[] args)
    {
        // Crear la matriz tabla con 'z'-'a'+1 por 'z'-'a'+1 elementos.
        // Java inicia los elementos de la matriz a cero.
        int[][] tabla = new int['z'-'a'+1]['z'-'a'+1];
        char f, c;           // subíndices
        char car;            // carácter actual
        char carant = ' ';  // carácter anterior
        final char eof = (char)-1;

        // Entrada de datos y cálculo de la tabla de frecuencias
        System.out.println("Introducir un texto.");
        System.out.println("Para finalizar pulsar [Ctrl][z]\n");
        try
        {
            // Leer el siguiente carácter del texto
            while ((car = (char)System.in.read()) != eof)
            {
                // Convertir el carácter a minúsculas si procede
                if (car >= 'A' && car <= 'Z') car += ('a' - 'A');
                // Si el carácter leído está entre la 'a' y la 'z'
                // incrementar el contador correspondiente
                if ((carant>='a' && carant<='z') && (car>='a' && car<='z'))
                    tabla[carant - 'a'][car - 'a']++;
                carant = car;
            }
        }
        catch (IOException ignorada) {}
        // Mostrar la tabla de frecuencias
        System.out.println("\n");
        // Visualizar una cabecera "a b c ... "
        System.out.print(" ");
        for (c = 'a'; c <= 'z'; c++)
            System.out.print(" " + c);
        System.out.println();
        // Visualizar la tabla de frecuencias
        for (f = 'a'; f <= 'z'; f++)
        {
            System.out.print(f);
            for (c = 'a'; c <= 'z'; c++)
                System.out.print(" " + tabla[f - 'a'][c - 'a']);
            System.out.println();
        }
    }
}

```

Analizando el código que muestra la tabla de frecuencias, observamos un primer bucle **for** que visualiza la cabecera “a b c ...”; esta primera línea especifica el segundo carácter de la pareja de letras que se contabiliza; el primer carácter aparece a la izquierda de cada fila de la tabla. Después observamos dos bucles **for** anidados cuya función es escribir los valores de la matriz *tabla* por filas; nótese que antes de cada fila se escribe el carácter primero de las parejas de letras que se contabilizan en esa línea.

	a	b	c	d	e	f	...	z
a	0	4	0	2	1	0	...	1
b	8	0	0	0	3	1	...	0
c
d
e
f
.
.
z

EJERCICIOS PROPUESTOS

- Se desea realizar un histograma con los pesos de los alumnos de un determinado curso.

Peso	Número de alumnos
21	**
22	****
23	*****
24	*****
..	..

El número de asteriscos se corresponde con el número de alumnos del peso especificado.

Realizar un programa que lea los pesos e imprima el histograma correspondiente. Suponer que los pesos están comprendidos entre los valores 10 y 100 Kg. En el histograma sólo aparecerán los pesos que se corresponden con 1 o más alumnos.

- Realizar un programa que lea una cadena de *n* caracteres e imprima el resultado que se obtiene cada vez que se realice una rotación de un carácter a la derecha sobre dicha cadena. El proceso finalizará cuando se haya obtenido nuevamente la cadena de caracteres original. Por ejemplo,

HOLA AHOL LAHO OLAH HOLA

3. Realizar un programa que lea una cadena de caracteres y la almacene en una matriz. A continuación, utilizando un método, deberá convertir los caracteres escritos en mayúsculas a minúsculas. Finalmente imprimirá el resultado.
4. La mediana de una lista de n números se define como el valor que es menor o igual que los valores correspondientes a la mitad de los números, y mayor o igual que los valores correspondientes a la otra mitad. Por ejemplo, la mediana de:

16 12 99 95 18 87 10

es 18, porque este valor es menor que 99, 95 y 87 (mitad de los números) y mayor que 16, 12 y 10 (otra mitad).

Realizar un programa que lea un número impar de valores y dé como resultado la mediana. La entrada de valores finalizará cuando se detecte la marca de fin de fichero,

5. Escribir un programa que utilice un método para leer una línea de la entrada y dé como resultado la línea leída y su longitud o número de caracteres.
6. Analice el programa que se muestra a continuación e indique el significado que tiene el resultado que se obtiene.

```
import java.io.*;
public class Test
{
    public static void Visualizar(byte car)
    {
        int i = 0, bit;
        for (i = 7; i >= 0; i--)
        {
            bit = ((car & (1 << i))!= 0) ? 1 : 0;
            System.out.print(bit);
        }
        System.out.println();
    }

    public static byte HaceAlgo(byte car)
    {
        return (byte)((((car & 0x01) << 7) | ((car & 0x02) << 5) |
                      ((car & 0x04) << 3) | ((car & 0x08) << 1) |
                      ((car & 0x10) >> 1) | ((car & 0x20) >> 3) |
                      ((car & 0x40) >> 5) | ((car & 0x80) >> 7));
    }
}
```

```

public static void main(String[] args)
{
    byte car;
    try
    {
        System.out.print("Introduce un carácter ASCII: ");
        car = (byte)System.in.read();
        Visualizar(car);
        System.out.println("\nCarácter resultante:");
        car = HaceAlgo(car);
        Visualizar(car);
    }
    catch (IOException ignorar){}
}
}

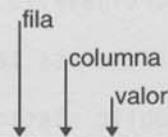
```

8. Para almacenar una matriz bidimensional que generalmente tiene muchos elementos nulos (matriz *sparse*) se puede utilizar una matriz unidimensional en la que sólo se guardarán los elementos no nulos precedidos por sus índices, *fila* y *columna*, lo que redundará en un aprovechamiento de espacio. Por ejemplo, la matriz:

6	0	0	0	4
0	5	0	0	2
2	0	0	0	0
0	0	7	0	0
0	0	0	8	0

se guardará en una matriz unidimensional así:

0	0	6	0	4	4	1	1	5	1	4	2	2	0	2	3	2	7	4	3	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



Se pide:

- a) Escribir un método que lea una matriz bidimensional por filas y la almacene en una matriz *m* unidimensional. El prototipo de este método será:

```
int CrearMatrizUni(int[] m, int fi, int co);
```

Los parámetros *fi* y *co* se corresponden con el número de filas y de columnas de la supuesta matriz bidimensional.

- b) Escribir un método que permita representar en pantalla la matriz bidimensional por filas y columnas. El prototipo de este método será:

```
int Visualizar(int f, int c, int[] m);
```

Los parámetros f y c se corresponden con la fila y la columna del elemento que se visualiza. El valor del elemento que se visualiza se obtiene, lógicamente de la matriz unidimensional creada en el apartado a , así: buscamos por los índices f y c ; si se encuentran, el método *Visualizar* devuelve el valor almacenado justamente a continuación; si no se encuentran, entonces devuelve un cero.

Escribir un programa que, utilizando el método *CrearMatrizUni*, cree una matriz unidimensional a partir de una supuesta matriz *sparse* bidimensional y a continuación, utilizando el método *Visualizar*, muestre en pantalla la matriz bidimensional.

CAPÍTULO 8

© 2008 F.J.Ceballos/RA-MA

MÉTODOS

En los capítulos anteriores aprendimos lo que es un programa, cómo escribirlo y qué hacer para que el ordenador lo ejecute y muestre los resultados perseguidos; adquirimos conocimientos generales acerca de la programación orientada a objetos; aprendimos acerca de los elementos que aporta Java; analizamos cómo era la estructura de una programa Java; aprendimos a leer datos desde el teclado y a visualizar resultados sobre el monitor; estudiamos las estructuras de control; y aprendimos a trabajar con matrices.

En este capítulo, utilizando los conocimientos adquiridos hasta ahora, vamos a centrarnos en cuestiones más específicas como pasar argumentos a métodos, escribir métodos que devuelvan matrices, copiar matrices, pasar argumentos en la línea de órdenes, imprimir resultados con formato, clasificar los elementos de una matriz, o bien buscar un elemento en una matriz, entre otras cosas.

PASAR UNA MATRIZ COMO ARGUMENTO A UN MÉTODO

En el capítulo 4 se expuso cómo definir un método en una clase y se explicó cómo pasar argumentos a un método. Recuerde que los objetos pasados a los parámetros de un método son siempre referencias a dichos objetos, lo cual significa que cualquier modificación que se haga a esos objetos dentro del método afecta al objeto original, y las matrices son objetos. En cambio, las variables de un tipo primitivo se pasan por valor, lo cual significa que se pasa una copia, por lo que cualquier modificación que se haga a esas variables dentro del método no afecta a la variable original.

En más de una ocasión, trabajando con cadenas de caracteres hemos pasado como argumento una matriz. Sirva de ejemplo el método *MinusculasMayusculas* expuesto en el apartado “Trabajar con cadenas de caracteres” del capítulo ante-

rior. Algunos de los métodos de la biblioteca Java también tienen parámetros que son matrices de caracteres, por ejemplo **read**. Pero no hemos estudiado nada análogo en el caso de matrices numéricas, lo cual es lógico porque mientras una cadena de caracteres, por ejemplo “nombre”, es un objeto que manejamos habitualmente como tal, no sucede lo mismo con una matriz numérica, donde cualquier operación pasa por el acceso individual a sus elementos.

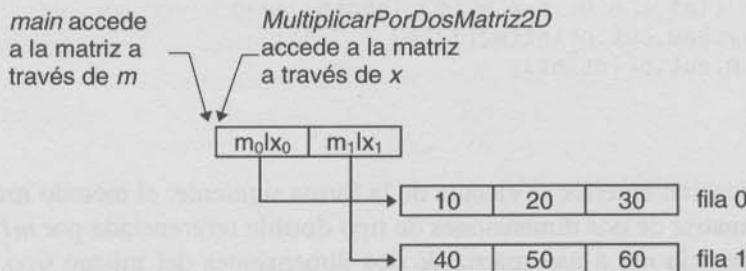
Para aclarar lo expuesto, el siguiente ejemplo implementa un método con un parámetro de tipo **double[][]**, que permite multiplicar por 2 los elementos de una matriz numérica de dos dimensiones pasada como argumento.

```
public class Test
{
    static void MultiplicarPorDosMatriz2D(double[][] x)
    {
        for (int f = 0; f < x.length; f++)
        {
            for (int c = 0; c < x[0].length; c++)
                x[f][c] *= 2;
        }
    }

    public static void main(String[] args)
    {
        double[][] m = {{10, 20, 30}, {40, 50, 60}};

        MultiplicarPorDosMatriz2D(m);
        // Visualizar la matriz por filas
        for (int f = 0; f < m.length; f++)
        {
            for (int c = 0; c < m[0].length; c++)
                System.out.print(m[f][c] + " ");
            System.out.println();
        }
    }
}
```

La aplicación anterior se ejecuta de la forma siguiente: el método **main** crea e inicia una matriz *m* de dos dimensiones de tipo **double**. Después invoca al método *MultiplicarPorDosMatriz2D* pasando como argumento la matriz *m*; esto implica que el método tenga un parámetro declarado así: *double[][] x*. Por ser *m* un objeto, el parámetro *x* recibe una referencia a la matriz *m*; esto es, *x* almacenará la posición de memoria de dónde se encuentra la matriz, no una copia de su contenido. Por lo tanto, ahora el método *MultiplicarPorDosMatriz2D* tiene acceso a la misma matriz que el método **main**. Gráficamente puede imaginárselo así:



¿Cuál es el resultado? Que cuando el método `main` visualice los elementos de la matriz `m`, éstos aparecerán con los cambios introducidos por el método `MultiplicarPorDosMatriz2D`. Esto es, ambos métodos trabajan sobre la misma matriz.

MATRIZ COMO VALOR RETORNADO POR UN MÉTODO

Según vimos en el capítulo 4, un método puede retornar un valor de cualquier tipo primitivo, o bien una referencia a cualquier clase de objetos. Por lo tanto, en el caso de que un método devuelva una matriz, lo que realmente devuelve es una referencia a la matriz. Aclaremos esto con un ejemplo.

La aplicación siguiente implementa un método que tiene un parámetro de tipo `double[][]` y permite copiar una matriz numérica bidimensional pasada como argumento, en otra matriz. El método devuelve como resultado la copia realizada.

```
public class Test
{
    static double[][] CopiarMatriz2D(double[][] x)
    {
        double[][] z = new double[x.length][x[0].length];

        for (int f = 0; f < x.length; f++)
            for (int c = 0; c < x[0].length; c++)
                z[f][c] = x[f][c];
        return z;
    }

    public static void main(String[] args)
    {
        double[][] m1 = {{10, 20, 30}, {40, 50, 60}};

        // Copiar una matriz utilizando un método
        double[][] m2 = CopiarMatriz2D(m1);
        m1[0][0] = 77; // modificar un elemento de la matriz original
        // Visualizar la matriz m2
        for (int f = 0; f < m2.length; f++)
```

```

        for (int c = 0; c < m2[0].length; c++)
            System.out.print(m2[f][c] + " ");
        System.out.println();
    }
}

```

se crea una
síntesis de
m al enviar

print | return

La aplicación anterior se ejecuta de la forma siguiente: el método **main** crea e inicia una matriz de dos dimensiones de tipo **double** referenciada por *m1*, y declara una referencia *m2* a una matriz de dos dimensiones del mismo tipo. Después invoca al método *CopiarMatriz2D* pasando como argumento la matriz *m1*. Esto implica que ese método tenga un parámetro declarado así: *double[][] x*, para que pueda recibir una referencia a la matriz *m1*. A continuación, *CopiarMatriz2D* crea una matriz *z* de las mismas características que *x*, copia los elementos de *x* en *z* y devuelve como resultado *z*. Finalmente, la referencia devuelta por *CopiarMatriz2D* es almacenada por el método **main** en *m2*, que como comprobación visualiza esa matriz.

Evidentemente, el método *CopiarMatriz2D* podría haberse diseñado según el siguiente prototipo, trabajo que se deja como ejercicio para el lector.

```
static void CopiarMatriz2D(double[][] destino, double[][] origen)
```

Otra forma de realizar una copia de una matriz es utilizando el método **clone** expuesto anteriormente. Quizás esta forma de proceder resulte más difícil de comprender cuando se manipulan matrices multidimensionales. Por eso es recomendable volver a analizar detenidamente la figura expuesta en el apartado “Matrices numéricas multidimensionales” del capítulo anterior. Si llega a la conclusión de que una matriz de dimensiones *f*×*c* es una matriz de una dimensión de *f* elementos que son referencias a otras tantas matrices de una dimensión de *c* elementos de un tipo especificado, le será fácil entender el código mostrado a continuación, el cual copia una matriz de dos dimensiones referenciada por *m1* en otra matriz referenciada por *m2*:

```

double[][] m1 = {{10, 20, 30}, {40, 50, 60}};
double[][] m2 = (double[][])m1.clone();
for (int f = 0; f < m1.length; f++)
    m2[f] = (int[])m1[f].clone();

```

Otra forma más de realizar una copia de una matriz es utilizando el método **arraycopy** de la clase **System**. Se trata de un método público y estático cuya sintaxis es la siguiente:

```
void arraycopy(Object origen, int posición_origen,
               Object destino, int posición_destino,
               int longitud);
```

donde *origen* es la matriz origen de los datos, *posición_origen* el índice de inicio en la matriz origen, *destino* es la matriz destino de los datos, *posición_destino* el índice de inicio en la matriz destino y *longitud* es el número de elementos que se desean copiar.

Por ejemplo, el código mostrado a continuación, copia una matriz de dos dimensiones referenciada por *m1* en otra matriz referenciada por *m2*:

```
int[][] m1 = {{10, 20, 30}, {40, 50, 60}};
int[][] m2 = new int[m1.length][m1[0].length];
System.arraycopy(m1, 0, m2, 0, m1.length);
```

REFERENCIA A UN TIPO PRIMITIVO

Cuando un método Java invoca a otro método y le pasa un argumento de un tipo primitivo, pasa una copia de ese argumento. Por ejemplo:

```
public static void Incrementar10(int param)
{
    param += 10;
}
public static void main(String[] args)
{
    int arg = 1234;
    Incrementar10(arg);
    System.out.println(arg);
}
```

La línea sombreada del ejemplo anterior invoca al método *Incrementar10* y copia el valor del argumento *arg* en el parámetro *param* del método. Esto significa que el argumento ha sido pasado por valor. Por lo tanto, cualquier modificación que haga el método sobre *param* no afectará a la variable original. Según lo expuesto el método *main* mostrará el resultado 1234, valor original de *arg*.

¿Qué hay que hacer para que un método pueda modificar el valor original del argumento que se le pasa? Pasar dicho argumento por referencia.

Según lo estudiado hasta ahora, cuando se pasa un argumento que es un objeto, Java no hace una copia del objeto sobre el parámetro correspondiente del método, sino que informa al método acerca del lugar de la memoria donde está ese objeto para que puede acceder al mismo, lo que se denomina pasar un argumento por referencia. Esto es, lo que se copia en el parámetro del método es una referencia al objeto.

También hemos estudiado que un valor de un tipo primitivo puede ser encapsulado en un objeto. Por ejemplo un valor de tipo **int** puede ser encapsulado en un objeto de la clase **Integer**. Entonces, si los objetos son pasados por referencia ¿puede un método modificar el valor original del argumento que se le pasa cuando éste es un objeto? Analicemos el siguiente ejemplo:

```
public static void Incrementar10(Integer param)
{
    int valor = param.intValue();
    valor += 10;
    param = new Integer(valor);
}

public static void main(String[] args)
{
    Integer arg = new Integer(1234);
    Incrementar10(arg);
    System.out.println(arg.intValue());
}
```

El método **main** del ejemplo anterior crea un objeto **Integer** con el valor 1234 y almacena una referencia a ese objeto en la variable *arg*. Cuando **main** invoca a *Incrementar10*, le pasa una referencia que este método almacena en su parámetro *param*. El método *Incrementar10* obtiene el valor entero del objeto, lo incrementa en 10 y crea un nuevo objeto **Integer** con el resultado, almacenando la referencia al mismo en *param*. Esto sobreescribe la referencia anterior que almacenaba *param*, pero lógicamente no afecta a la variable *arg*, así que el método **main** mostrará el valor 1234 original.

¿Qué ha sucedido? Que el método *Incrementar10* no sólo no modificó la estructura de datos del objeto referenciado por *param* (ya que *valor* es una variable local que no pertenece al objeto), sino que asignó a *param* un nuevo objeto. Para poder modificar el objeto pasado por referencia, la clase **Integer** debería proporcionar, según muestra el ejemplo siguiente, un método análogo a *AsignarValor*:

```
public static void Incrementar10(Integer param)
{
    int valor = param.intValue();
    valor += 10;
    param.AsignarValor(valor);
}
```

Como las clases que encapsulan los tipos primitivos no tienen métodos análogos al descrito, esta forma de pasar un valor de un tipo primitivo por referencia con la intención de que sea modificado, no sirve. ¿Cómo dar solución al problema planteado? Según se ha expuesto anteriormente, una matriz es un objeto, lo cual

significa que cuando se pase como argumento a un método, será pasado por referencia. Por lo tanto, los cambios que haga este método sobre los elementos de esa matriz afectarán a la original. Veamos el siguiente ejemplo:

```
public static void Incrementar10(int[] param)
{
    param[0] += 10;
}

public static void main(String[] args)
{
    int[] arg = { 1234 };
    Incrementar10(arg);
    System.out.println(arg[0]);
}
```

En el ejemplo anterior, el método **main** define un valor de tipo **int** mediante una matriz de un solo elemento. Después invoca al método *Incrementar10* pasándole como argumento esa matriz, lo que supone copiar la referencia *arg* en el parámetro *param*. Ahora *param* hace referencia a la misma matriz que *arg*. Por lo tanto, todos los cambios realizados por el método afectarán a la matriz original. Como consecuencia, el resultado mostrado por **main** será ahora 1244.

ARGUMENTOS EN LA LÍNEA DE ÓRDENES

Muchas veces, cuando invocamos a un programa desde el sistema operativo, necesitamos escribir uno o más argumentos a continuación del nombre del programa, separados por un espacio en blanco. Por ejemplo, piense en la orden *ls -l* del sistema operativo UNIX o en la orden *dir /p* del sistema operativo MS-DOS. Tanto *ls* como *dir* son programas; *-l* y */p* son opciones o argumentos en la línea de órdenes que pasamos al programa para que tenga un comportamiento diferente al que tiene de forma predeterminada; es decir, cuando no se pasan argumentos.

De la misma forma, nosotros podemos construir aplicaciones Java que admitan argumentos a través de la línea de órdenes ¿Qué método recibirá esos argumentos? El método **main**, ya que este método es el punto de entrada a la aplicación y también el punto de salida. Su definición, una vez más, es como se muestra a continuación:

```
public static void main(String[] args)
{
    // Cuerpo del método
}
```

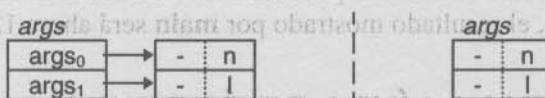
Como se puede observar, el método **main** tiene un argumento *args* que es una matriz unidimensional de tipo **String**. El nombre *args* puede ser cualquier otro. Esta matriz almacenará los argumentos pasados en la línea de órdenes cuando se invoque a la aplicación para su ejecución de la forma que se observa a continuación. Observe que cada argumento está separado por un espacio.

```
java MiAplicación argumento1 argumento2 ...
```

Cada elemento de la matriz *args* referencia a un argumento, de manera que *args[0]* contiene el primer argumento de la línea de órdenes, *args[1]* el segundo, etc. Por ejemplo, supongamos que tenemos una aplicación Java denominada *Test* que acepta los argumentos *-n* y *-l*. Entonces, podríamos invocar a esta aplicación escribiendo en la línea de órdenes del sistema operativo la siguiente orden:

```
java Test -n -l
```

Esto hace que automáticamente la matriz *args* de objetos **String** se cree para contener dos objetos **String**: uno con el primer argumento y otro con el segundo. Puede imaginárla de cualquiera de las dos formas siguientes:



Para clarificar lo expuesto vamos a realizar una aplicación que simplemente visualice los valores de los argumentos que se la han pasado en la línea de órdenes. Esto nos dará una idea de cómo acceder desde un programa a esos argumentos. Supongamos que la aplicación se denomina *Test* y que sólo admite los argumentos *-n*, *-k* y *-l*. Esto quiere decir que podremos especificar de cero a tres argumentos. Los argumentos repetidos y no válidos se desecharán. Por ejemplo, la siguiente línea invoca a la aplicación *Test* pasándole los argumentos *-n* y *-l*:

```
java Test -n -l
```

El código de la aplicación propuesta, se muestra a continuación.

```
public class Test {
    public static void main(String[] args) {
        // Código común a todos los casos
        System.out.println("Argumentos: ");
        if (args.length == 0)
            // Escriba aquí el código que sólo se debe ejecutar cuando
            // no se pasan argumentos
    }
}
```

MÉTODOS RECURSIVOS

```
System.out.println("    ninguno");
}
else
{
    boolean argumento_k = false, argumento_l = false,
    boolean argumento_n = false;

    // ¿Qué argumentos se han pasado?
    for (int i = 0; i < args.length; i++)
    {
        if (args[i].compareTo("-k") == 0) argumento_k = true;
        if (args[i].compareTo("-l") == 0) argumento_l = true;
        if (args[i].compareTo("-n") == 0) argumento_n = true;
    }

    if (argumento_k) // si se pasó el argumento -k:
    {
        // Escriba aquí el código que sólo se debe ejecutar cuando
        // se pasa el argumento -k
        System.out.println("    -k");
    }

    if (argumento_l) // si se pasó el argumento -l:
    {
        // Escriba aquí el código que sólo se debe ejecutar cuando
        // se pasa el argumento -l
        System.out.println("    -l");
    }

    if (argumento_n) // si se pasó el argumento -n:
    {
        // Escriba aquí el código que sólo se debe ejecutar cuando
        // se pasa el argumento -n
        System.out.println("    -n");
    }
}

// Código común a todos los casos
```

Al ejecutar este programa, invocándolo como se ha indicado anteriormente, se obtendrá el siguiente resultado:

Argumentos: + oracion + " ab_16107267_1367716769q.duo.midi

MÉTODOS RECURSIVOS

Se dice que un método es recursivo, si se llama a sí mismo. El compilador Java permite cualquier número de llamadas recursivas a un método. Cada vez que el método es llamado, sus parámetros y sus variables locales son iniciadas.

¿Cuándo es eficaz escribir un método recursivo? La respuesta es sencilla, cuando el proceso a programar sea por definición recursivo. Por ejemplo, el cálculo del factorial de un número, $n! = n(n-1)!$, es por definición un proceso recursivo que se enuncia así: $\text{factorial}(n) = n * \text{factorial}(n-1)$

Por lo tanto, la forma idónea de programar este problema es implementando un método recursivo. Como ejemplo, a continuación se muestra un programa que visualiza el factorial de un número. Para ello, se ha escrito un método *factorial* que recibe como parámetro un número entero positivo y devuelve como resultado el factorial de dicho número.

```
public class Test
{
    // Cálculo del factorial de un número
    public static long factorial(int n)
    {
        if (n == 0)
            return 1;
        else
            return n * factorial(n-1);
    }

    public static void main(String[] args)
    {
        int numero;
        long fac;

        do
        {
            System.out.print("¿Número? ");
            numero = Leer.datoInt();
        }
        while (numero < 0 || numero > 25);

        fac = factorial(numero);
        System.out.println("\nEl factorial de " + numero + " es: " + fac);
    }
}
```

En la tabla siguiente se ve el proceso seguido por el método *factorial*, durante su ejecución para $n = 4$.

<i>Nivel de recursión</i>	<i>Proceso de ida</i>	<i>Proceso de vuelta</i>
0	factorial(4)	24
1	4 * factorial(3)	4 * 6
2	3 * factorial(2)	3 * 2
3	2 * factorial(1)	2 * 1
4	1 * factorial(0)	1 * 1
	factorial(0)	1

Cada llamada al método factorial aumenta en una unidad el nivel de recursión. Cuando se llega a $n = 0$, se obtiene como resultado el valor 1 y se inicia la vuelta hacia el punto de partida, reduciendo el nivel de recursión en una unidad cada vez.

Los algoritmos recursivos son particularmente apropiados cuando el problema a resolver o los datos a tratar se definen en forma recursiva. Sin embargo, el uso de la recursión debe evitarse cuando haya una solución obvia por iteración.

En aplicaciones prácticas es imperativo demostrar que el nivel máximo de recursión es, no sólo finito, sino realmente pequeño. La razón es que, por cada ejecución recursiva del método, se necesita cierta cantidad de memoria para almacenar las variables locales y el estado en curso del proceso de cálculo con el fin de recuperar dichos datos cuando se acabe una ejecución y haya que reanudar la anterior.

VISUALIZAR DATOS CON FORMATO

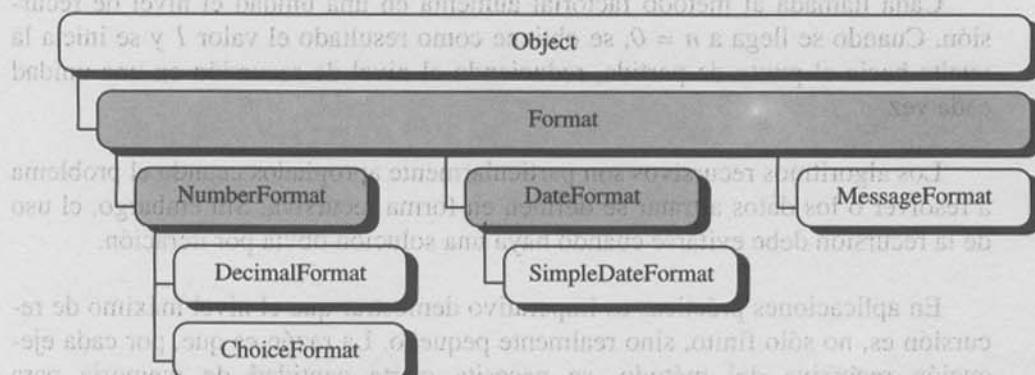
Los resultados producidos por las aplicaciones que hemos realizado hasta ahora han sido mostrados sin aplicar ningún tipo de formato. Pero quizás en alguna ocasión necesitemos expresar una cantidad:

- incluyendo la coma de los decimales y el punto de los miles,
- con un número determinado de dígitos enteros completando con ceros por la izquierda si fuera necesario,
- con un número determinado de decimales y ajustada a la derecha,
- o bien una serie de cantidades decimales, una debajo de otra, ajustadas por la coma.

Si en lugar de cantidades hablamos de fechas, también podríamos requerir diferentes modos de presentación.

Para controlar los distintos formatos, Java proporciona un conjunto de clases en el paquete **java.text**. La figura siguiente muestra estas clases. Los rectángulos sombreados son clases abstractas.

La clase **Format** es una clase base abstracta para dar formato a números, fechas/horas y mensajes. De esta clase se derivan tres subclases especializadas en cada una de las tareas mencionadas: **NumberFormat**, **DateFormat** y **MessageFormat**.



La clase **NumberFormat** es la clase base abstracta para todos los formatos numéricos. La clase **DateFormat** es también una clase abstracta para los formatos de fechas y horas. Pero las clases que son particularmente útiles y que estudiamos a continuación son: **DecimalFormat**, **SimpleDateFormat** y **MessageFormat**.

Dar formato a números

Para dar formato a un número, primero hay que crear un objeto formateador basado en un formato específico, y luego utilizar su método **format** para convertir el número en una cadena construida a partir del formato elegido. Los símbolos que se pueden utilizar para especificar un determinado formato son:

Símbolo	Significado
0	Representa un dígito cualquiera, incluyendo los ceros no significativos.
#	Representa un dígito cualquiera, excepto los ceros no significativos.
.	Representa el separador decimal.
,	Representa el separador de los miles.
E	Formato científico. E, separa la mantisa y el exponente.
:	Actúa como separador cuando se especifican varios formatos.
-	Signo negativo de forma predeterminada.

Multiplicar por 100 y mostrar el símbolo %. Representa el símbolo monetario. Cualquier carácter puede ser utilizado como prefijo o como sufijo. Por ejemplo \$ o '\u0024'.

El siguiente ejemplo crea un objeto *formato* que permitirá obtener números formateados con dos decimales, si los hay, y con el punto de los miles.

```
DecimalFormat formato = new DecimalFormat("###,###.##");
String salida = formato.format(dato);
```

Si el número de dígitos correspondiente a la parte entera excede el número de posiciones especificado para la misma, el formato se extiende en lo necesario. Si el número de dígitos decimales excede el número de posiciones especificado para los mismos, la parte decimal se trunca redondeando el resultado.

Se puede utilizar también un objeto formateador basado en la localidad actual. Por ejemplo, las siguientes líneas de código darán lugar a números formateados así: 123.456,00 Pts.

```
NumberFormat formato = NumberFormat.getCurrencyInstance();
String salida = formato.format(dato);
```

El método **getCurrencyInstance** devuelve el formato monetario de la localidad actual cuando no se especifica una, o el de la especificada. Por ejemplo:

```
Locale en_US = new Locale("en", "US");
NumberFormat formato = NumberFormat.getCurrencyInstance(en_US);
```

donde ("en", "US") significa inglés de Estados Unidos. Otros ejemplos de localidades son: ("en", "GB") que significa inglés del Reino Unido; ("es", "ES") que significa español de España; ("fr", "FR") que significa francés de Francia; ("de", "DE") que significa alemán de Alemania; etc. A continuación se explica la clase **Locale**.

Otros métodos de interés son **getNumberInstance** que devuelve el formato numérico predeterminado que se emplea en la localidad que se especifique, o bien en la actual si no se especifica ninguna localidad; y **getPercentInstance** que devuelve el formato para especificar un valor en tanto por ciento.

Localidad

Una localidad no es un idioma, ya que un mismo idioma se puede hablar en varios países. Cuando escriba un programa internacional tendrá que definir la localidad

actual y el conjunto de localidades que soportará el programa. Las localidades son definidas en Java por la clase **Locale** incluida en el paquete **java.util**. Un objeto de la clase **Locale** es simplemente un identificador para una localidad específica. Por ejemplo, el siguiente código crea dos objetos, *país[0]* y *país[1]*, uno para el español de España y otro para el inglés de Estados Unidos.

```
Locale[] país =
{
    new Locale("es", "ES"),
    new Locale("en", "US"),
};
```

Utilizando los métodos comentados en el apartado anterior, se puede obtener el formato predeterminado para cualquiera de estos países. Por ejemplo:

```
DecimalFormat df =
    (DecimalFormat)DecimalFormat.getNumberInstance(país[i]);
```

Alineación

Los valores numéricos que escribimos, con o sin formato, quedan alineados automáticamente a la izquierda. Para alinear a la derecha una serie de valores numéricos formateados, además del objeto formateador, hay que crear un objeto de la clase **FieldPosition** basado en la posición utilizada para realizar la alineación. Ésta puede ser: *INTEGER_FIELD*, alineación por el último dígito entero (por la coma decimal), o bien *FRACTION_FIELD*, alineación por el último dígito decimal (*INTEGER_FIELD* y *FRACTION_FIELD* son dos constantes pertenecientes a la clase **NumberFormat**). Por ejemplo:

```
String patrón = new String("###.##0.00");
DecimalFormat formato = new DecimalFormat(patrón);
FieldPosition fp = new FieldPosition(NumberFormat.FRACTION_FIELD);
```

Para utilizar el objeto **FieldPosition** definido, el método **format** invocado a través del objeto formateador debe tener tres parámetros: el valor numérico a formatear, un objeto **StringBuffer** donde se almacenará el número formateado y el objeto **FieldPosition**. Para realizar la alineación habrá que añadir al principio del objeto **StringBuffer** un número de espacios en blanco igual al *espacio de impresión deseado* menos el valor devuelto por **getEndIndex**.

¿Qué devuelve **getEndIndex**? Si el objeto **FieldPosition** se creó basado en la constante *INTEGER_FIELD*, el método **getEndIndex** devolverá el número de dígitos enteros del *dato* a formatear, y si se creó basado en la constante *FRACTION_FIELD*, el número total de dígitos (enteros y decimales). Para aclarar lo expuesto observe el siguiente ejemplo, continuación del anterior:

```

StringBuffer salida = new StringBuffer();
formato.format(dato, salida, fp);
for (int i = 0; i < (patrón.length() - fp.getEndIndex()); i++)
    salida.insert(0, ' ');

```

Una vez estudiado cómo dar formato a un número, el objetivo siguiente es escribir una clase *Obtener* que proporcione varios métodos que permitan obtener cualquier número en alguno de los formatos que consideremos más comunes. Posteriormente, podremos utilizar esta clase como soporte en otras aplicaciones.

Clase para formatos numéricos

Empleando los conocimientos expuestos hasta ahora podemos, como ejercicio, implementar una clase que incluya algunos métodos que después podamos utilizar para formatear números. Estos métodos los definiremos **static** para que puedan ser invocados sin necesidad de crear un objeto de la clase. La clase la denominaremos *Obtener* y sus métodos será los siguientes:

- *FormatoLocal*. Devuelve la cadena resultante de formatear un número utilizando el formato monetario predefinido en el país actual.
- *FormatoPer*. Devuelve la cadena resultante de formatear un número con un formato personalizado. Los símbolos de puntuación utilizados son los correspondientes al país actual.
- *AlinDer*. Realiza la misma operación que *FormatoPer* y además, alinea los números a la derecha del patrón utilizado.
- *FormatoPaís*. Realiza la misma operación que *FormatoPer* pero utilizando los símbolos de puntuación del país especificado.

```

import java.util.*;
import java.text.*;
public class Obtener
{
    static public String FormatoLocal(double dato)
    {
        NumberFormat formato = NumberFormat.getCurrencyInstance();
        String salida = formato.format(dato);
        return salida;
    }

    static public String FormatoPer(String patrón, double dato)
    {
        DecimalFormat formato = new DecimalFormat(patrón);
        String salida = formato.format(dato);
        return salida;
    }
}

```

```

    static public StringBuffer AlinDer(String patrón, double dato)
    {
        FieldPosition fp = new FieldPosition(NumberFormat.FRACTION_FIELD);
        DecimalFormat formato = new DecimalFormat(patrón);
        StringBuffer salida = new StringBuffer();
        formato.format(dato, salida, fp);
        for (int i = 0; i < (patrón.length() - fp.getEndIndex()); i++)
            salida.insert(0, ' ');
        return salida;
    }
}

```

Clases para formateos numéricos

```

    static public String FormatoPaís(String patrón, double dato,
                                     Locale lugar)
    {
        DecimalFormat df =
            (DecimalFormat)DecimalFormat.getNumberInstance(lugar);
        df.applyPattern(patrón);
        String salida = df.format(dato);
        return salida;
    }
}

```

Puede, si lo desea, añadir esta clase a la carpeta especificada por la variable de entorno **CLASSPATH**. A continuación escribimos una aplicación que nos permita probar cada uno de los métodos de la clase anterior.

```

import java.io.*;
import java.util.*;
public class CDemoFormatoNum
{
    static public void main(String[] args)
    {
        PrintStream flujoS = System.out;
        flujoS.println(Obtener.FormatoLocal(123456));
        flujoS.println(Obtener.FormatoLocal(123456.789));
        flujoS.println(Obtener.FormatoLocal(123.45));
        flujoS.println();
        flujoS.println(Obtener.FormatoPer("###,###.##", 123456));
        flujoS.println(Obtener.FormatoPer("#####", 123456));
        flujoS.println(Obtener.FormatoPer("###.##", 123456.789));
        flujoS.println(Obtener.FormatoPer("000000.000", 123.45));
        flujoS.println(Obtener.FormatoPer("###,###.##", 12345.67));
        flujoS.println(Obtener.FormatoPer("###,###.##", 12.34));
        flujoS.println();
        String patrón = new String("###,###,##0.00");
        flujoS.println(Obtener.AlinDer(patrón, 1.234));
        flujoS.println(Obtener.AlinDer(patrón, 12.345));
        flujoS.println(Obtener.AlinDer(patrón, -123.456));
    }
}

```

```

flujoS.println(Obtener.AlinDer(patrón, 123.456));
flujoS.println(Obtener.AlinDer(patrón, 1234.567));
flujoS.println(Obtener.AlinDer(patrón, 12345.678));
flujoS.println(Obtener.AlinDer(patrón, -12345));
flujoS.println();
Locale[] país =
{
    new Locale("es", "ES"),
    new Locale("en", "US"),
};
for (int i = 0; i < país.length; i++)
    flujoS.println(Obtener.FormatoPaís("###,###.###", 123456.789,
                                        país[i]));
}
}

```

Como ejercicio, ejecute esta aplicación y analice los resultados.

Dar formato a fechas/horas

Para dar formato a una fecha/hora dada, primero hay que crear un objeto formateador de la clase **SimpleDateFormat** basado en un formato específico, y luego utilizar su método **format** para convertir la fecha/hora en una cadena construida a partir del formato elegido. Los símbolos que se pueden utilizar para especificar un determinado formato son:

Símbolo	Significado	Presentación	Ejemplo
y	año	numérica	2002
M	mes del año	numérica y alfabética	08 y agosto
d	día del mes	numérica	15
h	hora (1 a 12)	numérica	10
H	hora (0 a 23)	numérica	13
m	minutos	numérica	30
s	segundos	numérica	55
S	milisegundos	numérica	658
E	día de la semana	alfabética	jueves
D	día del año	numérica	227
F	día de la semana del mes	numérica	3 (3º X de agosto)
w	semana del año	numérica	24
W	semana del mes	numérica	2
a	marca am/pm	alfabética	PM
z	zona horaria	alfabética	GMT+02:00

En las presentaciones alfabéticas, 4 o más símbolos dan lugar a la forma completa (por ejemplo, *MMMM* da lugar al nombre del mes completo: *agosto*); y

menos de 4 da lugar a la forma abreviada o a la numérica (por ejemplo, *MMM* da lugar a una abreviatura, *ago*, *MM* a un número de dos dígitos, *08*, y *M* a un número de un dígito, *8*).

En las presentaciones numéricas un símbolo da lugar al mínimo número de dígitos. Por ejemplo si *yyyy* da lugar a *2002*, *y*, *yy*, o *yyy* dan lugar a *02*.

Cualquier otro carácter fuera de los rangos ['A'..'Z'] y ['a'..'z'] será tratado como un separador.

El siguiente ejemplo almacena en *salida* la fecha y la hora actuales según el formato especificado por *patrón*.

```
Date hoy = new Date();
String patrón = "EEEE dd-MMM-yyyy, HH:mm:ss";
SimpleDateFormat formato = new SimpleDateFormat(patrón);
String salida = formato.format(hoy);
```

Un objeto **Date**, construido sin argumentos, encapsula el tiempo en milisegundos transcurridos desde el 1 de enero de 1970.

Se puede utilizar también un objeto formateador basado en la localidad actual. En este caso dicho objeto será creado y devuelto por alguno de los métodos de la clase **DateFormat**. Por ejemplo, las siguientes líneas de código almacenarán en *sFecha* y *sHora* la fecha y la hora según el formato local predeterminado. Por ejemplo: *16-ago-02* y *21:06:34*.

```
Date hoy = new Date();
String sFecha, sHora;
DateFormat formato;

formato = DateFormat.getDateInstance();
sFecha = formato.format(hoy);
formato = DateFormat.getTimeInstance();
sHora = formato.format(hoy);
```

El método **getDateInstance** sin argumentos devuelve el formato utilizado para mostrar la fecha en la localidad actual, y el método **getTimeInstance** sin argumentos devuelve el formato utilizado en la localidad actual para mostrar la hora. Ambos métodos pueden ser invocados con un argumento que especifique el estilo con el que será formateada la fecha o la hora; por ejemplo:

```
DateFormat.getDateInstance(DateFormat.MEDIUM);
```

o bien con el estilo y la localidad; por ejemplo:

```
DateFormat.getDateInstance(DateFormat.DEFAULT, país);
```

donde *país* es un objeto **Locale** según se explicó anteriormente en el apartado “Localidad”.

Dar formato a mensajes

Para dar formato a un mensaje que se desea construir durante la ejecución, como por ejemplo “Fueron verificados 1234 ficheros de la unidad C: en 125 segundos”, hay que crear un objeto formateador de la clase **MessageFormat**. Esta clase proporciona un medio para construir mensajes con partes variables que serán reemplazados durante la ejecución. Por ejemplo:

```
Object[] argumentos = {new Long(1234), "C:", new Long(125)};

MessageFormat mensaje = new MessageFormat("Fueron verificados " +
    "{0} ficheros de la unidad {1} en {2} segundos");
System.out.println(mensaje.format(argumentos));
```

LA CLASE Arrays

La clase **Arrays** del paquete **java.util** contiene varios métodos **static** para manipular matrices. Estos métodos son: **binarySearch**, **equals**, **fill** y **sort**.

binarySearch

Este método permite buscar un valor en una matriz que esté ordenada ascendente utilizando el algoritmo de *búsqueda binaria*. Este algoritmo será explicado más adelante en otro capítulo. Ahora basta con saber que se trata de un algoritmo muy eficiente en cuanto a que el tiempo requerido para realizar una búsqueda es muy pequeño. La sintaxis expresada de forma genérica para utilizar este método es la siguiente:

```
int binarySearch(tipo[] m, tipo clave)
```

donde *m* representa la matriz, *clave* es el valor que se desea buscar del mismo tipo que los elementos de la matriz, y *tipo* es cualquier tipo de datos de los siguientes: **byte**, **char**, **short**, **int**, **long**, **float**, **double** y **Object**.

El valor devuelto es un entero correspondiente al índice del elemento que coincide con el valor buscado. Si el valor buscado no se encuentra, entonces el valor devuelto es: *-(punto de inserción) - 1*. El valor de *punto de inserción* es el

índice del elemento de la matriz donde debería encontrarse el valor buscado. La expresión “*-(punto de inserción) – 1*” garantiza que el índice devuelto será mayor o igual que cero sólo si el valor buscado es encontrado.

Como ejemplo, analice el siguiente código:

```
double[] a = {10,15,20,25,30,35,40,45,50,55};
int i;
i = Arrays.binarySearch(a, 25); // i = 3
i = Arrays.binarySearch(a, 27); // i = -5
i = Arrays.binarySearch(a, 5); // i = -1
i = Arrays.binarySearch(a, 60); // i = -11
```

equals

Este método permite verificar si dos matrices son iguales. Dos matrices se consideran iguales cuando ambas tienen el mismo número de elementos y en el mismo orden. Asimismo, dos matrices también son consideradas iguales si sus referencias valen **null**. La sintaxis para utilizar este método expresada de forma genérica es la siguiente:

```
boolean equals(tipo[] m1, tipo[] m2)
```

donde *m1* y *m2* son matrices del mismo tipo y *tipo* es cualquier tipo de datos de los siguientes: **boolean, byte, char, short, int, long, float, double y Object**.

El valor devuelto será **true** si ambas matrices son iguales y **false** en caso contrario.

Como ejemplo, puede probar los resultados que produce el siguiente código:

```
double[] a = {10,15,20,25,30,35,40,45,50,55};
double[] b = {10,15,20,25,30,35,40,45,50,55};
if (Arrays.equals(a, b))
    System.out.println("Son iguales");
else
    System.out.println("No son iguales");
```

fill

Este método permite asignar un valor a todos los elementos de una matriz, o bien a cada elemento de un rango especificado. La sintaxis expresada de forma genérica para utilizar este método es la siguiente:

```
void fill(tipo[] m, tipo valor)
void fill(tipo[] m, int desdeInd, int hastaInd, tipo valor)
```

donde *m* es la matriz y *valor* es el valor a asignar. Cuando sólo queramos asignar el valor a un rango de elementos, utilizaremos el segundo formato de **fill** donde *desdeInd* y *hastaInd* definen ese rango. *tipo* es cualquier tipo de datos de los siguientes: **boolean**, **byte**, **char**, **short**, **int**, **long**, **float**, **double** y **Object**.

Un ejemplo de cómo utilizar este método es el siguiente:

```
double[] a = {10,15,20,25,30,35,40,45,50,55};
Arrays.fill(a, 0); // poner los elementos de la matriz a cero
```

sort

Este método permite ordenar los elementos de una matriz en orden ascendente utilizando el algoritmo *quicksort*. Este algoritmo será explicado más adelante en otro capítulo. Ahora basta con saber que se trata de un algoritmo muy eficiente en cuanto a que el tiempo requerido para realizar la ordenación es mínimo. La sintaxis expresada de forma genérica para utilizar este método es la siguiente:

```
void sort(tipo[] m)
void sort(tipo[] m, int desdeInd, int hastaInd)
```

donde *m* es la matriz a ordenar. Cuando sólo queramos ordenar un rango de elementos, utilizaremos el segundo formato de **sort** donde *desdeInd* y *hastaInd* definen los límites de ese rango. *tipo* es cualquier tipo de datos de los siguientes: **byte**, **char**, **short**, **int**, **long**, **float**, **double** y **Object**.

Como ejemplo, puede probar los resultados que produce el siguiente código:

```
double[] a = {55,50,45,40,35,30,25,20,15,10};
Arrays.sort(a);
```

LA CLASE Object

La clase **Object** es la clase raíz de la jerarquía de clases de la biblioteca Java; pertenece al paquete **java.lang**. Asimismo, cualquier clase que implementemos en nuestras aplicaciones pasará a ser automáticamente una subclase de esta clase. Esto se traduce en que todos los métodos de **Object** son heredados por las clases de la biblioteca Java y por cualquier otra clase que incluyamos en un programa. Tres de estos métodos (**wait**, **notify** y **notifyAll**) soportan el control de hilos, por lo tanto posponemos su estudio a un capítulo posterior; otros métodos, como

getClass y **clone**, los hemos utilizado en capítulos anteriores; y otros, como **equals**, **toString** y **finalize** son expuestos a continuación.

boolean equals(Object obj)

El método **equals** de la clase **Object** retorna **true** si y sólo si las dos referencias comparadas señalan al mismo objeto; esto es, proporciona el mismo resultado que el operador “**==**”. Esto es así porque la intención es proporcionar un método que pueda ser sobreescrito en cada una de las subclases de **Object** que requieran una funcionalidad más específica. Por ejemplo, consideremos el siguiente código que define dos referencias a otros dos objetos de la clase **String**:

```
public class Test
{
    public static void main(String[] args)
    {
        String str1 = new String("abc");
        String str2 = new String("abc");
        // Comparar referencias
        if (str1 == str2)
            System.out.println("Las referencias son al mismo objeto");
        else
            System.out.println("Las referencias son a objetos diferentes");
        // Comparar contenidos
        if (str1.equals(str2))
            System.out.println("Mismo contenido");
        else
            System.out.println("Diferente contenido");
    }
}
```

La expresión **str1 == str2** será **true** si la referencia **str1** es igual a la referencia **str2**; esto es, si ambas variables contienen idénticos valores, los cuales se corresponderán con la posición de memoria donde se localice un objeto.

En cambio, la expresión **str1.equals(str2)** compara el contenido de los objetos; en este caso verifica si ambas cadenas contienen los mismos caracteres. Esto es así, porque el método **equals** de **Object** ha sido sobreescrito en la clase **String** para que haga esta tarea más específica. Todas las subclases de **Object** deberían sobreescribir el método **equals** para que realicen una comparación que sea útil.

Cuando se ejecute la aplicación anterior se obtendrá el siguiente resultado:

```
Las referencias son a objetos diferentes
Mismo contenido
```

String **toString()**

El método **toString** de la clase **Object** retorna: un **String** que almacena el nombre de la clase del objeto que recibe el mensaje **toString**, el símbolo '@', y la representación hexadecimal del código *hash* del objeto. Esto es, la cadena de caracteres sería equivalente a la proporcionada por la siguiente expresión:

```
obj.getClass().getName()+'@'+Integer.toHexString(obj.hashCode())
```

El ejemplo siguiente permite verificar lo expuesto.

```
public class Test
{
    public static void main(String[] args)
    {
        Test obj = new Test();
        String s;
        s = obj.getClass().getName()+'@'+Integer.toHexString(obj.hashCode());
        System.out.println(s);
        System.out.println(obj.toString()); // mismo resultado que s
    }
}
```

Cuando se ejecute el ejemplo anterior la línea sombreada dará lugar al siguiente resultado:

```
Test@73bf4fe1
```

Con respecto al método **toString** diremos lo mismo que para **equals**; esto es, todas las subclases de **Object** deberían sobreescribir el método **toString** para que proporcione una información que sea útil. Por ejemplo, la clase **String** sobreescribe este método para que retorne el propio objeto **String** que recibe el mensaje **toString**.

void **finalize()**

Este método es invocado por el *recolector de basura* cuando Java determina que no hay más referencias a un objeto.

El método **finalize** de la clase **Object** no ejecuta ninguna acción en especial; simplemente retorna normalmente. Las subclases de **Object** deberán sobreescribir la definición de este método sólo cuando necesiten ejecutar alguna operación de finalización especial.

MÁS SOBRE REFERENCIAS Y OBJETOS String

En el capítulo anterior hicimos una breve exposición acerca de cómo crear un objeto **String** a partir de un literal o a partir de otro **String**. Por ejemplo:

```
String str = "abc";
```

Este ejemplo crea un objeto **String** con el contenido “abc”. Dicho proceso puede realizarse también así:

```
String str = new String("abc");
```

No obstante, es importante saber cuál es el comportamiento de Java ante las dos formas expuestas de crear un objeto **String**.

Cada literal de caracteres es representado internamente por un objeto **String**. Así mismo, Java mantiene un área de memoria destinada a almacenar tales objetos **String**. Entonces, cuando Java compila un literal (en el ejemplo “abc”) añade el objeto **String** correspondiente, a dicho área de memoria; posteriormente, si aparece el mismo literal en cualquier otra parte del código de la clase, el compilador no añade un nuevo objeto, sino que utiliza el que hay. Esta forma de proceder ahorra memoria y no causa problemas porque los **String** son objetos no modificables; por lo tanto, no hay posibilidad de que una parte del código pueda modificar un objeto **String** compartido por otra parte de código.

Anteriormente en este capítulo, vimos cómo utilizar el método **equals** para verificar los contenidos de dos objetos **String**. También vimos que, a diferencia del método **equals**, el operador **==** no compara los contenidos de los objetos referenciados sino las referencias. Esto nos permitirá analizar mediante algunos ejemplos lo expuesto en el párrafo anterior:

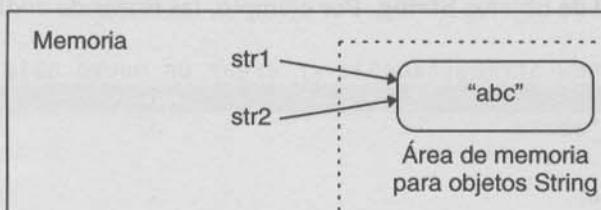
```
String str1 = "abc";
String str2 = "abc";
if (str1.equals(str2))
{
    // el resultado es true siempre
}
if (str1 == str2)
{
    // el resultado es true siempre
}
```

En este ejemplo, el resultado de *str1.equals(str2)* es siempre **true**, lo cual es lógico porque independientemente de que se trate o no de objetos diferentes, los contenidos son los mismos (veremos que se trata de un único objeto).

En cambio, el resultado de la expresión `str1 == str2` es **true** porque ambos identificadores se refieren al mismo objeto. Analicemos por qué.

Cuando se compila la primera línea, Java añade el objeto **String** “abc” al área de memoria destinada a tales objetos. Cuando se compila la segunda línea no se añade un nuevo objeto porque ya existe uno con el mismo literal.

Durante la ejecución, la primera línea almacena en `str1` una referencia al objeto que ya existe en el área de memoria destinada por el compilador a objetos **String**; y la segunda línea almacena en `str2` una referencia al mismo objeto. La figura siguiente muestra esto gráficamente:

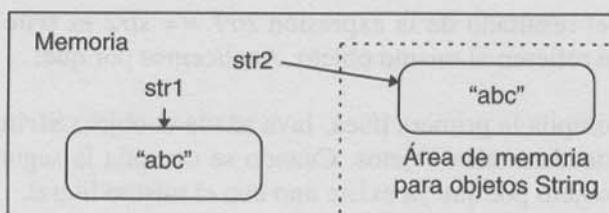


Sin embargo, la utilización del operador **new** hace que se asigne memoria para un nuevo objeto. Por ejemplo:

```
String str1 = new String("abc"); // crear un nuevo objeto
String str2 = "abc";
if (str1.equals(str2))
{
    // el resultado es true siempre
}
if (str1 == str2)
{
    // el resultado es false siempre
}
```

Cuando se compila la primera línea del ejemplo anterior, Java añade el objeto **String** “abc” al área de memoria destinada a tales objetos. Cuando se compila la segunda línea no se añade un nuevo objeto porque ya existe uno con el mismo literal.

Durante la ejecución, la primera línea construye un nuevo objeto **String** duplicando el existente en el área de memoria citada y almacena una referencia al mismo en `str1`. En cambio, la segunda línea simplemente almacena en `str2` una referencia al objeto que ya existe en el área de memoria destinada por el compilador a objetos **String**. La figura siguiente muestra esto gráficamente:



No obstante, es posible colocar los objetos **String** creados dinámicamente (objetos creados durante la ejecución mediante el operador **new**) en el espacio de memoria reservado por el compilador Java para tales objetos utilizando su método **intern**. Esto puede redundar en un ahorro de memoria en programas que utilicen una gran cantidad de objetos **String**. Por ejemplo, las líneas de código siguiente:

```
String str1 = new String("abc"); // crear un nuevo objeto
str1 = str1.intern();
String str2 = "abc";
```

son equivalentes a:

```
String str1 = "abc";
String str2 = "abc";
```

El método **intern** coloca el objeto que recibe el mensaje **intern** en el espacio de memoria reservado por el compilador Java para los objetos **String** si aún no estaba, o si estaba lo reutiliza.

Finalmente, es importante recordar que cuando un método actúa sobre un objeto **String** el resultado es un nuevo objeto lo que mantiene intacto el objeto original. Por ejemplo:

```
str1 = str1.replace('a', 'x');
```

Partiendo de que *str1* era el **String** “abc”, después de ejecutarse la línea de código anterior se genera un nuevo objeto “xbc” referenciado por *str1*. Esto hace que el objeto referenciado por *str2*, que antes de la ejecución era el mismo que el referenciado por *str1*, permanezca inalterado. El nuevo objeto generado no se añadirá al espacio reservado para los objetos **String** a no ser que se invoque explícitamente al método **intern**.

EJERCICIOS RESUELTOS

1. Un algoritmo que genere una secuencia aleatoria o aparentemente aleatoria de números, se llama generador de números aleatorios. Muchos programas requieren

de un algoritmo como éste. El algoritmo más comúnmente utilizado para generar números aleatorios es el de congruencia lineal que se enuncia de la forma siguiente:

$$r_k = (\text{multiplicador} * r_{k-1} + \text{incremento}) \% \text{módulo}$$

donde se observa que cada número en la secuencia r_k , es calculado a partir de su predecesor r_{k-1} (% es el operador módulo o resto de una división entera). La secuencia, así generada, es llamada más correctamente secuencia seudoaleatoria, ya que cada número generado, depende del anteriormente generado.

El método **random** de la clase **Math** del paquete **java.lang**, o bien los métodos de la clase **Random** del paquete **java.util** están basados en este algoritmo.

El siguiente método utiliza el algoritmo de *congruencia lineal* para generar un número aleatorio entre 0 y 1, y no causará sobrepasamiento en un ordenador que admita un rango de enteros de -2^{31} a $2^{31}-1$.

```
public static double rnd(int[] random)
{
    random[0] = (25173 * random[0] + 13849) % 65536;
    return ((double)random[0] / 65535);
}
```

El método *rnd* anterior tiene un parámetro de tipo **int[]** que permitirá pasar un argumento de tipo **int** por referencia. De esta forma, el método podrá modificar el argumento pasado con el valor del último número seudoaleatorio calculado, lo que permitirá calcular el siguiente número seudoaleatorio en función del anterior. Se puede observar que, en realidad, el número seudoaleatorio calculado es un valor entre 0 y 65535 y que para convertirlo a un valor entre 0 y 1 lo dividimos por 65535; el cociente de tipo **double** es el valor devuelto por el método.

El siguiente programa muestra como utilizar el método *rnd* para generar números seudoaleatorios entre 0 y 1:

```
import java.util.*;

public class CRandom
{
    // Números aleatorios entre 0 y 1
    public static double rnd(int[] random)
    {
        random[0] = (25173 * random[0] + 13849) % 65536;
        return ((double)random[0] / 65535);
    }
}
```

```

public static void main(String[] args)
{
    int inicio = (int)((new Date()).getTime()%65536); // semilla
    int[] random = {inicio}; // random = número entre 0 y 65535

    // Generar números seudoaleatorios
    double n;
    for (int i = 10; i != 0; i--)
    {
        n = rnd(random);
        System.out.println(n);
    }
}
}

```

El método **main** del ejemplo anterior primero calcula un valor entre 0 y 65535 a partir del cual se generará el primer número seudoaleatorio; este valor, que es el resto de dividir el número de milisegundos transcurridos desde el 1 de enero de 1970 devuelto por el método **getTime** de la clase **Date** entre 65536, se almacena en el primer elemento de una matriz denominada *random*. Después, para calcular cada número seudoaleatorio, invoca al método *rnd* pasando el argumento *random* por referencia; de esta forma, el método *rnd* podrá modificarlo con el número seudoaleatorio que calcule, lo que garantizará calcular cada número seudoaleatorio en función del anterior.

- Utilizando el método **random** de la clase **Math** del paquete **java.lang**, realizar un programa que muestre 6 números aleatorios diferentes entre 1 y 49 ordenados ascendenteamente.

Para producir enteros aleatorios en un intervalo dado puede utilizar la fórmula: *Parte_entera_de((límiteSup - límiteInf + 1) * random + límiteInf)*.

La solución al problema planteado puede ser de la siguiente forma:

- Definimos el rango de los números que deseamos obtener, así como una matriz para almacenar los 6 números aleatorios.

```

int límiteSup = 49, límiteInf = 1;
int n[] = new int[6], i, k;

```

- Obtenemos el siguiente número aleatorio y verificamos si ya existe en la matriz, en cuyo caso lo desechamos y volvemos a obtener otro. Este proceso lo repetiremos hasta haber generado todos los números solicitados.

```

for (i = 0; i < n.length; i++)
{

```

```

n[i] = (int)((límiteSup - límiteInf + 1) * Math.random() +
            límiteInf);
for (k = 0; k < i; k++)
    if (n[k] == n[i]) // ya existe
    {
        i--;
        break;
    }
}

```

La sentencia **for** externa define cuántos números se van generar. Cuando se genera un número se almacena en la siguiente posición de la matriz. Después, la sentencia **for** interna compara el último número generado con todos los anteriormente generados. Si ya existe, se decrementa el índice *i* de la matriz para que cuando sea incrementado de nuevo por el **for** externo apunte al elemento repetido y sea sobreescrito por el siguiente número generado.

- Una vez obtenidos todos los números, ordenamos la matriz y la visualizamos.

```

Arrays.sort(n);
for (i = 0; i < n.length; i++)
    System.out.print(n[i] + " ");

```

El programa completo se muestra a continuación.

```

import java.util.*;
public class CRandomJava
{
    // Obtener números dentro de un rango
    public static void main(String[] args)
    {
        int límiteSup = 49, límiteInf = 1;
        int n[] = new int[6], i, k;

        for (i = 0; i < n.length; i++)
        {
            // Obtener un número aleatorio
            n[i] = (int)((límiteSup - límiteInf + 1) * Math.random() +
                        límiteInf);
            // Verificar si ya existe el último número obtenido
            for (k = 0; k < i; k++)
                if (n[k] == n[i]) // ya existe
                {
                    i--; // i será incrementada por el for externo
                    break; // salir de este for
                }
        }
    }
}

```

```
// Clasificar la matriz  
Arrays.sort(n);  
// Mostrar la matriz  
for (i = 0; i < n.length; i++)  
    System.out.print(n[i] + " ");  
System.out.println();  
}
```

3. Realizar un programa que partiendo de dos matrices de cadenas de caracteres clasificadas en orden ascendente, construya y visualice una tercera matriz también clasificada en orden ascendente. La idea que se persigue es construir la tercera lista clasificada; no construirla y después clasificarla empleando el método `sort`.

Para ello, el método **main** proporcionará las dos matrices e invocará a un método cuyo prototipo será el siguiente:

```
int Fusionar(String[] lista1, String[] lista2, String[] lista3);
```

El primer parámetro y el segundo del método *Fusionar* son las dos matrices de partida, y el tercero es la matriz que almacenará los elementos de las dos anteriores.

El proceso de fusión consiste en:

- a) Partiendo de que ya están construidas las dos matrices de partida, tomar un elemento de cada una de las matrices.
 - b) Comparar los dos elementos (uno de cada matriz) y almacenar en la matriz resultado el menor.
 - c) Tomar el siguiente elemento la matriz a la que pertenecía el elemento almacenado en la matriz resultado, y volver al punto b).
 - d) Cuando no queden más elementos en una de las dos matrices de partida, se copian directamente en la matriz resultado, todos los elementos que queden en la otra matriz.

El programa completo se muestra a continuación.

```
public class CFusionarListas
{
    // Fusionar dos listas clasificadas
    public static int Fusionar(String[] listaA, String[] listaB,
                               String[] listaC)
    {
        ...
    }
}
```

```
int ind = 0, indA = 0, indB = 0, indC = 0;

if (listaA.length + listaB.length == 0)
    return 0;

// Fusionar las listas A y B en la C
while (indA < listaA.length && indB < listaB.length)
    if (listaA[indA].compareTo(listaB[indB]) < 0)
        listaC[indC++] = listaA[indA++];
    else
        listaC[indC++] = listaB[indB++];

// Los dos bucles siguientes son para prever el caso de que,
// lógicamente una lista finalizará antes que la otra.
for (ind = indA; ind < listaA.length; ind++)
    listaC[indC++] = listaA[ind];

for (ind = indB; ind < listaB.length; ind++)
    listaC[indC++] = listaB[ind];

return 1;
}

static public void main(String[] args)
{
    // Iniciamos las listas a clasificar (puede sustituir este
    // proceso, por otro de lectura con el fin de tomar los
    // datos de la entrada estándar).
    String[] listal = { "Ana", "Carmen", "David",
                        "Francisco", "Javier", "Jesús",
                        "José", "Josefina", "Luis",
                        "María", "Patricia", "Sonia" };

    String[] lista2 = { "Agustín", "Belén", "Daniel",
                        "Fernando", "Manuel",
                        "Pedro", "Rosa", "Susana" };

    // Declarar la matriz que va a almacenar el resultado de
    // fusionar las dos anteriores
    String[] lista3 = new String[listal.length + lista2.length];

    // Fusionar listal y lista2 y almacenar el resultado en lista3.
    // El método "Fusionar" devolverá un 0 cuando no se pueda
    // realizar la fusión.
    int ind, r;
    r = Fusionar(listal, lista2, lista3);

    // Escribir la matriz resultante
    if (r != 0)
    {
```

```

        for (ind = 0; ind < lista3.length; ind++)
            System.out.println(lista3[ind]);
    }
    else
        System.out.println("Error");
}
}

```

Observe que el método *Fusionar* copia referencias. Como se expuso anteriormente en este mismo capítulo, esta forma de proceder ahorra memoria y no causa problemas porque los **String** son objetos no modificables; por lo tanto, no hay posibilidad de que una parte del código pueda modificar un objeto **String** compartido por otra parte de código.

4. Escribir un programa que calcule la serie:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Para un valor de x dado, se calcularán y sumarán términos sucesivos de la serie, hasta que el último término sumado sea menor o igual que una constante de error predeterminada (por ejemplo 1e-7). Observe que cada término es igual al anterior por x/n para $n = 1, 2, 3, \dots$. El primer término es el 1. Para ello se pide:

- a) Escribir un método que tenga el siguiente prototipo:

```
static double exponencial(float x);
```

Este método devolverá como resultado el valor aproximado de e^x .

- b) Escribir el método **main** para que invoque al método *exponencial* y compruebe que para x igual a 1 el resultado es el número e .

El programa completo se muestra a continuación.

```

import java.io.*;
public class Test
{
    static double exponencial(double x)
    {
        int n = 1;
        double exp, término = 1;
        exp = término; // primer término
        while (término > 1e-7)
        {
            término *= x/n; // siguiente término
        }
    }
}

```

```

        exp += término; // sumar otro término
        n++;
    }
    return exp;
}

public static void main(String[] args)
{
    double exp, x;
    System.out.print("Valor de x: "); x = Leer.datoFloat();
    exp = exponencial(x);
    System.out.println("exp(" + x + ") = " + exp);
}
}

```

EJERCICIOS PROPUESTOS

- Realizar un programa que se comporte como un diccionario Inglés-Español; esto es, solicitará una palabra en inglés y escribirá la correspondiente palabra en español. El número de parejas de palabras es variable, pero limitado a un máximo de 100. La longitud máxima de cada palabra será de 40 caracteres. Por ejemplo, suponer que introducimos las siguientes parejas de palabras:

<i>book</i>	libro
<i>green</i>	verde
<i>mouse</i>	ratón

Una vez finalizada la introducción de las listas de palabras pasamos al modo traducción, de forma que si tecleamos *green*, la respuesta ha de ser *verde*. Si la palabra no se encuentra se emitirá un mensaje que lo indique.

El programa constará al menos de dos métodos:

- crearDiccionario*. Este método creará el diccionario.
- traducir*. Este método realizará la labor de traducción.

- Un cuadrado mágico se compone de números enteros comprendidos entre 1 y n^2 , donde n es un número impar que indica el orden de la matriz cuadrada que contiene los números que forman dicho cuadrado mágico. La matriz que forma este cuadrado mágico, cumple que la suma de los valores que componen cada fila, cada columna y cada diagonal es la misma. Por ejemplo, un cuadrado mágico de orden 3, implica un valor de $n = 3$ lo que dará lugar a una matriz de 3 por 3. Por lo tanto, los valores de la matriz estarán comprendidos entre 1 y 9 y dispuestos de la forma siguiente:

```

8 1 6
3 5 7
4 9 2

```

Realizar un programa que visualice un cuadrado mágico de orden impar n . El programa verificará que n es impar y que está comprendido entre 3 y 15.

Una forma de construirlo puede ser: situar el número 1 en el centro de la primera línea, el número siguiente en la casilla situada encima y a la derecha, y así sucesivamente. Es preciso tener en cuenta que el cuadrado se cierra sobre sí mismo, esto es, la línea encima de la primera es la última y la columna a la derecha de la última es la primera. Siguiendo esta regla, cuando el número caiga en una casilla ocupada, se elige la casilla situada debajo del último número situado.

Se deberán realizar al menos los métodos siguientes:

- a) *esImpar*. Este método verificará si n es impar.
 - b) *cuadradoMágico*. Este método construirá el cuadrado mágico.
3. Realizar un programa que:
- a) Lea dos cadenas de caracteres denominadas *cadena1* y *cadena2* y un número entero n .
 - b) Llame a un método:

```
static int compcads(cadena1, cadena2, n);
```

que compare los n primeros caracteres de *cadena1* y de *cadena2*, y devuelva como resultado un valor entero:

0	si <i>cadena1</i> y <i>cadena2</i> son iguales
1	si <i>cadena1</i> es mayor que <i>cadena2</i> (los n primeros caracteres)
-1	si <i>cadena1</i> es menor que <i>cadena2</i> (los n primeros caracteres)

Si n es menor que 1 o mayor que la longitud de la menor de las cadenas, la comparación se hará sin tener en cuenta este parámetro.

- c) Escriba la cadena que sea menor según los n primeros caracteres (esto es, la que esté antes por orden alfabético).
4. Realizar un programa que lea un conjunto de valores reales a través del teclado, los almacene en una matriz de m filas por n columnas y a continuación, visualice la matriz por filas.

La estructura del programa estará formada, además de por el método **main**, por los métodos siguientes:

```
static void leerMatriz2D(float[][] m);
```

El parámetro *m* del método *leerMatriz2D* es la matriz cuyos elementos deseamos leer.

```
static float[] sumaColsMatriz2D(float[][] m);
```

El método *sumaColsMatriz2D* devolverá una matriz unidimensional con la suma de las columnas de la matriz *m* de dos dimensiones pasada como argumento.

5. Escribir un programa para evaluar la expresión $(ax + by)^n$. Para ello, tenga en cuenta las siguientes expresiones:

$$(ax + by)^n = \sum_{k=0}^n \binom{n}{k} (ax)^{n-k} (by)^k$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$n! = n * (n-1) * (n-2) * ... * 2 * 1$$

- a) Escribir un método cuyo prototipo sea:

```
static long factorial(int n);
```

El método *factorial* recibe como parámetro un entero y devuelve el factorial del mismo.

- b) Escribir un método con el prototipo:

```
static long combinaciones(int n, int k);
```

El método *combinaciones* recibe como parámetros dos enteros *n* y *k*, y devuelve como resultado el valor de $\binom{n}{k}$.

- c) Escribir un método que tenga el prototipo:

```
static long potencia(int base, int exponente);
```

- El método *potencia* recibe como parámetros dos enteros, *base* y *exponente*, y devuelve como resultado el valor de $\text{base}^{\text{exponente}}$.
- d) El método **main** leerá los valores de a , b , n , x e y , y utilizando los métodos anteriores escribirá como resultado el valor de $(ax + by)^n$.

PARTE

2

Programación avanzada

- Clases y paquetes
- Subclases e interfaces
- Excepciones
- Trabajar con ficheros
- Estructuras dinámicas
- Algoritmos
- Hilos

algunas veces, obviamente, el nombre de la clase no coincide con el nombre del archivo que contiene el código fuente de la clase. La razón es que el nombre de la clase es el nombre que se le da a la clase en el código fuente, y el nombre del archivo es el nombre que se le da al archivo que contiene el código fuente.

Por ejemplo, si se tiene el siguiente código que define una clase llamada `Alumno`, el nombre de la clase es `Alumno` y el nombre del archivo que contiene el código fuente es `Alumno.java`.

`class Alumno {`
 `// Atributos y métodos de la clase`
`}`

CAPÍTULO 9

© F.J.Ceballos/RA-MA

CLASES Y PAQUETES

Seguro que a estas alturas el término *clase* ya le es familiar. En los capítulos expuestos hasta ahora se han desarrollado aplicaciones sencillas, para introducirle más bien en el lenguaje y en el manejo de la biblioteca de clases de Java que en el diseño de clases. No obstante, sí ha tenido que quedar claro que un programa orientado a objetos sólo se compone de objetos y que un objeto es la concreción de una clase. Sirva como ejemplo las aplicaciones que hemos desarrollado: todas están basadas en una clase aplicación. Es hora pues de entrar con detalle en la programación orientada a objetos la cual tiene un elemento básico: la *clase*. En este capítulo, aprenderemos también a organizar las clases en paquetes, lo que supone también un nivel más de protección para las mismas.

DEFINICIÓN DE UNA CLASE

Una *clase* es un tipo definido por el usuario que describe los atributos y los métodos de los objetos que se crearán a partir de la misma. Los *atributos* definen el estado de un determinado objeto y los *métodos* son las operaciones que definen su comportamiento. Forman parte de estos métodos los *constructores*, que permiten iniciar un objeto, y los *destructores*, que permiten destruirlo. Los atributos y los métodos se denominan en general *miembros* de la clase.

Según hemos aprendido, la definición de una clase consta de dos partes: el *nombre de la clase* precedido por la palabra reservada `class`, y el *cuerpo de la clase* encerrado entre llaves. Esto es:

```
class nombre_clase
{
    cuerpo de la clase
}
```

El *cuerpo de la clase* en general consta de modificadores de acceso (**public**, **protected** y **private**), atributos, mensajes y métodos. Un método implícitamente define un mensaje (el nombre del método es el mensaje).

Por ejemplo, un círculo puede ser descrito por la posición *x*, *y* de su centro y por su *radio*. Hay varias cosas que nosotros podemos hacer con un círculo: calcular la longitud de la circunferencia, calcular el área del círculo, etc. Cada círculo es diferente (por ejemplo, tienen el centro o el radio diferente); pero visto como una *clase* de objetos, el círculo tiene propiedades intrínsecas que nosotros podemos agrupar en una definición. El siguiente ejemplo define la clase *Círculo*. Observar cómo los atributos y los métodos forman el cuerpo de la clase.

```
class Círculo
{
    // miembros privados
    private double x, y;      // coordenadas del centro
    private double radio;     // radio del círculo

    // miembros protegidos
    protected void msgEsNegativo()
    {
        System.out.println("El radio es negativo. Se convierte a positivo");
    }

    // miembros públicos
    public Círculo() {} // constructor sin parámetros
    public Círculo(double cx, double cy, double r) // constructor
    {
        x = cx; y = cy;
        if (r < 0)
        {
            msgEsNegativo();
            r = -r;
        }
        radio = r;
    }

    public double longCircunferencia()
    {
        return 2 * Math.PI * radio;
    }

    public double áreaCírculo()
    {
        return Math.PI * radio * radio;
    }
}
```

Este ejemplo define un nuevo tipo de datos, *Círculo*, que puede ser utilizado dentro de un programa fuente exactamente igual que cualquier otro tipo. Un objeto de la clase *Círculo* tendrá los atributos *x*, *y* y *radio*, los métodos *msgEsNegativo*, *longCircunferencia* y *áreaCírculo*, y dos constructores *Círculo*, uno sin parámetros y otro con ellos.

Atributos

Los atributos constituyen la *estructura interna* de los objetos de una clase. Para declarar un atributo, proceda exactamente igual que ha hecho para declarar cualquier otra variable dentro de un método. Por ejemplo:

```
class Círculo
{
    private double x, y;
    private double radio;
    // ...
}
```

En una clase, cada atributo debe tener un nombre único. En cambio, se puede utilizar el mismo nombre con atributos, en general con miembros, que pertenezcan a diferentes clases.

Es posible asignar un valor inicial a un atributo de una clase. Por ejemplo, en la clase *Círculo* podemos iniciar el radio con el valor 1, aunque generalmente esto no se hace, ya que como expondremos un poco más adelante este tipo de operaciones son típicas del constructor de la clase:

```
class Círculo
{
    private double x, y;
    private double radio = 1;
    // ...
}
```

También podemos declarar como atributos de una clase, referencias a otros objetos de clases existentes. El siguiente ejemplo define la clase *Punto* y después declara el atributo *centro* de *Círculo*, de la clase *Punto*.

```
class Punto
{
    private double x, y;

    Punto(double cx, double cy) { x = cx; y = cy; }
}
```

```

class Círculo
{
    private Punto centro; // coordenadas del centro
    private double radio; // radio del círculo
    // ...
}

```

El orden de las clases es indiferente. Esta forma de proceder ya ha sido utilizada en capítulos anteriores. Recuerde, por ejemplo, que en más de una ocasión hemos declarado un atributo de la clase **String**.

Métodos de una clase

Los métodos generalmente forman lo que se denomina *interfaz* o medio de acceso a la estructura interna de los objetos; ellos definen las operaciones que se pueden realizar con sus atributos. Desde el punto de vista de la POO, el conjunto de todos estos métodos se corresponde con el conjunto de mensajes a los que los objetos de una clase pueden responder.

Para definir un método miembro de una clase, proceda exactamente igual que ha hecho para definir cualquier otro método en las aplicaciones realizadas en los capítulos anteriores. No olvide que una aplicación se basa en una clase. Como ejemplo puede observar los métodos *Círculo* y *longCircunferencia* de la clase *Círculo*.

```

class Círculo
{
    // ...
    public Círculo(double cx, double cy, double r) // constructor
    {
        x = cx; y = cy;
        if (r < 0)
        {
            msgEsNegativo();
            r = -r;
        }
        radio = r;
    }

    public double longCircunferencia()
    {
        return 2 * Math.PI * radio;
    }
    // ...
}

```

En Java un método es una definición incluida siempre dentro del cuerpo de una clase. Asimismo, recuerde que los métodos no se pueden anidar.

Control de acceso a los miembros de la clase

El concepto de clase incluye la idea de ocultación de datos, que básicamente consiste en que no se puede acceder a los atributos directamente, sino que hay que hacerlo a través de métodos de la clase. Esto quiere decir que, de forma general, el usuario de la clase sólo tendrá acceso a uno o más métodos que le permitirán acceder a los miembros privados, ignorando la disposición de éstos (dichos métodos se denominan *métodos de acceso*). De esta forma se consiguen dos objetivos importantes:

1. Que el usuario no tenga acceso directo a la estructura de datos interna de la clase, para que no pueda generar código basado en esa estructura.
2. Que si en un momento determinado alteramos la definición de la clase, excepto el prototipo de los métodos, todo el código escrito por el usuario basado en estos métodos no tendrá que ser retocado.

Piense que si el objetivo uno no se cumpliera, cuando se diera el objetivo dos el usuario tendría que reescribir el código que hubiera desarrollado basándose en la estructura interna de los datos.

Para controlar el acceso a los miembros de una clase, Java provee las palabras clave **private** (privado), **protected** (protegido) y **public** (público), aunque también es posible omitirlas (acceso predeterminado). Estas palabras clave, denominadas *modificadores de acceso*, son utilizadas para indicar el tipo de acceso permitido a cada miembro de la clase. Si observamos la clase *Círculo* expuesta anteriormente identificamos miembros privados, protegidos y públicos.

Acceso predeterminado

En muchos de los ejemplos realizados en los capítulos anteriores, no se ha especificado ningún tipo de control de acceso. Esto es, los atributos y los métodos se declararon de forma análoga a como puede observar en el ejemplo siguiente:

```
class Cacional
{
    int Numerador;
    int Denominador;

    void AsignarDatos(int num, int den)
    {
```

```

        Numerador = num;
        if (den == 0) den = 1; // el denominador no puede ser cero
        Denominador = den;
    }

    void VisualizarRacional()
    {
        System.out.println(Numerador + "/" + Denominador);
    }
}

```

Un miembro de una clase declarado sin modificadores que indiquen el control de acceso, puede ser accedido por cualquier clase perteneciente al mismo paquete. Ninguna otra clase, o subclase, fuera de este paquete puede tener acceso a estos miembros (estudiaremos las subclases en el capítulo siguiente). Recuerde que las clases implementadas en nuestros programas pertenecen, por omisión, al paquete predeterminado (vea en el capítulo 4 "Paquetes y protección de clases"). De esta forma Java asegura que toda clase pertenece a un paquete.

Como se puede observar este tipo de control de acceso no tiene mucho dominio sobre el mismo. Si lo que se pretende es tener un control preciso sobre cómo va a ser utilizada nuestra clase por otras, deberemos utilizar los modificadores **private**, **protected** o **public** en vez de aceptar el tipo de control predeterminado.

Acceso público

Un miembro declarado **public** (público) está accesible para cualquier otra clase o subclase que necesite utilizarlo. La interfaz pública de una clase, o simplemente interfaz, está formada por todos los miembros públicos de la misma. Asimismo, los atributos **static** de la clase generalmente son declarados públicos. Sirva como ejemplo el atributo **PI** de la clase **Math**: *public static final double PI*.

Acceso privado

Un miembro declarado **private** (privado) es accesible solamente por los métodos de su propia clase. Esto significa que no puede ser accedido por los métodos de cualquier otra clase, incluidas las subclases.

Acceso protegido

Un miembro declarado **protected** (protegido) se comporta exactamente igual que uno privado para los métodos de cualquier otra clase, excepto para los métodos de las clases del mismo paquete o de sus subclases con independencia del paquete al que pertenezcan, para las que se comporta como un miembro público.

IMPLEMENTACIÓN DE UNA CLASE

La programación orientada a objetos sugiere separar la implementación de cada clase en un fichero *.class*, fundamentalmente para posteriormente reutilizar y mantener dicha clase. Como ejemplo, diseñaremos una clase que almacene una fecha, verificando que es correcta; esto es, que el día esté entre los límites 1 y días del mes, que el mes esté entre los límites 1 y 12 y que el año sea mayor o igual que 1582.

Parece lógico que la estructura de datos de un objeto fecha esté formada por los miembros *día*, *mes* y *año*, y permanezca oculta al usuario. Por otra parte, las operaciones sobre estos objetos tendrán que permitir asignar una fecha, método *asignarFecha*, obtener una fecha de un objeto existente, método *obtenerFecha*, y verificar si la fecha que se quiere asignar es correcta, método *fechaCorrecta*. Estos tres métodos formarán la interfaz pública. Cuando el día corresponda al mes de febrero, el método *fechaCorrecta* necesitará comprobar si el año es bisiesto para lo que añadiremos el método *bisiesto*. Ya que un usuario no necesita acceder a este método, lo declararemos protegido con la intención de que, en un futuro, sí pueda ser accedido desde una subclase. Según lo expuesto, podemos escribir una clase denominada *CFecha* así:

```
public class CFecha
{
    // Atributos
    private int dia, mes, año;

    // Métodos
    protected boolean bisiesto()
    {
        // cuerpo del método
    }

    public void asignarFecha(int dd, int mm, int aaaa)
    {
        // cuerpo del método
    }

    public void obtenerFecha(int[] fecha)
    {
        // cuerpo del método
    }

    public boolean fechaCorrecta()
    {
        // cuerpo del método
    }
}
```

El paso siguiente es definir cada uno de los métodos. Al hablar de los modificadores de acceso quedó claro que cada uno de los métodos de una clase tiene acceso directo al resto de los miembros. Según esto, la definición del método *asignarFecha* puede escribirse así:

```
public void asignarFecha(int dd, int mm, int aaaa)
{
    día = dd; mes = mm; año = aaaa;
}
```

Observe que por ser *asignarFecha* un método miembro de la clase *CFecha*, puede acceder directamente a los atributos *día*, *mes* y *año* de su misma clase, independientemente de que sean privados. Estos atributos corresponderán en cada caso al objeto que recibe el mensaje *asignarFecha* (objeto para el que se invoca el método; vea más adelante, en este mismo capítulo, la referencia implícita **this**). Por ejemplo, si declaramos los objetos *fecha1* y *fecha2* de la clase *CFecha*, y enviamos a *fecha1* el mensaje *asignarFecha*:

```
fecha1.asignarFecha(dd, mm, aaaa);
```

como respuesta a este mensaje, se ejecuta el método *asignarFecha* que asigna los datos *dd*, *mm* y *aaaa* al objeto *fecha1*; esto es, a *fecha1.día*, *fecha1.mes* y *fecha1.año*; y si a *fecha2* le enviamos también el mensaje *asignarFecha*:

```
fecha2.asignarFecha(dd, mm, aaaa);
```

como respuesta a este mensaje, se ejecuta el método *asignarFecha* que asigna los datos *dd*, *mm* y *aaaa* al objeto *fecha2*; esto es, a *fecha2.día*, *fecha2.mes* y *fecha2.año*.

Siguiendo las reglas enunciadas, finalizaremos el diseño de la clase escribiendo el resto de los métodos. El resultado que se obtendrá será la clase *CFecha* que se observa a continuación:

```
///////////////////////////////
// Definición de la clase CFecha
public class CFecha
{
    // Atributos
    private int día, mes, año;

    // Métodos
    protected boolean bisiesto()
    {
        return ((año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0));
    }
}
```

```

public void asignarFecha(int dd, int mm, int aaaa)
{
    día = dd; mes = mm; año = aaaa;
}

public void obtenerFecha(int[] fecha)
{
    fecha[0] = día;
    fecha[1] = mes;
    fecha[2] = año;
}

public boolean fechaCorrecta()
{
    boolean díaCorrecto, mesCorrecto, añoCorrecto;
    // ¿año correcto?
    añoCorrecto = (año >= 1582);
    // ¿mes correcto?
    mesCorrecto = (mes >= 1) && (mes <= 12);
    switch (mes)
    // ¿día correcto?
    {
        case 2:
            if (bisielto())
                díaCorrecto = (día >= 1 && día <= 29);
            else
                díaCorrecto = (día >= 1 && día <= 28);
            break;
        case 4: case 6: case 9: case 11:
            díaCorrecto = (día >= 1 && día <= 30);
            break;
        default:
            díaCorrecto = (día >= 1 && día <= 31);
    }
    return díaCorrecto && mesCorrecto && añoCorrecto;
}

```

Resumiendo: la funcionalidad de esta clase está soportada por los atributos privados *día*, *mes* y *año*, y por los métodos *asignarFecha*, *obtenerFecha*, *fechaCorrecta* y *bisielto*.

El método público *asignarFecha*, recibe tres enteros y los almacena en los atributos *día*, *mes* y *año* del objeto que recibe el mensaje *asignarFecha* (objeto para el que se invoca dicho método).

El método público *obtenerFecha*, permite extraer los datos *día*, *mes* y *año* del objeto que recibe el mensaje *obtenerFecha*.

El método público *fechaCorrecta* verifica si la fecha que se desea asignar al objeto que recibe este mensaje es correcta. Este método devuelve **true** si la fecha es correcta y **false** en caso contrario.

El método protegido *bisiesto* verifica si el año de la fecha que se desea asignar al objeto que recibe este mensaje es bisiesto. Este método retorna **true** si el año es bisiesto y **false** en caso contrario.

MÉTODOS SOBRECARGADOS

En los capítulos anteriores, al trabajar con las clases de la biblioteca de Java nos hemos encontrado con clases que implementan varias veces el mismo método. Por ejemplo, en el capítulo 5 dijimos que la clase **InputStream** implementa tres formas del método **read**:

```
public int read()
public int read(byte[] b)
public int read(byte[] b, int off, int len)
```

y que la clase **PrintStream** implementa múltiples formas de los métodos **print** y **println**; por ejemplo:

```
public void print(int i)
public void print(double d)
public void print(char[] s)
```

¿En qué se diferencian los métodos **read**? En su número de parámetros. Y, ¿en qué se diferencian los métodos **print** expuestos? En el tipo de su parámetro.

Pues bien, cuando en una clase un mismo método se define varias veces con distinto número de parámetros, o bien con el mismo número de parámetros pero diferenciándose una definición de otra en que al menos un parámetro es de un tipo diferente, se dice que el método está *sobrecargado*.

Los métodos sobrecargados pueden diferir también en el tipo del valor retornado. Ahora bien, el compilador Java no admite que se declararen dos métodos que sólo difieran en el tipo del valor retornado; deben diferir también en la lista de parámetros; esto es, lo que importa son el número y el tipo de los parámetros.

La sobrecarga de métodos elimina la necesidad de definir métodos diferentes que en esencia hacen lo mismo, como es el caso del método **print**, o también hace posible que un método se comporte de una u otra forma según el número de argumentos con el que sea invocado, como es el caso del método **read**.

Como ejemplo, sobre cargaremos el método *asignarFecha* para que pueda ser invocado con cero argumentos; con un argumento, el día; con dos argumentos, el día y el mes; y con tres argumentos, el día, el mes y el año. Los datos día, mes o año omitidos en cualquiera de los casos, serán obtenidos de la fecha actual proporcionada por el sistema.

La fecha actual del sistema se puede obtener a partir de un objeto de la clase **GregorianCalendar**, que es una subclase de **Calendar**, del paquete **java.util**. La clase **Calendar** es una clase abstracta que proporciona una serie de constantes tales como *DAY_OF_MONTH*, *MONTH* o *YEAR* que podemos utilizar como argumento en el método *get* para obtener el dato al que alude.

```
public void asignarFecha()
{
    // Asignar, por omisión, la fecha actual.
    GregorianCalendar fechaActual = new GregorianCalendar();
    día = fechaActual.get(Calendar.DAY_OF_MONTH);
    mes = fechaActual.get(Calendar.MONTH)+1;
    año = fechaActual.get(Calendar.YEAR);
}

public void asignarFecha(int dd)
{
    asignarFecha();
    día = dd;
}

public void asignarFecha(int dd, int mm)
{
    asignarFecha();
    día = dd; mes = mm;
}

public void asignarFecha(int dd, int mm, int aaaa)
{
    día = dd; mes = mm; año = aaaa;
}
```

Como se puede observar, el que una definición del método invoque a otra es una técnica de método abreviado que da como resultado métodos más cortos.

Por cada llamada al método *asignarFecha* que escribamos en un programa, el compilador Java debe resolver cuál de los métodos con el nombre *asignarFecha* es invocado. Esto lo hace comparando el número y tipos de los argumentos especificados en la llamada, con los parámetros especificados en las distintas definiciones del método. El siguiente ejemplo muestra las posibles formas de invocar al método *asignarFecha*:

```

fecha.asignarFecha();
fecha.asignarFecha(día);
fecha.asignarFecha(día, mes);
fecha.asignarFecha(día, mes, año);

```

Si el compilador Java no encontrara un método exactamente con los mismos tipos de argumentos especificados en la llamada, realizaría sobre dichos argumentos las conversiones implícitas permitidas entre tipos, tratando de adaptarlos a alguna de las definiciones existentes del método. Si este intento fracasa, entonces se producirá un error.

IMPLEMENTACIÓN DE UNA APLICACIÓN

Recordando lo expuesto en capítulos anteriores, las aplicaciones son programas Java que se ejecutan por sí mismos, a diferencia de los *applets* que requieren de un explorador. Una aplicación consiste en una o más clases, de las cuales una de ellas tiene que ser una clase aplicación: clase que incluya el método **main**. Cuando compile el fichero que contiene su aplicación, el compilador Java generará un fichero *.class* por cada una de las clases que la componen; cada fichero generado tendrá el mismo nombre que la clase que contiene.

Para comprobar que la clase *CFecha* que acabamos de diseñar trabaja correctamente, podemos escribir una aplicación *Test* según se muestra a continuación:

```

///////////////////////////////
// Aplicación que utiliza la clase CFecha
//
public class Test
{
    // Visualizar una fecha
    public static void visualizarFecha(CFecha fecha)
    {
        int[] f = new int[3];
        fecha.obtenerFecha(f);
        System.out.println(f[0] + "/" + f[1] + "/" + f[2]);
    }

    // Establecer una fecha, verificarla y visualizarla
    public static void main(String[] args)
    {
        CFecha fecha = new CFecha(); // objeto de tipo CFecha
        int día, mes, año;
        do
        {
            System.out.print("día, ## : "); día = Leer.datoInt();
            System.out.print("mes, ## : "); mes = Leer.datoInt();

```

```
        System.out.print("año, #### : "); año = Leer.datToInt();
        fecha.asignarFecha(día, mes, año);
    }
    while (!fecha.fechaCorrecta());
    visualizarFecha( fecha );
}
```

Notar que la clase *CFecha* declara los atributos *día*, *mes* y *año* privados. Esto quiere decir que sólo son accesibles por los métodos de su clase. Si un método de otra clase intenta acceder a uno de estos atributos, el compilador genera un error. En cambio, como *CFecha* y *Test* pertenecen al mismo paquete, al *predeterminado*, los métodos de *Test* sí podrían acceder el método protegido *bisiesto* de *CFecha*. Por ejemplo:

```
public static void main(String[] args)
{
    CFecha fecha = new CFecha();
    // ...
    int dd = fecha.día; // error: día es un miembro privado
    fecha.mes = 1; // error: mes es un miembro privado
    boolean esBisiesto = fecha.bisiesto(); // correcto
}
```

En cambio, los métodos *asignarFecha*, *obtenerFecha* y *fechaCorrecta* son públicos. Por lo tanto, son accesibles, además de por los métodos de su clase, por cualquier otro método de otra clase. Sirva como ejemplo el método *visualizarFecha* de la clase *Test*. Este método presenta en la salida estándar la fecha almacenada en el objeto que se le pasa como argumento. Observe que tiene que invocar al método *obtenerFecha* para acceder a los datos de un objeto *CFecha*. Esto es así porque un método que no es miembro de la clase del objeto, no tiene acceso a sus datos privados.

CONTROL DE ACCESO A UNA CLASE

El control de acceso a una clase determina la relación que tiene esa clase con otras clases de otros paquetes. Distinguimos dos niveles de acceso: *de paquete* y *público*. Una clase con nivel de acceso de *paquete* sólo puede ser utilizada por las clases de su paquete (no está disponible para otros paquetes, ni siquiera para los subpaquetes). En cambio, una clase pública puede ser utilizada por cualquier otra clase de otro paquete.

Por omisión una clase tiene el nivel de acceso de *paquete*; por ejemplo, la clase *Círculo* expuesta anteriormente tiene este nivel de acceso (no ha sido declarada

public, por lo que tiene el nivel de acceso de *paquete*). En cambio, cuando se desea que una clase tenga nivel de acceso *público*, hay que calificarla como tal utilizando la palabra reservada **public**; la clase *CFecha* del ejemplo anterior tiene este nivel de acceso. Otro ejemplo: la clase *Leer* utilizada desde la clase *Test* anterior es pública (en nuestro caso está ubicada en la carpeta *misClases* especificada por una de las rutas de la variable de entorno *CLASSPATH*); pero aunque no hubiese sido pública también se podría utilizar desde la clase *Test*, ya que ambas pertenecen al mismo paquete, al predeterminado.

REFERENCIA **this**

Recuerde, en el capítulo 4 aprendió que cada objeto mantiene su propia copia de los atributos pero no de los métodos de su clase, de los cuales sólo existe una copia para todos los objetos de esa clase. Esto es, cada objeto almacena sus propios datos, pero para acceder y operar con ellos, todos comparten los mismos métodos definidos en su clase. Por lo tanto, para que un método conozca la identidad del objeto particular para el que ha sido invocado, Java proporciona una referencia al objeto denominada **this**. Así, por ejemplo, si creamos un objeto *fechal* y a continuación le enviamos el mensaje *asignarFecha*,

```
fechal.asignarFecha(día, mes, año);
```

Java define la referencia **this** para permitir referirse al objeto *fechal* en el cuerpo de el método que se ejecuta como respuesta al mensaje. Esa definición es así:

```
final CFecha this = fechal;
```

Y cuando realizamos la misma operación con otro objeto *fecha2*,

```
fecha2.asignarFecha(día, mes, año);
```

Java define la referencia **this**, para referirse al objeto *fecha2*, de la forma:

```
final CFecha this = fecha2;
```

Según lo expuesto, el método *asignarFecha* podría ser definido también como se muestra a continuación:

```
public void asignarFecha(int dd, int mm, int aa)
{
    this.día = dd; this.mes = mm; this.año = aa;
}
```

¿Qué representa **this** en este método? Según lo explicado, **this** es una referencia al objeto que recibió el mensaje *asignarFecha*; esto es, al objeto sobre el que se está realizando el proceso llevado a cabo por el método *asignarFecha*.

Observe ahora el método **main** de la clase *Test* presentada anteriormente. En él hemos declarado un objeto *fecha* de la clase *CFecha* y posteriormente le hemos enviado un mensaje *fechaCorrecta*:

```
do
{
    System.out.print("día, ## : "); día = Leer.datoInt();
    System.out.print("mes, ## : "); mes = Leer.datoInt();
    System.out.print("año, #### : "); año = Leer.datoInt();
    fecha.asignarFecha(día, mes, año);
}
while (!fecha.fechaCorrecta());
```

En este caso, igual que en el ejemplo anterior, el método *fechaCorrecta* conoce con exactitud el objeto sobre el que tiene que actuar, puesto que se ha expresado explícitamente. Pero ¿qué pasa con el método *bisiesto* que se encuentra sin referencia directa alguna en el cuerpo del método *fechaCorrecta*?

```
public boolean fechaCorrecta()
{
    // ...
    if (bisiesto())
    // ...
}
```

En este otro caso, la llamada no es explícita como en el caso anterior. Lo que ocurre en la realidad es que todas las referencias a los atributos y métodos del objeto para el que se invocó el método *fechaCorrecta* (objeto que recibió el mensaje *fechaCorrecta*), son implícitamente realizadas a través de **this**. Según esto, la sentencia **if** anterior podría escribirse también así:

```
if (this.bisiesto())
```

Normalmente en un método no es necesario utilizar esta referencia para acceder a los miembros del objeto implícito, pero es útil cuando haya que devolver una referencia al mismo.

VARIABLES, MÉTODOS Y CLASES FINALES

En el capítulo 3 ya fueron expuestas las variables finales, generalmente denominadas constantes porque nunca cambian su valor. También vimos su uso junto con

static para hacer que la constante sea de la clase y no del objeto. Declarar una referencia **final** a un objeto supone que esa referencia sólo pueda utilizarse para referenciar ese objeto; cualquier intento accidental de modificar dicha referencia para que señale a otro objeto será detectado durante la compilación, en vez de causar errores durante la ejecución. Por ejemplo:

```
final CFecha cumpleaños = new CFecha();
CFecha fecha = new CFecha();
// ...
cumpleaños = fecha; // Error: referencia constante
```

Declarar un método **final** supone que la clase se ejecute con más eficiencia, porque el compilador puede colocar el código de bytes del método directamente en el lugar del programa donde se invoque a dicho método, ya que se garantiza que el método no va a cambiar. Por ejemplo:

```
public final void asignarFecha(int dd, int mm, int aa)
{
    día = dd; mes = mm; año = aa;
}
```

¿Quién puede cambiar el método? Una subclase que intente redefinirlo, pero sólo podrá hacerlo si el método no es **final** (estudiaremos las subclases en el capítulo siguiente).

Quizás cuando desarrolle una clase por primera vez no tenga muchas razones para decidir qué métodos puede declarar **final**. Hágalo cuando necesite que la clase se ejecute con más rapidez, pero pensando en la limitación que está imponiendo a las posibles subclases de esa clase. La biblioteca de Java declara **final** muchos de los métodos que se utilizan con mayor frecuencia con la intención de obtener una mayor eficiencia durante la ejecución.

Una clase, también se puede declarar **final**. Por ejemplo:

```
public final class CFecha
{
    // ...
}
```

Cuando una clase se declara **final** estamos impidiendo que de esa clase se puedan derivar subclases. Además todos sus métodos se convierten automáticamente en **final**. No hay muchas razones para hacer esto ya que sacrificamos una de las características más potentes de la POO: la reutilización del código. En algunos casos excepcionales, como ocurre con la clase **Math** de la biblioteca de Java, puede ser beneficioso por las razones expuestas al hablar de los métodos **final**.

INICIACIÓN DE UN OBJETO

Sabemos que un objeto consta de una estructura interna (los atributos) y de una interfaz que permite acceder y manipular tal estructura (los métodos). Ahora, ¿cómo se construye un objeto de una clase cualquiera? Pues, de forma análoga a como se construye cualquier otra variable de un tipo predefinido. Por ejemplo:

```
int edad;
```

Este ejemplo define la variable *edad* del tipo predefinido **int**. En este caso, el compilador automáticamente reserva memoria para su ubicación, le asigna un valor (cero si se trata de un atributo de una clase, o indeterminado si es local a un método) y procederá a su destrucción, cuando el flujo de ejecución vaya fuera del ámbito donde haya sido definida.

Esto nos hace pensar en la idea de que de alguna manera el compilador llama a un método de iniciación, *constructor*, para iniciar cada una de las variables declaradas, y a un método de eliminación, *destructor*, para liberar el espacio ocupado por dichas variables, justo al salir del ámbito en el que han sido definidas.

Pues bien, con un objeto de una clase ocurre lo mismo. Por ejemplo,

```
CFecha fecha = new CFecha();
```

Con objetos, el compilador proporciona un *constructor* público por omisión para cada clase definida. Este constructor será ejecutado después que el operador **new**, secuencial y recursivamente (un miembro de una clase puede ser iniciado con un objeto de otra clase) reserve memoria para cada uno de los miembros y los inicie. Igualmente, el compilador proporciona para cada clase de objetos un *destructor* protegido por omisión, que será invocado justo antes de que se destruya un objeto con el fin de permitir realizar tareas de limpieza y liberar recursos.

No obstante, como veremos a continuación, cuando el constructor proporcionado por omisión por Java no satisfaga las necesidades de nuestra clase de objetos, podemos definir uno. Ídem para el destructor.

Constructor

En Java, una forma de asegurar que los objetos siempre contengan valores válidos es escribir un constructor. Un *constructor* es un método especial de una clase que es llamado automáticamente siempre que se crea un objeto de la misma. Su función es iniciar nuevos objetos de su clase. Cuando se crea un objeto, Java hace lo siguiente:

- Asigna memoria para el objeto por medio del operador **new**.
- Inicia los atributos de ese objeto, ya sea a sus valores iniciales (si los atributos fueron iniciados en su propia declaración) o a los valores predeterminados por el sistema: los atributos numéricos a ceros, los alfanuméricos a nulos, y las referencias a objetos a **null**.
- Llama al constructor de la clase que puede ser uno entre varios, según se expone a continuación.

Dado que los constructores son métodos, admiten parámetros igual que éstos. Cuando en una clase no especificamos ningún constructor, el compilador añade uno público por omisión sin parámetros.

Un *constructor por omisión* de una clase *C* es un constructor sin parámetros que no hace nada. Sin embargo, es necesario porque según lo que acabamos de exponer, será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores especificados cuando se declararon los atributos en su clase, o en su defecto, con los valores predeterminados por el sistema.

Un *constructor* se distingue fácilmente porque tiene el mismo nombre que la clase a la que pertenece (por ejemplo, el constructor para la clase *CFecha* se denomina también *CFecha*), no se hereda, no puede retornar un valor (incluyendo **void**) y no puede ser declarado **final**, **static**, **abstract**, **synchronized** o **native** (los dos primeros modificadores ya son conocidos; los otros lo serán en la medida que ampliemos nuestros conocimientos sobre Java).

Como ejemplo, vamos a añadir un constructor a la clase *CFecha* con el fin de poder iniciar los atributos de cada nuevo objeto con unos valores determinados:

```
public class CFecha
{
    // Atributos
    private int dia, mes, año;
    // Métodos
    public CFecha(int dd, int mm, int aaaa) // constructor
    {
        dia = dd; mes = mm; año = aaaa;
        if (!fechaCorrecta())
        {
            System.out.println("Fecha incorrecta. Se asigna la actual.");
            asignarFecha();
        }
    }
    // ...
}
```

Observe que el constructor, salvo en casos excepcionales, debe declararse siempre público para que pueda ser invocado desde cualquier parte, aunque la clase, que se supone pública, pertenezca a otro paquete.

Cuando una clase tiene un constructor, éste será invocado automáticamente siempre que se cree un nuevo objeto de esa clase. El objeto se considera construido con los valores por omisión justo antes de iniciarse la ejecución del constructor. Por lo tanto, a continuación, desde el cuerpo del constructor según se puede observar en el ejemplo anterior, es posible asignar valores a sus atributos, invocar a los métodos de su clase, o bien llamar a métodos de otros objetos.

En el caso de que el constructor tenga parámetros, para crear un nuevo objeto hay que especificar la lista de argumentos correspondiente entre los paréntesis que siguen al nombre de la clase del objeto. El siguiente ejemplo muestra esto con claridad:

```
public class Test
{
    // Visualizar una fecha
    public static void visualizarFecha(CFecha fecha)
    {
        int[] f = new int[3];

        fecha.obtenerFecha(f);
        System.out.println(f[0] + "/" + f[1] + "/" + f[2]);
    }

    public static void main(String[] args)
    {
        // La siguiente linea invoca al constructor de la clase CFecha
        CFecha fecha = new CFecha(1, 3, 2002); // objeto de tipo CFecha
        visualizarFecha( fecha );
    }
}
```

Este ejemplo define un objeto *fecha* e inicia sus datos miembro *día*, *mes* y *año* con los valores 1, 3 y 2002, respectivamente. Para ello, invoca al constructor *CFecha(int dd, int mm, int aaaa)*, le pasa los argumentos 1, 3 y 2002 y ejecuta el código que se especifica en el cuerpo del mismo. Una vez construido el objeto, visualizamos su contenido haciendo uso de un método del mismo que permite obtener sus atributos. La siguiente línea es la salida de la aplicación anterior:

1/3/2002

Añadamos ahora al método **main** de la clase *Test* del ejemplo anterior, la línea de código que se indica a continuación. ¿Qué ocurrirá?

```
CFecha otraFecha = new CFecha();
```

Quizás se sorprenda cuando el compilador Java le indique que la clase *CFecha* no tiene ningún constructor sin parámetros, cuando anteriormente habíamos dicho que Java proporciona para toda clase uno. Lo que sucede es que siempre que en una clase se define explícitamente un constructor, el constructor implícito (constructor por omisión) es reemplazado por éste.

Según lo expuesto, la definición explícita del constructor con parámetros *CFecha(int dd, int mm, int aaaa)*, ha sustituido al constructor por omisión que Java añadió a esa clase. Para solucionar este problema, hay que añadir a la clase un constructor sin parámetros. Por ejemplo, el siguiente:

```
public CFecha() { /* Sin código */ }
```

El constructor anterior, realiza la misma función que el constructor por omisión. No obstante, en el caso de la clase *CFecha*, quizás sea más conveniente, añadir un constructor sin parámetros que inicie cada objeto creado con los valores correspondientes a la fecha actual:

```
public CFecha() // constructor
{
    asignarFecha(); // asignar fecha actual
}
```

Sobrecarga del constructor

Según lo expuesto, es evidente que podemos definir múltiples constructores con el mismo nombre y diferentes parámetros con el fin de poder iniciar un objeto de una clase de diferentes formas. Esto no es nuevo, simplemente es aplicar la técnica de sobrecargar un método, expuesta anteriormente, al constructor de una clase.

Por ejemplo, aplicando lo expuesto, podemos añadir a la clase *CFecha* constructores para iniciar un objeto, por omisión con la fecha actual proporcionada por la función *asignarFecha* sin parámetros, o bien especificando sólo el día, o el día y el mes, o el día, el mes y el año; los valores no especificados se obtendrán de la fecha actual del sistema proporcionada por *asignarFecha*. El código siguiente muestra las distintas sobrecargas que satisfacen lo anteriormente expuesto:

```
public CFecha() // constructor sin parámetros
{
    asignarFecha(); // asignar fecha actual
}
```

```

public CFecha(int dd) // constructor con un parámetro
{
    asignarFecha(); // asignar fecha actual
    dia = dd;
    if (!fechaCorrecta())
    {
        System.out.println("Fecha incorrecta. Se asigna la actual.");
        asignarFecha();
    }
}

public CFecha(int dd, int mm) // constructor con dos parámetros
{
    asignarFecha(); // asignar fecha actual
    dia = dd; mes = mm;
    if (!fechaCorrecta())
    {
        System.out.println("Fecha incorrecta. Se asigna la actual.");
        asignarFecha();
    }
}

public CFecha(int dd, int mm, int aaaa) // construc. con tres pars.
{
    dia = dd; mes = mm; año = aaaa;
    if (!fechaCorrecta())
    {
        System.out.println("Fecha incorrecta. Se asigna la actual.");
        asignarFecha();
    }
}

```

Ahora, podemos invocar al constructor *CFecha* con 0, 1, 2 ó 3 argumentos, según se puede observar en las líneas de código siguientes:

```

CFecha fechal = new CFecha();
CFecha fecha2 = new CFecha(3);
CFecha fecha3 = new CFecha(15, 3);
CFecha fecha4 = new CFecha(1, 3, 2002);

```

Es posible escribir un método que tenga el mismo nombre que el constructor; lógicamente, a diferencia de éste, ahora hay que especificar el tipo del valor retornado. No obstante, esta forma de proceder no es aconsejable porque puede crear confusión a la hora de interpretar el código de la clase. Por ejemplo:

```

public void CFecha(int a, int b, int c)
{
    // ...
}

```

Llamar a un constructor

A diferencia de los otros métodos de la clase, un constructor no puede ser invocado directamente, pero sí indirectamente a través de `this`. Esto permite utilizar la técnica de método abreviado, expuesta al hablar de métodos sobrecargados, también con los constructores. Para llamar a un constructor en la clase actual desde otro constructor utilice la siguiente sintaxis:

```
this([[[[arg1], arg2], arg3], ...]);
```

La llamada a un constructor sólo puede realizarse desde dentro de otro constructor de su misma clase y debe ser siempre la primera sentencia. Por ejemplo, el constructor con un parámetro podría escribirse también así:

```
public CFecha(int dd) // constructor
{
    this(); // invoca al constructor CFecha sin parámetros
    día = dd;
    if (!fechaCorrecta())
    {
        System.out.println("Fecha incorrecta. Se asigna la actual.");
        asignarFecha();
    }
}
```

Asignación de objetos

No olvide que cuando trabaja con objetos lo que realmente manipula desde cualquier método son referencias a los objetos. Por ejemplo:

```
CFecha fechal = new CFecha();
CFecha fecha2 = new CFecha(15);
CFecha fecha3 = new CFecha(22, 3);
CFecha fecha4 = fechal;
fecha3 = fecha2;
```

Este ejemplo crea tres objetos: `fechal`, `fecha2` y `fecha3`. Después declara una nueva referencia `fecha4` y le asigna `fechal`, pero tanto `fechal` como `fecha4` son referencias a objetos `CFecha`, que ahora apuntan al mismo objeto (al referenciarlo por `fechal`). Finalmente, asigna `fecha2` a `fecha3`, con lo que ambas referencias apuntarán al objeto referenciado por `fecha2`.

Lo anteriormente expuesto demuestra que el operador de asignación no sirve para copiar un objeto en otro. ¿Cuál es la solución para resolver el problema planteado? Pues, añadir a la clase `CFecha` un método como el siguiente:

```
public void copiar(CFecha obj)
{
    día = obj.día;
    mes = obj.mes;
    año = obj.año;
}
```

El método *copiar* copia miembro a miembro el objeto pasado como argumento en el objeto que recibe el mensaje *copiar*. Por ejemplo, la siguiente línea de código copia el objeto *fecha2* en el objeto *fecha1*.

```
fecha1.copiar(fecha2);
```

Si ahora quisiéramos copiar el objeto *fecha3* en el objeto *fecha2* y en el objeto *fecha1*, podríamos proceder, por ejemplo, así:

```
fecha2.copiar(fecha3);
fecha1.copiar(fecha2);
```

Pero ¿qué podemos hacer para poder escribir las dos líneas anteriores en una sola? Esto es, para poder escribir:

```
fecha1.copiar(fecha2.copiar(fecha3));
```

Tendríamos que modificar el método *copiar* como se observa a continuación:

```
public CFecha copiar(CFecha obj)
{
    día = obj.día;
    mes = obj.mes;
    año = obj.año;
    return this;
}
```

El método *copiar* devuelve ahora una referencia al objeto resultado de la copia, con lo cual podemos utilizar esta referencia para copiar este objeto en otro; esto es, el hecho de que el método *copiar* retorne una referencia al objeto resultante permite realizar copias múltiples encadenadas.

Constructor copia

Otra forma de iniciar un objeto es asignándole otro objeto de su misma clase en el momento de su creación. Lógicamente, si se crea un objeto tiene que intervenir un constructor. El prototipo para este constructor es de la forma:

nombre_clase(nombre_clase referencia_objeto)

Se puede observar que un constructor de las características especificadas tiene un solo parámetro, que es una referencia a un objeto de su misma clase. Por tratarse de un constructor no hay un valor retornado. Pues bien, un constructor que se invoca para iniciar un nuevo objeto creado a partir de otro existente es denominado *constructor copia*.

Como ejemplo, añada un constructor copia a la clase *CFecha*. Éste será como se indica a continuación:

```
public CFecha(CFecha obj) // constructor copia
{
    día = obj.día;
    mes = obj.mes;
    año = obj.año;
}
```

Vemos que el constructor copia acepta como argumento una referencia al objeto a copiar y después, asigna miembro a miembro ese objeto al nuevo objeto construido. Para probar cómo trabaja, puede añadir a la función **main** de la clase *Test* que escribimos anteriormente, las siguientes líneas de código:

```
CFecha fecha1 = new CFecha(1, 3, 2002);
CFecha fecha2 = new CFecha(fecha1);
```

Este ejemplo crea e inicia un objeto *fecha1* y a continuación crea otro objeto *fecha2* iniciándole con *fecha1*. A diferencia del método *copiar* expuesto en el apartado anterior, inicialmente aquí sólo existe un objeto (*fecha1*); después se crea otro objeto (*fecha2*) y se inicia con el primero.

DESTRUCCIÓN DE OBJETOS

De la misma forma que existe un método que se ejecuta automáticamente cada vez que se construye un objeto, también existe un método que se invoca automáticamente cada vez que se destruye. Este método recibe el nombre genérico de *destructor* y en el caso concreto de Java se corresponde con el método **finalize**.

Cuando un objeto es destruido ocurren varias cosas: se llama al método **finalize** y después, el recolector de basura se encarga de eliminar el objeto, lo que conlleva liberar los recursos que dicho objeto tenga adjudicados, como por ejemplo, la memoria que ocupa.

Un objeto es destruido automáticamente cuando se eliminan todas las referencias al mismo. Una referencia a un objeto puede ser eliminada porque el flujo de ejecución salga fuera del ámbito donde ella está declarada, o porque explícitamente se le asigne el valor **null**.

Destructor

Un *destructor* es un método especial de una clase que se ejecuta antes de que un objeto de esa clase sea eliminado físicamente de la memoria. Un *destructor* se distingue fácilmente porque tiene el nombre predeterminado **finalize**. Cuando en una clase no especificamos un destructor, el compilador proporciona uno a través de la clase **Object** cuya sintaxis es la siguiente:

```
protected void finalize() throws Throwable { /* sin código */ }
```

Para definir un destructor en una clase tiene que reescribir el método anterior. A diferencia de lo que ocurría con los constructores, en una clase sólo es posible definir un *destructor*. En el cuerpo del mismo puede escribir cualquier operación que quiera realizar relacionada con el objeto que se vaya a destruir.

Resumiendo: un destructor es invocado automáticamente justo antes de que el objeto sea recolectado como basura por el *recolector de basura* de Java. Y ¿cuándo ocurre esto? Cuando no queden referencias al objeto.

Como ejemplo vamos a añadir a la clase *CFecha* del programa anterior, un destructor para que simplemente nos muestre un mensaje cada vez que se destruya un objeto de esa clase. Esto es:

```
public class CFecha
{
    // ...

    protected void finalize() throws Throwable // destructor
    {
        System.out.println("Objeto destruido");
    }
    // ...
}
```

Ejecute ahora la aplicación *Test* cuyo código se muestra a continuación y observe los resultados.

```
public class Test
{
    // Visualizar una fecha
```

```

public static void visualizarFecha(CFecha fecha)
{
    int[] f = new int[3];
    fecha.obtenerFecha(f);
    System.out.println(f[0] + "/" + f[1] + "/" + f[2]);
}

public static void main(String[] args)
{
    CFecha fechal = new CFecha(1, 3, 2002);
    // Empieza un bloque de código
    {
        CFecha fecha2 = new CFecha(fechal);
        visualizarFecha(fecha2);
    } // fin del bloque
    visualizarFecha(fechal);
}
}

```

Analizando este ejemplo, observamos que en el método **main** se crean dos objetos: uno al nivel del bloque de **main**, y otro local a un bloque interno a **main**. Por lo tanto, cada objeto quedará desreferenciado cuando el flujo de ejecución salga fuera del bloque en el que está definido, instante a partir del cual el recolector de basura puede recolectar esos objetos.

Observará que cuando finalice la ejecución del método **main** no se visualiza el mensaje “Objeto destruido” tantas veces como objetos hay. Esto significa que el recolector de basura, justo en este instante, no está en ejecución (sólo por una pequeña cantidad de tiempo); un poco más tarde, posiblemente cuando el sistema esté libre, el recolector de basura identificará los objetos que no tienen referencias y los eliminará liberando la memoria que ocupan.

No obstante, la aplicación Java puede finalizar sin que el recolector haya identificado todos los objetos que debía enviar a la basura y por lo tanto, los destructores correspondientes no se ejecutarán. En este caso, será el sistema operativo el que se encargue de liberar los recursos que fueron ocupados.

Si una clase tiene miembros que son objetos de otras clases, su destructor se ejecuta antes que los destructores de los objetos miembro. En otras palabras, el orden de destrucción es inverso al orden de construcción.

Un destructor también se puede llamar explícitamente así: *objeto.finalize()*.

Sin embargo, invocar a **finalize** no activa un objeto para que sea enviado a la basura. Sólo cuando se eliminan todas las referencias que apuntan al mismo, éste se marca como destruible.

Ejecutar el recolector de basura

El *recolector de basura* se ejecuta en un subproceso paralelamente a su aplicación limpiando la basura (objetos desreferenciados) en forma silenciosa y en segundo plano y nunca se detiene por más de una pequeña cantidad de tiempo.

Ahora bien, si desea forzar una completa recolección de basura (marcar y barrer), puede hacerlo llamando al método `gc` (*garbage collector*: recolector de basura) de la clase **System**. Por ejemplo:

```
public class Test
{
    // Visualizar una fecha
    public static void visualizarFecha(CFecha fecha)
    {
        int[] f = new int[3];

        fecha.obtenerFecha(f);
        System.out.println(f[0] + "/" + f[1] + "/" + f[2]);
    }

    public static void main(String[] args)
    {
        CFecha fechal = new CFecha(1, 3, 2002);
        // Empieza un bloque de código
        {
            CFecha fecha2 = new CFecha(fechal);
            visualizarFecha(fecha2);
            fecha2 = null;
            Runtime runtime = Runtime.getRuntime();
            runtime.gc();
            runtime.runFinalization();
        } // fin del bloque
        visualizarFecha(fechal);
    }
}
```

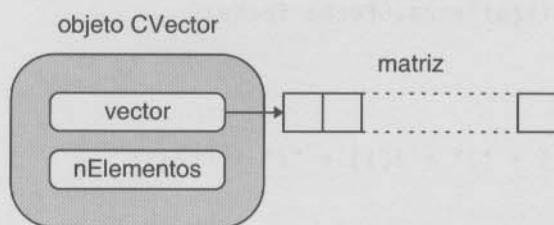
El ejemplo anterior fuerza la recolección de basura en el bloque de código interno a **main**. Si embargo, esto rara vez será necesario; a lo mejor, si acaba de liberar muchos objetos ya inservibles y quiere que se lleven pronto la basura.

REFERENCIAS COMO MIEMBROS DE UNA CLASE

Un miembro de una clase que sea una referencia requiere, generalmente, de una asignación de memoria, proceso que normalmente realizará el constructor. Sucede entonces que el espacio de memoria asignado es referenciado desde el objeto pe-

ro, lógicamente, no pertenece al objeto, lo que puede dar lugar a problemas si no se implementan adecuadamente los métodos que generan un objeto copia de otro de su misma clase. Un ejemplo de este tipo de clases es la clase *Círculo* expuesta al principio de este capítulo; recuerde que tenía un miembro *centro* de la clase *Punto*.

Para ver lo expuesto con detalle, vamos a escribir una clase *CVector* para construir objetos que representen matrices numéricas con un número cualquiera de elementos. Por lo tanto, sería inapropiado definir como miembro privado de la clase *CVector* una matriz con un número fijo de elementos. En su lugar, definiremos una referencia, *vector*, a una matriz de tipo **double**, por ejemplo, para después asignar dinámicamente la cantidad de memoria necesaria para la matriz.



Según lo expuesto, la funcionalidad de la clase *CVector* estará soportada por los atributos:

- *vector*: una referencia a una matriz de valores de tipo **double**.
- *nElementos*: número de elementos de dicha matriz.

```
public class CVector
{
    private double[] vector;
    private int nElementos;
    // ...
}
```

Y por los métodos:

- *constructores* para crear un objeto *CVector* con un número de elementos pre-determinado, con un número de elementos especificado, a partir de una matriz unidimensional, o bien a partir de otro objeto *CVector*.

El trabajo que tienen que realizar los constructores de la clase *CVector*, dependiendo de los casos, es asignar la memoria necesaria para la matriz de datos e iniciar dicha matriz con ceros (iniciación por omisión), con otra matriz o con otro vector, como podemos ver a continuación:

```

public CVector() // número de elementos por omisión: 10
{
    nElementos = 10;
    vector = new double[nElementos];
}

public CVector(int ne) // ne elementos
{
    if ( ne < 1 )
    {
        System.out.println("Nº de elementos no válido: " + ne);
        System.out.println("Se asignan 10 elementos");
        ne = 10;
    }
    nElementos = ne;
    vector = new double[nElementos];
}

public CVector(double[] m) // crea un CVector desde una matriz
{
    nElementos = m.length;
    vector = new double[nElementos];
    // Copiar los elementos de la matriz m
    for ( int i = 0; i < nElementos; i++ )
        vector[i] = m[i];
}

public CVector(CVector v) // constructor copia
{
    nElementos = v.nElementos;
    vector = new double[nElementos];
    // Copiar el objeto v
    for ( int i = 0; i < nElementos; i++ )
        vector[i] = v.vector[i];
}

```

Observar que este método además de copiar los atributos del objeto *v* en el objeto referenciado por **this**, copia también los valores de la matriz; si no hiciera esto último tendríamos una sola matriz referenciada por dos objetos.

- *copiar*: método que permite asignar un objeto *CVector* a otro. Observar que este método realiza el mismo proceso que el constructor copia; además, retorna una referencia al objeto resultante de la copia.

```

public CVector copiar(CVector v) // copia un CVector en otro
{
    nElementos = v.nElementos;
    vector = new double[nElementos];
    // Copiar el objeto v

```

```

        for ( int i = 0; i < nElementos; i++ )
            vector[i] = v.vector[i];
        return this;
    }
}

```

- *ponerValorEn*: método que permite asignar un dato al elemento especificado de un objeto *CVector*.

```
public void ponerValorEn( int i, double valor ) { vector[i] = valor; }
```

- *valorEn*: método que devuelve el dato almacenado en el elemento especificado de un objeto *CVector*.

```
public double valorEn( int i ) { return vector[i]; }
```

- *longitud*: método que devuelve el número de elementos de un objeto *CVector*.

```
public int longitud() { return nElementos; }
```

El resultado de encapsular los métodos anteriormente expuestos es la clase *CVector* que se muestra a continuación:

```

////////// Definición de la clase CVector
// Definición de la clase CVector
//
public class CVector
{
    private double[] vector; // matriz vector
    private int nElementos; // número de elementos de la matriz

    public CVector() // número de elementos por omisión
    {
        nElementos = 10;
        vector = new double[nElementos];
    }

    public CVector(int ne) // ne elementos
    {
        if ( ne < 1 )
        {
            System.out.println("Nº de elementos no válido: " + ne);
            System.out.println("Se asignan 10 elementos");
            ne = 10;
        }
        nElementos = ne;
        vector = new double[nElementos];
    }
}

```

```
public CVector(double[] m) // crea un CVector desde una matriz
{
    nElementos = m.length;
    vector = new double[nElementos];
    // Copiar los elementos de la matriz m
    for ( int i = 0; i < nElementos; i++ )
        vector[i] = m[i];
}

public CVector(CVector v) // constructor copia
{
    nElementos = v.nElementos;
    vector = new double[nElementos];
    // Copiar el objeto v
    for ( int i = 0; i < nElementos; i++ )
        vector[i] = v.vector[i];
}

public CVector copiar(CVector v) // copia un CVector en otro
{
    nElementos = v.nElementos;
    vector = new double[nElementos];
    // Copiar el objeto v
    for ( int i = 0; i < nElementos; i++ )
        vector[i] = v.vector[i];

    return this;
}

public void ponerValorEn( int i, double valor )
{
    if ( i >= 0 && i < nElementos )
        vector[i] = valor;
    else
        System.out.println("Índice fuera de límites");
}

public double valorEn( int i )
{
    if ( i >= 0 && i < nElementos )
        return vector[i];
    else
        System.out.println("Índice fuera de límites");
        return Double.NaN;
}

public int longitud() { return nElementos; }
```

El resultado es que cada objeto *CVector* consta de dos bloques de memoria, uno de tamaño fijo que almacena su estructura interna (*vector* y *nElementos*) y otro de longitud variable que almacena los datos (la matriz de tipo **double**).

Para probar la clase expuesta escriba, por ejemplo, la siguiente aplicación:

```
///////////////////////////////
// Aplicación que utiliza la clase CVector
//
public class Test
{
    // Visualizar un vector
    public static void visualizarVector(CVector v)
    {
        int ne = v.longitud();
        for (int i = 0; i < ne; i++)
            System.out.print(v.valorEn(i) + " ");
        System.out.println();
    }

    public static void main(String[] args)
    {
        CVector vector1 = new CVector(5);
        visualizarVector(vector1);

        CVector vector2 = new CVector();
        for (int i = 0; i < vector2.longitud(); i++)
            vector2.ponerValorEn(i, (i+1)*10);
        visualizarVector(vector2);

        CVector vector3 = new CVector(vector2);
        visualizarVector(vector3);

        double x[] = { 1, 2, 3, 4, 5, 6, 7 }; // matriz x
        CVector vector4 = new CVector(x);
        visualizarVector(vector4);

        System.out.println("Fin de la aplicación");
    }
}
```

Analizando a grandes rasgos el código presentado anteriormente, podemos ver que la línea:

```
CVector vector1 = new CVector(5);
```

llama al constructor *CVector(int ne)* y crea un objeto *vector1* con 5 elementos.
Las líneas:

```
CVector vector2 = new CVector();
```

```
for (int i = 0; i < vector2.longitud(); i++)
    vector2.ponerValorEn(i, (i+1)*10);
```

llaman al constructor *CVector* sin argumentos y crea un objeto *vector2* con 10 elementos por omisión. Después asigna valores a cada uno de los elementos de *vector2*. La línea:

```
CVector vector3 = new CVector(vector2);
```

llama al constructor copia y crea un objeto *vector3* iniciado con los datos del objeto *vector2*. Las líneas:

```
double x[] = { 1, 2, 3, 4, 5, 6, 7 }; // matriz x
CVector vector4 = new CVector(x);
```

la primera define la matriz *x* y la última llama al constructor *CVector(double[] m)* y crea un objeto *vector4* iniciado con los datos de la matriz *x*.

Como se puede observar, cada vez que se crea un objeto es llamado automáticamente un constructor, lo que garantiza la iniciación del objeto. El que se llame a uno o a otro constructor, depende del número y tipo de argumentos especificados.

Cuando el flujo de ejecución sale fuera del ámbito donde ha sido definido un objeto *CVector*, el recolector de basura marcará y barrerá tanto el objeto como la matriz referenciada por el mismo, liberando la memoria ocupada.

Sin embargo, una clase con miembros que son referencias a otros objetos, como es *CVector*, potencialmente tiene problemas. Para comprobarlo, suponga que al diseñador de la clase *CVector* se le hubiera ocurrido escribir el constructor copia así:

```
public CVector(CVector v) // constructor copia
{
    nElementos = v.nElementos;
    vector = v.vector;
}
```

Suponga también que en la aplicación anterior el método **main** fuera como sigue:

```
public static void main(String[] args)
{
    double x[] = { 1, 2, 3, 4, 5, 6, 7 }; // matriz x
    CVector vector1 = new CVector(x);
    visualizarVector(vector1); // escribe 1 2 3 4 5 6 7

    // El siguiente bloque define vector2
    {
        CVector vector2 = new CVector(vector1);
```

```

        for (int i = 0; i < vector2.longitud(); i++)
            vector2.ponerValorEn(i, vector2.valorEn(i)*10);
        visualizarVector(vector2); // escribe 10 20 30 40 50 60 70
    }
    // vector2 ha sido destruido
    visualizarVector(vector1); // escribe 10 20 30 40 50 60 70
}
System.out.println("Fin de la aplicación");
}

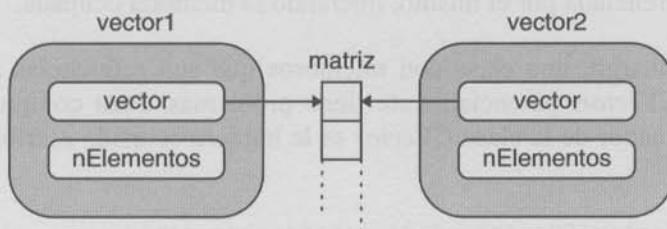
```

Ahora el método **main** crea un objeto *vector1* iniciado con los valores de una matriz *x* e incluye un bloque que crea un nuevo objeto *vector2* a partir de *vector1*, para lo cual se invoca al constructor copia.

Observe que ahora este constructor simplemente copia los atributos del objeto *v* en los correspondientes atributos del nuevo objeto creado. Por lo tanto, el resultado de una sentencia como:

```
CVector vector2 = new CVector(vector1);
```

será dos objetos, *vector1* y *vector2*, referenciando la misma matriz. La figura siguiente muestra esto con claridad:



Esto significa que cualquier modificación en uno de los objetos afectará a ambos, justo lo que sucede cuando se ejecuta el código siguiente. Las modificaciones realizadas en el objeto *vector2* afectan de la misma forma a *vector1*:

```

// El siguiente bloque define vector2
{
    CVector vector2 = new CVector(vector1);
    for (int i = 0; i < vector2.longitud(); i++)
        vector2.ponerValorEn(i, vector2.valorEn(i)*10);
    visualizarVector(vector2);
}

```

Piense ahora qué sucederá cuando el flujo de ejecución salga fuera del ámbito de *vector2*. Pues que el objeto *vector2* será enviado a la basura y eliminado por el recolector de basura. ¿Será enviado también a la basura el objeto *matriz* referen-

ciado por el atributo *vector* de *vector2*? No, porque dicho objeto matriz tiene aún una referencia: *vector* de *vector1*.

Esta misma teoría es aplicable al método *copiar*. Esto significa que debemos poner un especial interés cuando escribamos métodos que tengan como finalidad duplicar objetos que tienen atributos que son referencias a otros objetos.

COMPARAR OBJETOS

Según vimos en el capítulo anterior, la clase **Object** es la clase raíz de la jerarquía de clases de la biblioteca Java y de cualquier otra clase que implementemos en nuestras aplicaciones, lo que se traduce en que todas ellas heredan los métodos de **Object**, como **equals**, **toString** o **finalize**, por ejemplo.

¿Cómo han sido implementados estos métodos? Pues de una forma muy genérica, sin pensar en ningún objeto en particular. Por ejemplo, **equals** proporciona el mismo resultado que el operador “==”; esto es, compara las referencias a los objetos, no sus contenidos, lo cual es lógico: no podemos comparar dos objetos que aún no sabemos cómo son. Ahora bien, una vez diseñada una clase como *CVector*, si necesitamos que el método **equals** nos diga cómo es un objeto *CVector* con respecto a otro, tenemos que sobreescribir dicho método.

Método **equals**

Como ejemplo, añada la definición del método **equals** a la clase *CVector*. Para poder escribir este método, primero debe responder a la siguiente pregunta: ¿cuándo dos objetos *CVector* son iguales? La respuesta es, cuando contengan los mismos valores; esto es, cuando las matrices que representan sean idénticas. Basta entonces con el método **equals** de la clase *CVector* compare las matrices de los dos objetos a comparar:

```
public boolean equals(CVector v)
{
    return Arrays.equals(vector, v.vector);
}
```

El código anterior implica importar la clase **Arrays** del paquete **java.util**. Para probar los resultados que podemos obtener a partir de este método a diferencia de los obtenidos por el operador “==” escriba la siguiente aplicación:

```
///////////
// Aplicación que utiliza la clase CVector
//
```

```

public class Test
{
    // Visualizar un vector
    public static void visualizarVector(CVector v)
    {
        int ne = v.longitud();
        for (int i = 0; i < ne; i++)
            System.out.print(v.valorEn(i) + " ");
        System.out.println();
    }

    public static void main(String[] args)
    {
        double x[] = { 1, 2, 3, 4, 5, 6, 7 }; // matriz x
        CVector vector1 = new CVector(x);
        visualizarVector(vector1); // escribe 1 2 3 4 5 6 7

        CVector vector2 = new CVector(vector1);
        for (int i = 0; i < vector2.longitud(); i++)
            vector2.ponerValorEn(i, vector2.valorEn(i)*10);
        visualizarVector(vector2); // escribe 10 20 30 40 50 60 70

        if (vector1 == vector2)
            System.out.println("referencias al mismo objeto");
        else
            System.out.println("referencias a objetos diferentes");

        if (vector1.equals(vector2))
            System.out.println("objetos iguales");
        else
            System.out.println("objetos diferentes");
    }
}

```

Si ejecuta la aplicación *Test* anterior, obtendrá los siguientes resultados:

```

1.0 2.0 3.0 4.0 5.0 6.0 7.0
10.0 20.0 30.0 40.0 50.0 60.0 70.0
referencias a objetos diferentes
objetos diferentes

```

MIEMBROS STATIC DE UNA CLASE

Este tema ya fue introducido en el capítulo 4. Por lo tanto, el propósito ahora que ya tiene un mayor conocimiento de la POO es abundar en detalles con el fin de dejar suficientemente claro cuál es la utilidad de estos miembros.

Atributos static

La última versión de la clase *Círculo* definida al principio de este capítulo declaraba dos atributos: *centro* y *radio*. Esto se traduce en que cada objeto de la clase, cada círculo, tiene su propia copia de esos dos atributos. Pero seguro que en más de una ocasión querremos utilizar un atributo (una variable) del cual exista una única copia que pueda ser utilizada por todos los objetos de la misma clase; esto es, una variable con ámbito global. El problema es que Java no permite declarar variables globales tal como se interpretan en otros lenguajes de programación; cada variable en Java debe ser declarada dentro de una clase, la cual define su propio ámbito. La alternativa que Java ofrece para dar solución al problema planteado es declarar el atributo **static**.

Un atributo **static** no es un atributo específico de un objeto (el *radio* si es un atributo específico de un círculo; cada círculo tiene su radio), sino más bien es un atributo de la clase; esto es, un atributo del que sólo hay una copia que comparten todos los objetos de la clase. Por esta razón, un atributo **static** existe y puede ser utilizado aunque no exista ningún objeto de la clase.

Como ejemplo, supongamos que queremos por una parte, no tener que acceder a la constante **PI** de la clase **Math** cada vez que calculemos el área del círculo o la longitud de la circunferencia, y por otra, conocer el número de objetos *Círculo* que hay creados en cada instante. Para hacer esto, obviamente es más eficiente asociar dos atributos con la clase, *pi* y *numCírculos*, que con cada objeto. El código mostrado a continuación muestra cómo añadir estos atributos a la clase:

```
class Punto
{
    private double x, y;

    Punto(double cx, double cy)
    {
        x = cx; y = cy;
    }
}

public class Círculo
{
    // Atributos
    private static double pi = 3.141592;
    public static int numCírculos;

    private Punto centro; // coordenadas del centro
    private double radio; // radio del círculo

    // Métodos
}
```

```

protected void msgEsNegativo()
{
    System.out.println("El radio es negativo. Se convierte a positivo");
}

public Círculo() // constructor sin parámetros
{
    this(100.0, 100.0, 100.0);
}

public Círculo(double cx, double cy, double r) // constructor
{
    centro = new Punto(cx, cy);
    if (r < 0)
    {
        msgEsNegativo();
        r = -r;
    }
    radio = r;
    numCírculos++;
}

public double longCircunferencia()
{
    return 2 * pi * radio;
}

public double áreaCírculo()
{
    return pi * radio * radio;
}
}

```

Un atributo **static** puede ser calificado como **private** (privado), **protected** (protegido), **public** (público), o no calificado (acceso predeterminado). Asimismo, podemos calificarlo **final** para que sea una constante en lugar de una variable.

Acceder a los atributos static

En el apartado anterior podemos ver cómo los métodos de la clase *Círculo* pueden acceder directamente a los atributos *numCírculos* y *pi* de la misma, igual que acceden al resto de los atributos. Pero, desde otra clase ¿cómo podemos acceder a esa información? Puesto que *numCírculos* es una variable **static** declarada **public** podemos acceder a ella directamente a través del nombre de la clase (utilizar el nombre de un objeto, aunque es válido, puede dar lugar a malas interpretaciones del código). Por ejemplo:

```

public class Test
{
    public static void main(String[] args)
    {
        Círculo obj1 = new Círculo();
        System.out.println(obj1.longCircunferencia());
        System.out.println(obj1.áreaCírculo());

        Círculo obj2 = new Círculo(100, 100, 10);
        System.out.println(obj2.longCircunferencia());
        System.out.println(obj2.áreaCírculo());

        System.out.println(Círculo.numCírculos);
    }
}

```

Observe que para acceder a la información proporcionada por el atributo *numCírculos* se utiliza el nombre de su clase y no el de un objeto de la misma. En cambio, desde la clase *Test* no se puede acceder al atributo *pi* porque es privado.

Anteriormente dijimos que Java no permite declarar variables globales. No obstante, *Círculo.numCírculos* se comporta igual que si lo fuera, ya que utilizando esta sintaxis podemos acceder a *numCírculos* desde cualquier otra clase.

Métodos static

Un método declarado **static** carece de la referencia **this** por lo que no puede ser invocado para un objeto de su clase, sino que se invoca en general allí donde se necesite utilizar la operación para la que ha sido escrito. Desde este punto de vista es imposible que un método **static** pueda acceder a un miembro no **static** de su clase; por la misma razón, sí puede acceder a un miembro **static**. Como ejemplo, recuerde la clase **Math** estudiada en el capítulo 5; uno de sus métodos es **sqrt** y la forma de invocarlo desde cualquier método de otra clase es: *Math.sqrt(n)*. Como vemos, utilizamos esta expresión para invocar al método **sqrt** de la clase **Math** y calcular la raíz cuadrada de *n* sin pensar en ningún objeto en particular.

Como ejemplo, vamos a añadir a la clase *Círculo* un método *cambiarPrecisiónPiA* que permite cambiar la precisión de *pi* siempre que su valor se mantenga entre 3.14 y 3.1416.

```

public static void cambiarPrecisiónPiA(double nuevoValor)
{
    if (nuevoValor < 3.14 || nuevoValor > 3.1416) return;
    pi = nuevoValor;
}

```

Un método **static** puede acceder a los miembros (atributo o método) **static** de su clase pero no puede acceder a los miembros **no static**. Por ejemplo, si en el método anterior intenta establecer el atributo *radio* a 0, el compilador le mostrará un error indicándole que no se puede hacer referencia a una variable no estática desde un método estático.

Por otra parte, un miembro **static** sí puede ser accedido por un método independientemente de que sea **static** o no. Por ejemplo, el miembro *pi* de la clase *Círculo* es accedido por los métodos no estáticos *longCircunferencia* y *áreaCírculo* y por el método estático *cambiarPrecisiónPiA* de su misma clase. Si el acceso se hace desde un método de otra clase, dicho miembro tiene que ser invocado a través del nombre de la clase según se explicó anteriormente. Por ejemplo:

```
public class Test
{
    public static void main(String[] args)
    {
        Círculo obj1 = new Círculo();
        System.out.println(obj1.longCircunferencia());
        System.out.println(obj1.áreaCírculo());

        Círculo.cambiarPrecisiónPiA(3.14);
        Círculo obj2 = new Círculo(100, 100, 10);
        System.out.println(obj2.longCircunferencia());
        System.out.println(obj2.áreaCírculo());

        System.out.println(Círculo.numCírculos);
    }
}
```

Se puede observar que el comportamiento de *Círculo.cambiarPrecisiónPiA* es igual que el de cualquier otro método de un lenguaje no orientado a objetos. Esto hace posible escribir programas Java utilizando solamente esta clase de métodos, pero entonces se frustraría el propósito más importante de este lenguaje: la POO. No piense por ello que utilizar este tipo de métodos es una trampa. Hay muchas y buenas razones para utilizarlos y sino observe la utilidad de las clases **Math** y **System** en las que todos sus métodos son estáticos.

Iniciador estático

Sabemos que tanto los atributos del objeto como los de la clase pueden ser iniciados en la propia declaración. Sirva como ejemplo el atributo *pi* de la clase *Círculo*. Ahora, mientras que los atributos de la clase son iniciados cuando la clase es cargada por primera vez, los atributos del objeto son iniciados para cada objeto en el instante de su creación.

En más de una ocasión necesitaremos iniciaciones más complejas que esas que podemos hacer en la propia declaración. Para esto podemos utilizar alguno de los constructores de la clase. Pero en el caso de que la clase tenga atributos estáticos, los constructores sólo serían adecuados cuando la iniciación de esos atributos no sea requerida antes de que se cree un primer objeto. Si no es así, será necesario añadir a la clase un *iniciador estático* cuya sintaxis es la siguiente:

```
static
{
    // iniciación de los atributos de la clase
}
```

Un *iniciador estático* es un método anónimo que no tiene parámetros, no retorna ningún valor, y es invocado automáticamente por el sistema cuando se carga la clase.

Como ejemplo, vamos a añadir a la clase *Círculo* dos atributos **static**, *seno* y *coseno*, que proporcionen las tablas del seno y coseno de grado en grado. Dichos atributos serán iniciados a través de un iniciador estático como se puede observar a continuación:

```
public class Círculo
{
    // Atributos
    private static double pi = 3.141592;
    public static int numCírculos;
    public static double seno[] = new double[360];
    public static double coseno[] = new double[360];
    // Iniciador estático
    static
    {
        // Tablas del seno y coseno de grado en grado
        for (int i = 0; i < 360; i++)
        {
            double s, c;
            // Calcular el seno y el coseno de i
            s = Math.sin(Math.toRadians(i));
            c = Math.cos(Math.toRadians(i));
            // Almacenar los valores redondeados a 6 decimales
            seno[i] = Math.rint(s*1000000)/1000000;
            coseno[i] = Math.rint(c*1000000)/1000000;
        }
    }
    // ...
}
```

Java permite cualquier número de iniciadores estáticos aunque el compilador finalmente los fusionará en uno sólo en el mismo orden en el que aparezcan en la definición de la clase. Este iniciador se ejecutará solamente una vez: cuando el sistema cargue la clase por primera vez.

MATRICES DE OBJETOS

Se puede crear una matriz de objetos de cualquier clase, de la misma forma que se crea una matriz de números, de caracteres, de objetos **String**, etc. Por ejemplo, suponiendo que tenemos definida una clase *CPersona* podemos definir la matriz *listaTeléfonos* con 100 elementos de la forma siguiente:

```
CPersona[] listaTeléfonos = new CPersona[100];
```

listaTeléfonos es una matriz de referencias a objetos de la clase *CPersona*. Cada elemento de esta matriz será iniciado por Java con el valor **null**, indicando así que la matriz inicialmente no referencia a ningún objeto *CPersona*; esto es, la matriz está vacía.

Una vez creada la matriz, para asignar un objeto al elemento *i* de la misma se puede utilizar una línea de código como la siguiente:

```
listaTeléfonos[i] = new CPersona([argumentos]);
```

Como ejemplo, supongamos que deseamos mantener una lista de teléfonos. La lista será un objeto que encapsule la matriz de objetos persona, y muestre una interfaz que permita añadir, eliminar y buscar una en la lista.

En un primer análisis sobre el enunciado identificamos dos clases de objetos: personas y lista de teléfonos.

La clase de objetos persona (que denominaremos *CPersona*) encapsulará el nombre, la dirección y el teléfono de cada una de las personas de la lista; asimismo proporcionará la funcionalidad necesaria para establecer u obtener los datos de cada persona individual.

El listado siguiente muestra un ejemplo de una clase *CPersona* que define los atributos privados *nombre*, *dirección* y *teléfono* relativos a una persona, y los métodos públicos que forman la interfaz de esta clase de objetos:

- Constructores, con y sin argumentos, para iniciar un objeto persona.
- Métodos de acceso (*asignar...* y *obtener...*) para cada uno de los atributos.

```
//////////  
// Definición de la clase CPersona  
//  
public class CPersona  
{  
    // Atributos  
    private String nombre;  
    private String dirección;  
    private long teléfono;  
  
    // Métodos  
    public CPersona() {}  
    public CPersona(String nom, String dir, long tel)  
    {  
        nombre = nom;  
        dirección = dir;  
        teléfono = tel;  
    }  
  
    public void asignarNombre(String nom)  
    {  
        nombre = nom;  
    }  
  
    public String obtenerNombre()  
    {  
        return nombre;  
    }  
  
    public void asignarDirección(String dir)  
    {  
        dirección = dir;  
    }  
  
    public String obtenerDirección()  
    {  
        return dirección;  
    }  
  
    public void asignarTeléfono(long tel)  
    {  
        teléfono = tel;  
    }  
  
    public long obtenerTeléfono()  
    {  
        return teléfono;  
    }  
}
```

Un método como *asignarNombre* simplemente asigna el nombre pasado como argumento al atributo *nombre* del objeto que recibe el mensaje. Y un método como *obtenerNombre* devuelve el atributo *nombre* del objeto que recibe el mensaje. La explicación para los otros métodos es análoga. Por ejemplo:

```
CPersona obj = new CPersona();
obj.asignarNombre("Javier");
System.out.println(obj.obtenerNombre()); // escribe: Javier
```

El listado siguiente muestra un ejemplo de lo que puede ser la clase lista de teléfonos, que denominaremos *CListaTfnos*. Define los atributos privados *listaTeléfonos*, matriz de objetos *CPersona*, y *nElementos*, número de elementos de la matriz, y los métodos que se describen a continuación:

```
////////////////////////////////////////////////////////////////
// Definición de la clase CListaTfnos.
//
public class CListaTfnos
{
    private CPersona[] listaTeléfonos; // matriz de objetos
    private int nElementos; // número de elementos de la matriz

    private void unElementoMás(CPersona[] listaActual) { ... }
    private void unElementoMenos(CPersona[] listaActual) { ... }
    public CListaTfnos() { ... } // constructor
    public void ponerValorEn( int i, CPersona objeto ) { ... }
    public CPersona valorEn( int i ) { ... }
    public int longitud(){ ... }
    public void añadir(CPersona obj) { ... }
    public void eliminar(long tel) { ... }
    public int buscar(String str, int pos) { ... }
}
```

Para crear un objeto lista de teléfonos escribiremos una línea de código como la siguiente:

```
CListaTfnos listatfnos = new CListaTfnos();
```

Según este ejemplo, la clase *CListaTfnos* tiene que tener un constructor sin argumentos ¿Qué debe hacer este constructor? Iniciar un objeto *CListaTfnos* con una matriz *listaTeléfonos* con 0 elementos:

```
public CListaTfnos()
{
    // Crear una lista vacía
    nElementos = 0;
    listaTeléfonos = new CPersona[nElementos];
}
```

Antes de que se ejecute el cuerpo del constructor anterior, *nElementos* vale 0 y *listaTeléfonos* null; y después de que se ejecute, *nElementos* sigue valiendo 0 y *listaTeléfonos* referencia una matriz de longitud 0 (propiedad **length** = 0).

El código que escribiremos para añadir un teléfono (objeto *CPersona*) a la lista de teléfonos (objeto *CListaTfnos*) será análogo al siguiente:

```
listatfnos.añadir(new CPersona(nombre, dirección, teléfono));
```

Cuando el objeto *listatfnos* de la clase *CListaTfnos* recibe el mensaje *añadir*, responde ejecutando su método *añadir* que incrementará en uno el tamaño del atributo matriz *listaTeléfonos* y asignará a este nuevo elemento el objeto *CPersona* pasado como argumento. Para realizar estas dos tareas añadiremos a la clase *CListaTfnos* los métodos *unElementoMás* y *ponerValorEn*.

```
public void añadir(CPersona obj)
{
    unElementoMás(listaTeléfonos);
    ponerValorEn( nElementos - 1, obj );
}
```

Observamos que cuando se invoca al método *unElementoMás* se pasa como argumento la lista de teléfonos actual que ahora quedará referenciada por su parámetro *listaActual* ¿Qué tiene que hacer este método? Pues, asignar al atributo *listaTeléfonos* un nuevo espacio de memoria que permita albergar un elemento más de los que tiene actualmente, copiar uno a uno los elementos que tenía hasta ahora la matriz y que están referenciados por *listaActual*, e incrementar el atributo *nElementos*. Observe que el bloque de memoria viejo quedará desreferenciado cuando el flujo de control salga fuera del método *unElementoMás*, por ser el parámetro *listaActual* local ¿Quién liberará ese bloque de memoria y los bloques de memoria de los objetos referenciados por él? De esta tarea se encarga el recolector de basura de Java.

```
private void unElementoMás(CPersona[] listaActual)
{
    nElementos = listaActual.length;
    listaTeléfonos = new CPersona[nElementos + 1];
    // Copiar la lista actual
    for ( int i = 0; i < nElementos; i++ )
        listaTeléfonos[i] = listaActual[i];
    nElementos++;
}
```

El método *ponerValorEn* tiene como misión asignar la referencia a un nuevo objeto *CPersona*, al elemento *i* de la matriz *listaTeléfonos*. Ambos datos, objeto y posición, son pasados como argumentos.

```

public void ponerValorEn( int i, CPersona objeto )
{
    if ( i >= 0 && i < nElementos)
        listaTeléfonos[i] = objeto;
    else
        System.out.println("Índice fuera de límites");
}

```

El código que escribiremos para eliminar un teléfono (objeto *CPersona*) de la lista de teléfonos (objeto *CListaTfnos*) será análogo al siguiente:

```
eliminado = listatfnos.eliminar(teléfono);
```

Cuando el objeto *listatfnos* de la clase *CListaTfnos* recibe el mensaje *eliminar*, responde ejecutando su método *eliminar* que quitará de la lista el elemento correspondiente al teléfono pasado como argumento y decrementará en uno el tamaño de la lista. Para realizar estas dos tareas, primero buscará en la matriz *listaTeléfonos* el elemento que referencia al objeto *CPersona* que tiene el número de teléfono pasado como argumento y asignará a este elemento el valor **null** (de esta forma, el objeto *CPersona* será enviado a la basura y recolectado por el recolector de basura); después, invocará al método *unElementoMenos* para quitar ese elemento de la lista. El método *eliminar* devuelve **true** si se encontró y eliminó el elemento especificado y **false** en caso contrario.

```

public boolean eliminar(long tel)
{
    // Buscar el teléfono y eliminar registro
    for ( int i = 0; i < nElementos; i++ )
        if ( listaTeléfonos[i].obtenerTeléfono() == tel )
    {
        listaTeléfonos[i] = null;
        unElementoMenos(listaTeléfonos);
        return true;
    }
    return false;
}

```

Observamos que cuando se invoca al método *unElementoMenos* se pasa como argumento la lista de teléfonos actual que ahora quedará referenciada por su parámetro *listaActual*. ¿Qué tiene que hacer este método? Pues, asignar al atributo *listaTeléfonos* un nuevo espacio de memoria que permita albergar un elemento menos de los que tiene actualmente, copiar uno a uno los elementos que tenía hasta ahora la matriz (referenciados por *listaActual*) menos el que tiene asignado la referencia **null** y decrementar el atributo *nElementos*. Cuando la ejecución de este método finalice, el bloque de memoria viejo referenciado por *listaActual* será enviado a la basura y recolectado por el recolector de basura.

```

private void unElementoMenos(CPersona[] listaActual)
{
    if (listaActual.length == 0) return;
    int k = 0;
    nElementos = listaActual.length;
    listaTeléfonos = new CPersona[nElementos - 1];
    // Copiar la lista actual
    for (int i = 0; i < nElementos; i++)
        if (listaActual[i] != null)
            listaTeléfonos[k++] = listaActual[i];
    nElementos--;
}

```

El código que escribiremos para buscar un teléfono (objeto *CPersona*) en la lista de teléfonos (objeto *CListaTfnos*) será análogo al siguiente:

```
pos = listatfnos.buscar(cadenabuscar, posición_inicio_búsqueda);
```

Cuando el objeto *listatfnos* de la clase *CListaTfnos* recibe el mensaje *buscar*, responde ejecutando su método *buscar* que recorrerá la lista de teléfonos en busca de un elemento (objeto *CPersona* referenciado) que contenga en su campo *nombre* la subcadena pasada como argumento. La búsqueda se iniciará en la posición pasada como argumento. El método *buscar* devolverá la posición del elemento buscado, si se encuentra, o -1 en caso contrario.

```

public int buscar(String str, int pos)
{
    String nom;
    if (str == null) return -1;
    if (pos < 0) pos = 0;
    for (int i = pos; i < nElementos; i++)
    {
        nom = listaTeléfonos[i].obtenerNombre();
        if (nom == null) continue;
        // iStr está contenida en nom?
        if (nom.indexOf(str) > -1)
            return i;
    }
    return -1;
}

```

Otros métodos de interés son *valorEn* y *longitud*. El método *valorEn* devuelve el objeto *CPersona* referenciado por el elemento *i* de la matriz *listaTeléfonos*.

```

public CPersona valorEn( int i )
{
    if (i >= 0 && i < nElementos)
        return listaTeléfonos[i];
}

```

```

    else
    {
        System.out.println("Índice fuera de límites");
        return null;
    }
}
}

```

El método *longitud* devuelve el número de elementos que tiene actualmente la matriz *listaTeléfonos*.

```
public int longitud() { return nElementos; }
```

Hasta aquí, el diseño de la clase *CPersona* y *CListaTfnos*. El siguiente paso será escribir una aplicación que se ejecute así:

1. Buscar
2. Buscar siguiente
3. Añadir
4. Eliminar
5. Salir

```

Opción: 3
nombre: Javier
dirección: Santander
teléfono: 942232323

```

1. Buscar
2. Buscar siguiente
3. Añadir
4. Eliminar
5. Salir

```
Opción:
```

A la vista del resultado anterior, esta aplicación mostrará un menú que presentará las operaciones que se pueden realizar sobre la lista de teléfonos. Posteriormente, la operación elegida será identificada por una sentencia **switch** y procesada de acuerdo al esquema presentado a continuación:

```

public class Test
{
    public static int menú() { ... }

    public static void main(String[] args)
    {
        // Definir un flujo de caracteres de entrada y otro de salida
    }
}

```

```

// Crear un objeto lista de teléfonos vacío
CListaTfnos listatfnos = new CListaTfnos();
do
{
    opción = menú();
    switch (opción)
    {
        case 1: // buscar
            // Buscar un elemento que contenga "cadenabuscar".
            // Esta subcadena será obtenida del teclado.
            pos = listatfnos.buscar(cadenabuscar, 0);
            // Si se encuentra, mostrar sus datos
            break;
        case 2: // buscar siguiente
            // Buscar el siguiente elemento que contenga la subcadena
            // utilizada en la última búsqueda.
            pos = listatfnos.buscar(cadenabuscar, pos + 1);
            // Si se encuentra, mostrar sus datos.
            break;
        case 3: // añadir
            // Obtener del teclado los datos nombre, dirección y
            // teléfono del nuevo elemento a añadir, y añadirlo.
            listatfnos.añadir(new CPersona(nombre, dirección, teléfono));
            break;
        case 4: // eliminar
            // Obtener del teclado el número de teléfono a eliminar y
            // eliminarlo de la lista.
            eliminado = listatfnos.eliminar(teléfono);
            break;
        case 5: // salir
            listatfnos = null;
    }
}
while(opción != 5);
}
}

```

El listado completo de la aplicación *Test* se muestra a continuación:

```

import java.io.*;
/////////////////////////////////////////////////////////////////
// Aplicación para trabajar con matrices de objetos
//
public class Test
{
    public static int menú()
    {
        System.out.print("\n\n");
        System.out.println("1. Buscar");
        System.out.println("2. Buscar siguiente");
        System.out.println("3. Añadir");
        System.out.println("4. Eliminar");
        System.out.println("5. Salir");
    }
}

```

```
System.out.println("2. Buscar siguiente");
System.out.println("3. Añadir");
System.out.println("4. Eliminar");
System.out.println("5. Salir");
System.out.println();
System.out.print("    Opción: ");
int op;
do
    op = Leer.datoInt();
    while (op < 1 || op > 5);
    return op;
}

public static void main(String[] args)
{
    // Definir un flujo de caracteres de entrada: flujoE
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader flujoE = new BufferedReader(isr);
    // Definir una referencia al flujo estándar de salida: flujoS
    PrintStream flujoS = System.out;

    // Crear un objeto lista de teléfonos vacío (con cero elementos)
    CListaTfnos listatfnos = new CListaTfnos();

    int opción = 0, pos = -1;
    String cadenabuscar = null;
    String nombre, dirección;
    long teléfono;
    boolean eliminado = false;

    do
    {
        try
        {
            opción = menú();
            switch (opción)
            {
                case 1: // buscar
                    flujoS.print("conjunto de caracteres a buscar ");
                    cadenabuscar = flujoE.readLine();
                    pos = listatfnos.buscar(cadenabuscar, 0);
                    if (pos == -1)
                        if (listatfnos.longitud() != 0)
                            flujoS.println("búsqueda fallida");
                        else
                            flujoS.println("lista vacía");
                    else
                        flujoS.println(listatfnos.valorEn(pos).obtenerNombre());
                        flujoS.println(listatfnos.valorEn(pos).obtenerDirección());
            }
        }
    }
}
```

```

        flujoS.println(listatfnos.valorEn(pos).obtenerTeléfono());
    }
    break;
case 2: // buscar siguiente
    pos = listatfnos.buscar(cadenabuscar, pos + 1);
    if (pos == -1)
        if (listatfnos.longitud() != 0)
            flujoS.println("búsqueda fallida");
        else
            flujoS.println("lista vacía");
    else
    {
        flujoS.println(listatfnos.valorEn(pos).obtenerNombre());
        flujoS.println(listatfnos.valorEn(pos).obtenerDirección());
        flujoS.println(listatfnos.valorEn(pos).obtenerTeléfono());
    }
    break;
case 3: // añadir
    flujoS.print("nombre:    "); nombre = flujoE.readLine();
    flujoS.print("dirección: "); dirección = flujoE.readLine();
    flujoS.print("teléfono:  "); teléfono = Leer.datoLong();
    listatfnos.añadir(new CPersona(nombre, dirección, teléfono));
    break;
case 4: // eliminar
    flujoS.print("teléfono: "); teléfono = Leer.datoLong();
    eliminado = listatfnos.eliminar(teléfono);
    if (eliminado)
        flujoS.println("registro eliminado");
    else
        if (listatfnos.longitud() != 0)
            flujoS.println("teléfono no encontrado");
        else
            flujoS.println("lista vacía");
    break;
case 5: // salir
    listatfnos = null;
}
catch (IOException ignorada) {}
if (opción != 5);

```

PAQUETES

En el capítulo 4 ya fue expuesto el concepto de paquete. Si recuerda, dijimos que un paquete es un conjunto de clases, lógicamente relacionadas entre sí, agrupadas bajo un nombre; incluso, un paquete puede contener a otros paquetes.

También vimos que la propia biblioteca de clases de Java estaba organizada en paquetes dispuestos jerárquicamente. La jerarquía a la que nos referimos es análoga a la estructura jerárquica de carpetas o directorios que utilizamos para organizar los ficheros en un disco duro.

Asimismo sabemos que para referirnos a una *clase* de un paquete, tenemos que hacerlo anteponiendo al nombre de la misma el nombre de su paquete, excepto cuando el paquete haya sido importado explícitamente, como se indica en el siguiente ejemplo, o implícitamente (caso del paquete **java.lang**). Por ejemplo, la aplicación *Test* anterior utiliza, entre otras, la clase **InputStreamReader** del paquete **java.io**. Debido a que la aplicación incluye la línea de código:

```
import java.io.*;
```

podemos referirnos a esa clase simplemente por su nombre. En otro caso, tendríamos que haber utilizado su nombre completo: **java.io.InputStreamReader**.

Resumiendo: los paquetes ayudan a organizar las clases en grupos para facilitar el acceso a las mismas cuando las necesitemos en un programa; reducen los conflictos de nombres (lógicamente, la probabilidad de que dos nombres coincidan será menor cuantos más elementos intervengan); y permiten proteger las clases (una clase con nivel de protección de paquete, clase no pública, no está disponible para otros paquetes, ni siquiera para los subpaquetes).

Crear un paquete

Para crear un paquete hay que seguir básicamente los pasos indicados a continuación:

1. *Seleccionar el nombre del paquete.* Para nombrar un paquete, *Sun Microsystems* recomienda utilizar el nombre de su dominio de Internet, pero con los elementos a la inversa. Por ejemplo, si Sun hubiera seguido esta recomendación en todos los casos, todos sus paquetes empezarían por **com.sun.java** ya que el dominio de Internet para Java es **java.sun.com**. Puede alargar el nombre para describir genéricamente las clases del paquete; por ejemplo: **com.sun.java.swing**. La idea que se persigue es la exclusividad del nombre del paquete, con el fin de no causar conflictos con los paquetes de otros.

Realicemos un ejemplo para practicar. Dejando ahora a un lado el dominio de Internet, supongamos que deseamos crear los paquetes:

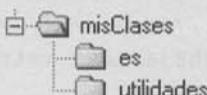
```
misClases.es  
misClases.utilidades
```

2. *Crear una estructura jerárquica de carpetas en el disco duro.* Hemos dicho que la biblioteca de clases de Java está organizada en paquetes dispuestos jerárquicamente. Pues bien, esta estructura jerárquica se hace corresponder en el disco duro con una estructura jerárquica de carpetas, de forma que los nombres de las carpetas coincidan con los de los elementos del paquete. La ruta de la carpeta raíz de esta estructura jerárquica tiene que estar especificada por la variable *CLASSPATH*.

Para el ejemplo propuesto en el punto 1, la variable *CLASSPATH* debe indicar, entre otras, la ruta de la carpeta *misClases*:

```
CLASSPATH=.;c:\java\jdk1.3\misClases;c:\java\jdk1.3
```

Siguiendo con el ejemplo, creamos la carpeta *misClases* en la ruta especificada y, dentro de ella, las carpetas *es* y *utilidades*.



3. *Especificar el paquete al que pertenece la clase.* Cuando defina una clase puede especificar a qué paquete pertenece utilizando la sentencia:

```
package nombre_paquete;
```

Esta sentencia debe ser la primera línea de código del fichero fuente.

Para finalizar el ejemplo, coloque la clase *Leer* que implementamos en el capítulo 5, en la carpeta *es* (entrada salida). Edítela y añada la siguiente línea:

```
package misClases.es;
import java.io.*;
public class Leer
{
    // Cuerpo de la clase
}
```

A continuación coloque las clases *CPersona* y *CListaTfnos* implementadas anteriormente, en la carpeta *utilidades*. Edítelas y añada a cada una de ellas la sentencia *package* para especificar el paquete al que pertenecen:

```
package misClases.utilidades;
public class CPersona
{
    // Cuerpo de la clase
}
```

```
package misClases.utilidades;
public class CListaTfnos
{
    // Cuerpo de la clase
}
```

Para probar los paquetes que acabamos de crear, copie en un nuevo directorio la aplicación *Test* que realizamos anteriormente (sólo el fichero *Test.java*). Despues, editela y añada las sentencias **import** necesarias para especificar el paquete al que pertenecen las clases *Leer*, *CPersona* y *CListaTfnos* utilizadas por la aplicación. Finalmente, compile y ejecute la aplicación para comprobar los resultados. Puede observar que al compilar la aplicación *Test* tambien serán compiladas las clases *Leer*, *CPersona* y *CListaTfnos*, si aun no lo estaban.

```
import misClases.es.*;
import misClases.utilidades.*;
import java.io.*;
// Aplicación para trabajar con matrices de objetos
//
public class Test
{
    // Cuerpo de la clase
}
```

UN EJEMPLO DE DISEÑO DE UNA CLASE

El siguiente ejemplo muestra cómo construir una clase para operar con números racionales. Un número racional es un número representado por el cociente de dos números enteros (lo que normalmente llamamos quebrado), como 5/7. El número de la izquierda se denomina numerador y el de la derecha denominador.

Una clase que envuelva un número racional es útil porque muchos de estos números no pueden ser representados exactamente utilizando el tipo **float**. Por ejemplo, $1/3 + 1/3 + 1/3$, que es 1, utilizando el tipo **float** sería 0,333333 + 0,333333 + 0,333333, que es 0,999999. Para evitar este tipo de errores debemos considerar a un número racional como un objeto con entidad propia. Esto se consigue diseñando una clase, denominada por ejemplo *CRacional*, con los atributos *numerador* y *denominador*, y con una interfaz que permita realizar cualquier operación en la que pueda intervenir un número racional.

```
public class CRacional
{
    // Atributos
    private long numerador;
    private long denominador;
```

```
// Métodos
};
```

Pensemos ahora en el conjunto de operaciones que deseamos realizar con los números racionales (a modo de ejemplo, sólo expondremos algunas de las varias posibles):

- Construir un número racional. El constructor implícito (sin argumentos) no es adecuado puesto que $0/0$ es una indeterminación. Por ello definiremos explícitamente varios constructores.
- Operaciones aritméticas. Suma, resta, multiplicación y división.
- Comparación de dos números racionales. Igual, menor y mayor.
- Operaciones para facilitar la entrada y salida.
- Copiar un racional en otro y verificar si un racional es cero.
- Incremento, decremento y cambio de signo.

Empecemos con el constructor. La construcción de un número racional cuando se omiten los argumentos parece lógico que resulte ser el racional $0/1$. Partiendo de este supuesto, el constructor *CRacional* puede ser:

```
public CRacional() // constructor
{
    numerador = 0;
    denominador = 1;
}
```

Cuando utilicemos argumentos para construir un número racional, otras operaciones que debe realizar el constructor son verificar si el denominador es cero, en cuyo caso podemos forzar a que sea 1, o negativo, en cuyo caso invertimos el signo del numerador y del denominador. Asimismo, simplificará la fracción siempre que sea posible. Según esto la definición del constructor *CRacional* con dos argumentos puede ser así:

```
public CRacional( long num, long den ) // constructor
{
    numerador = num;
    denominador = den;

    if ( denominador == 0 )
    {
        System.out.println("Error: denominador 0. Se asigna 1.");
        denominador = 1;
    }
}
```

```

    if ( denominador < 0 )
    {
        numerador = -numerador;
        denominador = -denominador;
    }
    Simplificar();
}

```

La función miembro *Simplificar* utiliza el algoritmo de Euclides para obtener el máximo común divisor (*mcd*) del numerador y del denominador, y simplificar el número racional dividiendo el numerador y el denominador por ese *mcd*.

```

protected CRacional Simplificar()
{
    // Máximo común divisor
    long mcd, temp, resto;
    mcd = Math.abs( numerador );
    temp = Math.abs( denominador );
    while ( temp > 0 )
    {
        resto = mcd % temp;
        mcd = temp;
        temp = resto;
    }
    // Simplificar
    if ( mcd > 1 )
    {
        numerador /= mcd;
        denominador /= mcd;
    }
    return this;
}

```

Otros constructores de interés pueden ser: uno que nos convierta un entero en un número racional y otro, el constructor copia:

```

public CRacional( long num ) // constructor
{
    numerador = num;
    denominador = 1;
}

public CRacional( CRacional r ) // constructor copia
{
    numerador = r.numerador;
    denominador = r.denominador;
}

```

Pensemos ahora en las operaciones aritméticas; por ejemplo, en la operación de *sumar*. Supongamos las siguientes declaraciones:

```
CRacional r1 = new CRacional(1);
CRacional r2 = new CRacional(1, 4);
CRacional r3;
```

Como el operador `+` no está definido para los números racionales y tampoco podemos definirlo, la solución puede ser utilizar una sintaxis como la siguiente:

```
r3 = r1.sumar(r2);
```

Como vemos, la solución es escribir un método *sumar* con un parámetro que haga referencia a un objeto *CRacional*, el operando de la derecha; el operando de la izquierda referencia el objeto que recibe el mensaje *sumar*. La función debe devolver una referencia al objeto *CRacional* resultado de la suma. Según lo expuesto, el método puede ser el siguiente:

```
public CRacional sumar( CRacional r )
{
    CRacional temp;
    temp = new CRacional(numerador * r.denominador +
                          denominador * r.numerador,
                          denominador * r.denominador );
    return temp;
}
```

Esta versión crea un objeto *temp* invocando al constructor *CRacional* con los valores resultantes de realizar la suma, y devuelve *temp* como resultado ya simplificado por el constructor. Este método podría escribirse también así:

```
public CRacional sumar( CRacional r )
{
    return new CRacional(numerador * r.denominador +
                          denominador * r.numerador,
                          denominador * r.denominador );
}
```

Esta versión crea un objeto temporal invocando al constructor *CRacional* con los valores resultantes de realizar la suma, y lo devuelve como resultado una vez simplificado por el constructor.

Supongamos ahora que uno de los operandos que intervienen en la suma es un entero. En este caso podríamos proceder así:

```
long n = 2;
CRacional r2 = new CRacional(1, 4);
```

```
CRacional r3;
r3 = new CRacional(n).sumar(r2);
```

Cuando ejecute este código, observará que todo funciona correctamente. Esto se debe a que la expresión *new CRacional(n)*, utilizando el constructor de la clase, construye un objeto temporal que es el que recibe el mensaje *sumar*. El resto del proceso ocurre como se ha explicado anteriormente.

La implementación de los métodos *restar*, *multiplicar* y *dividir*, sabiendo cómo se obtiene el numerador y el denominador del resultado, siguen un desarrollo análogo al explicado para *sumar*.

Pensemos ahora en las operaciones de comparación; por ejemplo en la operación que nos permita saber si dos racionales son iguales. Supongamos las siguientes declaraciones:

```
CRacional r1 = new CRacional(1);
CRacional r2 = new CRacional(1, 4);
CRacional r3;
r3 = r1.sumar(r2);
CRacional r4 = new CRacional(r2);
// ...
```

Para comprobar si dos objetos *r2* y *r3* son iguales, podemos proceder como se puede observar a continuación:

```
if (r3.equals(r2)) r1 = r3.sumar(r4);
```

La expresión *r3.equals(r2)* sugiere redefinir el método **equals** con un parámetro que haga referencia a un objeto *CRacional*; el otro objeto implicado en la comparación, el de la izquierda, es aquel que recibe el mensaje **equals**. El método debe devolver un valor **true** o **false**. Según lo expuesto, la definición del método puede ser así:

```
public boolean equals( CRacional r )
{
    return ( numerador * r.denominador ==
              denominador * r.numerador );
}
```

El resto de las operaciones de relación se desarrollan de forma similar a la expuesta.

A continuación pasamos a resolver la entrada salida de números racionales. Los métodos que implementemos como parte de la interfaz que los usuarios de esta clase utilizarán, deben tratar el número racional como un objeto indivisible;

esto es, el usuario no debe tener la posibilidad de acceder a los atributos *numerador* y *denominador* de forma independiente.

Java tiene varias sobrecargas de los métodos **print** y **println** para permitir la salida de valores de tipos predefinidos y objetos de las clases **Object**, **String** y matriz de caracteres. Si nuestra intención es utilizar la misma sintaxis para visualizar un objeto *CRacional*, nos encontraremos con que, lógicamente, no existe una sobrecarga de estos métodos para esta clase de objetos. Pensando en lo que realmente hacen estos métodos, convertir su argumento en un objeto **String**, podemos redefinir el método **toString** heredado de la clase **Object**, para convertir un objeto *CRacional* en un objeto **String**. De esta forma, mostrar un objeto *CRacional* resultará tan sencillo como se muestra a continuación:

```
CRacional r1 = new CRacional(1, 4);
System.out.println(r1.toString());
```

La expresión *r1.toString()* debe devolver un objeto **String** con el contenido correspondiente al objeto *CRacional* que recibe el mensaje *toString* expresado de la forma “*numerador/denominador*”:

```
public String toString()
{
    return new String(numerador + "/" + denominador);
}
```

Igual que para los métodos **print** y **println**, Java tiene varias sobrecargas del método **read** pero no existe una sobrecarga que permita leer objetos de la clase *CRacional* a través del teclado. Por esta razón, vamos a añadir a esta clase un método *leer* que permita teclear un número racional según el formato siguiente: [-]entero[/entero]. Este método, utilizando el método *dato* de la clase *Leer* que expusimos en el capítulo 5, leerá en número racional como una cadena de caracteres. Una vez leído, el método *leer* verificará si el formato es válido; para ello debe cumplirse que el primer carácter sea el signo menos o un dígito del 0 al 9, que los siguientes caracteres sean dígitos y que si hay una “/”, sólo sea una y no esté en la última posición. La lectura se repetirá mientras la cadena no sea válida; en otro caso, se extraerá el numerador y el denominador que utilizaremos como argumentos en la construcción del objeto *CRacional* que será devuelto por el método. A continuación puede ver el código completo para este método:

```
public static CRacional leer()
{
    long num, den;
    int i, barras;
    boolean carácterVálido;
    String racional;
```

```

do {
    barras = 0;
    System.out.print("[-]entero[/entero]: ");
    racional = Leer.dato(); // leer el racional
    if (racional.length() == 0)
        carácterVálido = false;
    else
    {
        // El primer carácter puede ser un dígito o el signo menos
        carácterVálido =
            (racional.charAt(0) >= '0' && racional.charAt(0) <= '9') ||
            (racional.charAt(0) == '-' && racional.length() > 1);
        // El último carácter no puede ser una /
        if (racional.charAt(racional.length()-1) == '/')
            carácterVálido = false;
    }
    // El resto de los caracteres pueden ser dígitos o / (sólo una)
    for (i = 1; carácterVálido && i < racional.length(); i++)
    {
        carácterVálido = racional.charAt(i) >= '0' &&
            racional.charAt(i) <= '9' ||
            racional.charAt(i) == '/';
        if (racional.charAt(i) == '/') barras++;
        if (barras > 1) carácterVálido = false;
    }
    if (!carácterVálido) System.out.println("Entrada no válida.");
}
while (!carácterVálido);
// Extraer el numerador y el denominador
if ((i = racional.indexOf('/')) == -1) // no hay denominador
{
    num = Long.parseLong(racional);
    den = 1;
}
else
{
    num = Long.parseLong(racional.substring(0, i)); // 0 a i-1
    den = Long.parseLong(racional.substring(i+1));
}
// Construir y devolver el objeto CRacional
return new CRacional(num, den);
}

```

Pensemos ahora en copiar un objeto *CRacional* en otro. Supongamos las siguientes declaraciones:

```

CRacional r1 = new CRacional(1);
CRacional r2 = new CRacional(1, 4);
// ...

```

Para poder copiar un objeto *CRacional* en otro, por ejemplo *r2* en *r1*, podemos proceder así:

```
r1.copiar(r2);
```

La expresión *r1.copiar(r2)* requiere redefinir el método *copiar* con un parámetro que haga referencia al objeto *CRacional* a copiar; el objeto sobre el que se realiza la copia, el de la izquierda, es aquel que recibe el mensaje *copiar*. El método deberá devolver el objeto copiado con el fin de poder encadenar esta operación cuando se solicite. Según lo expuesto, este método puede ser así:

```
public CRacional copiar( CRacional r )
{
    numerador = r.numerador;
    denominador = r.denominador;
    return this;
}
```

En ocasiones, puede ser necesario saber si un número racional es cero. Por ejemplo, para saber si el racional *r* es cero podríamos escribir:

```
if (r.esCero()) ...
```

En este ejemplo se observa que el objeto *r* recibe el mensaje *esCero*. La respuesta a este mensaje será la ejecución del método *esCero* que deberá devolver **true** si el racional es cero, o **false** en caso contrario. Dicho método puede escribirse así:

```
// Verificar si es 0
public boolean esCero()
{
    return numerador == 0;
}
```

Otras operaciones de interés pueden ser incrementar y decrementar en una unidad un número racional. Por ejemplo:

```
r2.copiar(r1.incrementar()); // incrementar r1 y copiarlo en r2
r3.decrementar(); // incrementar r3
```

Los métodos correspondientes que permiten realizar las operaciones mencionadas son los siguientes:

```
// Incrementar en 1
public CRacional incrementar()
{
```

```

        numerador += denominador;
        return this;
    }

// Decrementar en 1
public CRacional decrementar()
{
    numerador -= denominador;
    return this;
}

```

Y, cómo realizar una operación de la forma $a = -b$; tenga en cuenta que b no cambia. Esta operación podríamos requerirla así:

```
r2.copiar(r1.cambiadoDeSigno());
```

El método *cambiadoDeSigno* debe devolver el valor cambiado de signo del racional que recibió este mensaje, pero sin modificar éste. La solución se muestra a continuación:

```

// - unario
public CRacional cambiadoDeSigno()
{
    CRacional temp = new CRacional( -numerador, denominador );
    return temp;
}

```

Los métodos expuestos no son los únicos; simplemente son un ejemplo de las muchas operaciones que se pueden programar. A continuación se muestra el código completo que hemos escrito para la clase *CRacional*:

```

///////////////////////////////
// Clase para operar con números racionales (utiliza la clase Leer)
//
public class CRacional
{
    // Atributos
    private long numerador;
    private long denominador;

    // Métodos
    protected CRacional Simplificar()
    {
        // MÁximo común divisor
        long mcd, temp, resto;
        mcd = Math.abs( numerador );
        temp = Math.abs( denominador );

```

```
while ( temp > 0 )
{
    resto = mcd % temp;
    mcd = temp;
    temp = resto;
}
// Simplificar
if ( mcd > 1 )
{
    numerador /= mcd;
    denominador /= mcd;
}
return this;
}

public CRacional() // constructor
{
    numerador = 0;
    denominador = 1;
}

public CRacional( long num ) // constructor
{
    numerador = num;
    denominador = 1;
}

public CRacional( long num, long den ) // constructor
{
    numerador = num;
    denominador = den;
    if ( denominador == 0 )
    {
        System.out.println("Error: denominador 0. Se asigna 1.");
        denominador = 1;
    }
    if ( denominador < 0 )
    {
        numerador = -numerador;
        denominador = -denominador;
    }
    Simplificar();
}

public CRacional( CRacional r ) // constructor copia
{
    numerador = r.numerador;
    denominador = r.denominador;
}
```

```
// Sumar números racionales
public CRacional sumar( CRacional r )
{
    return new CRacional(numerador * r.denominador +
                          denominador * r.numerador,
                          denominador * r.denominador );
}

// Restar números racionales
public CRacional restar( CRacional r )
{
    return new CRacional(numerador * r.denominador -
                          denominador * r.numerador,
                          denominador * r.denominador );
}

// Multiplicar números racionales
public CRacional multiplicar( CRacional r )
{
    return new CRacional(numerador * r.numerador,
                          denominador * r.denominador );
}

// Dividir números racionales
public CRacional dividir( CRacional r )
{
    return new CRacional(numerador * r.denominador,
                          denominador * r.numerador );
}

// Verificar si dos números racionales son iguales
public boolean equals( CRacional r )
{
    return ( numerador * r.denominador ==
             denominador * r.numerador );
}

// Verificar si un racional es menor que otro
public boolean menor( CRacional r )
{
    return ( numerador * r.denominador <
             denominador * r.numerador );
}

// Verificar si un racional es mayor que otro
public boolean mayor( CRacional r )
{
    return ( numerador * r.denominador >
             denominador * r.numerador );
}
```

```
// Devolver un número racional como cadena
public String toString()
{
    return new String(numerador + "/" + denominador);
}

// Establecer un número racional
public static CRacional leer()
{
    long num, den;
    int i, barras;
    boolean carácterVálido;
    String racional;

    do
    {
        barras = 0;
        System.out.print("[-]entero[/entero]: ");
        racional = Leer.dato(); // leer el racional

        if (racional.length() == 0)
            carácterVálido = false;
        else
        {
            // El primer carácter puede ser un dígito o el signo menos
            carácterVálido =
                (racional.charAt(0) >= '0' && racional.charAt(0) <= '9') ||
                (racional.charAt(0) == '-' && racional.length() > 1);
            // El último carácter no puede ser una /
            if (racional.charAt(racional.length()-1) == '/')
                carácterVálido = false;
        }
        // El resto de los caracteres pueden ser dígitos o / (sólo una)
        for (i = 1; carácterVálido && i < racional.length(); i++)
        {
            carácterVálido = racional.charAt(i) >= '0' &&
                            racional.charAt(i) <= '9' ||
                            racional.charAt(i) == '/';
            if (racional.charAt(i) == '/') barras++;
            if (barras > 1) carácterVálido = false;
        }
        if (!carácterVálido) System.out.println("Entrada no válida.");
    }
    while (!carácterVálido);
    // Extraer el numerador y el denominador
    if ((i = racional.indexOf('/')) == -1) // no hay denominador
    {
        num = Long.parseLong(racional);
        den = 1;
    }
}
```

```
        else
    {
        num = Long.parseLong(racional.substring(0, i)); // 0 a i-1
        den = Long.parseLong(racional.substring(i+1));
    }
    // Construir y devolver el objeto CRacional
    return new CRacional(num, den);
}

// Copiar un racional en otro
public CRacional copiar( CRacional r )
{
    numerador = r.numerador;
    denominador = r.denominador;
    return this;
}

// Verificar si es 0
public boolean esCero()
{
    return numerador == 0;
}

// Incrementar en 1
public CRacional incrementar()
{
    numerador += denominador;
    return this;
}

// Decrementar en 1
public CRacional decrementar()
{
    numerador -= denominador;
    return this;
}

// - unario
public CRacional cambiadoDeSigno()
{
    CRacional temp = new CRacional( -numerador, denominador );
    return temp;
}
}
```

EJERCICIOS RESUELTOS

Una matriz multidimensional en Java representa un conjunto de elementos que pueden ser accedidos mediante variables suscritas o de subíndices. Dichos subíndices son especificados utilizando uno o más corchetes: `[]`. Por ejemplo:

```
double[][][] miMatrizDouble = new double[5][10][4];
int i, j, k, conta = 1;
// ...
miMatrizDouble[i][j][k] = conta++;
```

Una construcción similar puede realizarse utilizando una matriz unidimensional y manipularla como si fuera una matriz multidimensional. Para ello, definiremos una clase `CMatriz` con los siguientes atributos:

```
public class CMatriz
{
    private double[] matriz; // matriz unidimensional
    private int nDims; // número de dimensiones
    private int[] dimsMatriz; // valor de cada dimensión
    // ...
};
```

La clase `CMatriz` tiene como función representar una matriz multidimensional. Observe que el miembro `matriz` sirve para referenciar una matriz de una dimensión de elementos de tipo `double`, que el miembro `nDims` contiene el número de dimensiones y `dimsMatriz` es una referencia a una matriz que contendrá el valor de cada una de ellas.

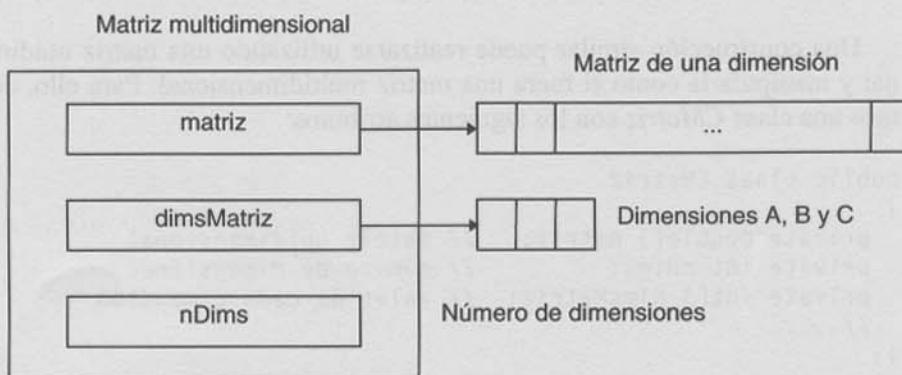
Un ejemplo de manipulación de un objeto `CMatriz` es el siguiente:

```
final int A = 5;
final int B = 10;
int i, j, conta = 1;
CMatriz m = new CMatriz( A, B ); // matriz de 2 dimensiones (A*B)

// Asignar datos a la matriz m
for ( i = 0; i < A; i++ )
    for ( j = 0; j < B; j++ )
        m.asignarDato(conta++, i, j );
// Visualizar la matriz m
for ( i = 0; i < A; i++ )
{
    for ( j = 0; j < B; j++ )
        System.out.print(m.obtenerDato( i, j ) + " ");
    System.out.println();
}
```

En este ejemplo *m* representa una matriz de dos dimensiones. Observe que para acceder a un elemento utilizamos dos subíndices *i* y *j*. Pero como la matriz físicamente es una matriz de una dimensión, la idea fundamental es implementar un mecanismo que convierta una posición dada por 1, 2 ó 3 subíndices en la posición equivalente de la matriz unidimensional. Por ejemplo, si los subíndices del elemento al que deseamos acceder son *i1*, *i2* e *i3* y las dimensiones de la matriz *m* son *d1*, *d2* y *d3*, el desplazamiento se calcula así: $((i1*d2)+i2)*d3+i3$.

La representación gráfica de la estructura de datos construida es la siguiente:



Según lo expuesto, la clase *CMatriz* estará formada por los atributos privados mencionados, por el método privado,

```
void construir(int[] dim )
```

y por los métodos públicos,

```

CMatriz()
CMatriz( int d1 )
CMatriz( int d1, int d2 )
CMatriz( int d1, int d2, int d3 )
int totalElementos()
int desplazamiento( int[] subind )
void asignarDato( int dato, int i1 )
void asignarDato( int dato, int i1, int i2 )
void asignarDato( int dato, int i1, int i2, int i3 )
double obtenerDato( int i1 )
double obtenerDato( int i1, int i2 )
double obtenerDato( int i1, int i2, int i3 )
  
```

Suponiendo que queremos manipular matrices de 1, 2 ó 3 dimensiones, responda a las siguientes preguntas:

1. Escriba el esqueleto de la definición de la clase *CMatriz*.
2. Escriba los constructores *CMatriz*. Sus parámetros se corresponden con los valores de las dimensiones de la matriz. Estos métodos invocan al método *construir* para crear un objeto *CMatriz*.
3. Escriba el método *construir*. Este método es invocado por los constructores de la clase y comprueba si todas las dimensiones son positivas. Después establece los atributos de *CMatriz*. Tenga presente que *matriz* referencia a una matriz unidimensional que representa a la matriz de 1, 2 ó 3 dimensiones.

```
void construir(int[] dim)
```

dim matriz unidimensional de enteros que contiene el valor de cada una de las dimensiones.

Por ejemplo, si *n* es 2, *dim[0]* y *dim[1]* tienen que ser valores mayores que cero y *dim[2]* no interviene. Entonces el número de elementos de la matriz sería *dim[0] * dim[1]*. Este valor será calculado por el método *totalElementos* que se expone en el apartado siguiente.

5. Escriba el método *totalElementos*. Este método calcula el número total de elementos de la matriz de 1, 2 ó 3 dimensiones.

```
int totalElementos()
```

El método *totalElementos* retorna el número total de elementos de la matriz.

6. Escriba el método *desplazamiento*. Este método calcula la posición que tiene dentro de la matriz unidimensional referenciada por *matriz*, el elemento que está en la matriz multidimensional en la posición especificada por los subíndices almacenados en la matriz referenciada por *subind*. Previamente, verifica si los subíndices están dentro de los límites permitidos.

```
int desplazamiento( int[] subind )
```

El método *desplazamiento* retorna la posición en la matriz unidimensional del elemento especificado por *subind* o -1 si algún subíndice es inválido.

7. Escriba el método *asignarDato*. Este método asigna un dato *d* al elemento de la matriz multidimensional, especificado por los subíndices *i1*, *i2* e *i3*. *asignarDato* invoca al método *desplazamiento* para calcular el desplazamiento.

```
void asignarDato( int dato, int i1 )
void asignarDato( int dato, int i1, int i2 )
```

```
void asignarDato( int dato, int i1, int i2, int i3 )
```

8. Escriba el método *obtenerDato*. Este método obtiene un dato del elemento de la matriz multidimensional, especificado por sus subíndices *i1*, *i2* e *i3*. *obtenerDato* invoca al método *desplazamiento* para calcular el desplazamiento.

```
double obtenerDato( int i1 )
double obtenerDato( int i1, int i2 )
double obtenerDato( int i1, int i2, int i3 )
```

El método *obtenerDato* retorna el valor almacenado en el elemento especificado de la matriz.

9. Utilizando la clase *CMatriz* que acaba de construir, escriba un programa que utilice como cuerpo del método **main**, el expuesto en el enunciado. El resultado que tiene que obtener con este método es:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50
```

10. ¿Por qué es necesario sobrecargar los métodos *asignarDato* y *obtenerDato*?
 11. ¿Es necesario un destructor para esta clase? ¿Por qué?
 12. ¿Qué métodos se invocan y en qué orden, cuando se ejecuta la sentencia?

```
CMatriz m( A, B );
```

13. ¿Qué métodos se invocan y en qué orden, cuando se ejecuta la sentencia?

```
m.obtenerDato( i, j );
```

La solución a las preguntas 1 a 8 puede obtenerlas del código presentado a continuación. Dicho código corresponde a la definición de la clase *CMatriz*.

```
///////////////////////////////
// Matriz multidimensional basada en una unidimensional
//
public class CMatriz
{
    private double[] matriz; // matriz unidimensional
    private int nDims; // número de dimensiones
    private int[] dimsMatriz; // valor de cada dimensión
```

```
private void construir( int[] dim )
{
    int i;
    for ( i = 0; i < dim.length; i++ )
        if ( dim[i] < 1 )
    {
        System.out.println("Dimensión nula o negativa");
        System.exit(-1);
    }
    // Establecer los atributos
    dimsMatriz = new int[dim.length];
    for ( i = 0; i < dim.length; i++ ) dimsMatriz[i] = dim[i];
    nDims = dim.length;
    matriz = new double[totalElementos()];
}

public CMatriz() // constructor
{
    int dim[] = { 10 }; // dimensión por omisión
    construir( dim );
}

public CMatriz( int d1 ) // constructor
{
    int dim[] = { d1 }; // una dimensión
    construir( dim );
}

public CMatriz( int d1, int d2 ) // constructor
{
    int dim[] = { d1, d2 }; // dos dimensiones
    construir( dim );
}

public CMatriz( int d1, int d2, int d3 ) // constructor
{
    int dim[] = { d1, d2, d3 }; // tres dimensiones
    construir( dim );
}

public int totalElementos()
{
    int i;
    int nTElementos = 1;
    // Calcular el número total de elementos de la matriz
    for ( i = 0; i < nDims; i++ )
        nTElementos *= dimsMatriz[i];
    return nTElementos;
}
```

```
public int desplazamiento( int[] subind )
{
    int i;
    int desplazamiento = 0;

    for ( i = 0; i < nDims; i++ )
    {
        // Verificar si los subíndices están dentro del rango
        if ( subind[i] < 0 || subind[i] > dimsMatriz[i] )
        {
            System.out.println("Subíndice fuera de rango");
            return -1;
        }
        // Desplazamiento equivalente en la matriz unidimensional
        desplazamiento += subind[i];
        if ( i+1 < nDims )
            desplazamiento *= dimsMatriz[i+1];
    }
    return desplazamiento;
}

public void asignarDatos( int dato, int i1 )
{
    asignarDatos(dato, i1, 0, 0);
}

public void asignarDatos( int dato, int i1, int i2 )
{
    asignarDatos(dato, i1, i2, 0);
}

public void asignarDatos( int dato, int i1, int i2, int i3 )
{
    // Asignar un valor al elemento especificado de la matriz
    int subind[] = { i1, i2, i3 };
    int i = desplazamiento( subind );
    if ( i == -1 ) System.exit(-1); // subíndice fuera de rango
    matriz[i] = dato;
}

public double obtenerDatos( int i1 )
{
    return obtenerDatos( i1, 0, 0 );
}

public double obtenerDatos( int i1, int i2 )
{
    return obtenerDatos( i1, i2, 0 );
}
```

```

public double obtenerDato( int i1, int i2, int i3 )
{
    // Obtener el valor al elemento especificado de la matriz
    int subind[] = { i1, i2, i3 };
    int i = desplazamiento( subind );
    if ( i == -1 ) System.exit(-1); // subíndice fuera de rango
    return matriz[i];
}
}

```

A continuación se muestra la respuesta a la pregunta 9.

```

// Aplicación para trabajar con CMatriz
//
public class Test
{
    public static void main(String[] args)
    {
        final int A = 5;
        final int B = 10;
        int i, j, conta = 1;
        CMatriz m = new CMatriz( A, B ); // matriz de 2 dimensiones

        // Asignar datos a la matriz m
        for ( i = 0; i < A; i++ )
            for ( j = 0; j < B; j++ )
                m.asignarDato(conta++, i, j );

        // Visualizar la matriz m
        for ( i = 0; i < A; i++ )
        {
            for ( j = 0; j < B; j++ )
                System.out.print(m.obtenerDato( i, j ) + " ");
            System.out.println();
        }
    }
}

```

Respuesta a la pregunta 10. Los métodos *asignarDato* y *obtenerDato* están sobrecargados para poder utilizar sus formas adecuadas según se trate de una matriz de 1, 2 ó 3 dimensiones.

Respuesta a la pregunta 11. No es necesario escribir el método **finalize** porque de liberar la memoria asignada dinámicamente (operador **new**) se encarga el recolector de basura.

Respuesta a la pregunta 12. Los métodos invocados cuando se ejecuta la sentencia `CMatriz m(A, B)` son:

```
public CMatriz( int d1, int d2 ) // constructor
private void construir( int[] dim )
public int totalElementos()
```

Respuesta a la pregunta 13. Los métodos invocados cuando se ejecuta la sentencia `m.obtenerDatos(i, j)` son:

```
public double obtenerDatos( int i1, int i2 )
public double obtenerDatos( int i1, int i2, int i3 )
public int desplazamiento( int[] subind )
```

EJERCICIOS PROPUESTOS

- Suponiendo un texto escrito en minúsculas y sin signos de puntuación (una palabra estará separada de otra por un espacio en blanco), realizar un programa que lea texto de la entrada estándar (del teclado) y dé como resultado la frecuencia con que aparece cada palabra leída del texto.

El resultado se almacenará en un matriz en la que cada elemento será un objeto `CPalabra` con los atributos:

```
String palabra; // palabra
int contador; // número de veces que aparece en el texto
```

A su vez, la matriz, que irá creciendo a medida que se vayan añadiendo palabras, será un atributo de la clase `CFrecuenciasPalabras`. La interfaz de esta clase incluirá al menos los métodos: `BuscarPalabra`, `InsertarPalabra` y `ObtenerObjPalabra`.

- Escribir una clase `Complejo` para trabajar con números complejos.

¿Qué es un número complejo? Un número complejo está compuesto por dos números reales y se representa de la forma $a+bi$; a recibe el nombre de componente real y b el de componente imaginaria. Si $b = 0$, se obtiene el número real a , lo que quiere decir que los números reales son un caso particular de los números complejos.

Los números complejos cubren un campo que no tiene sentido en el campo de los números reales. Por ejemplo, no existe ningún número real que sea igual a $\sqrt{-9}$. Tampoco tienen sentido las expresiones $(-2)^{3/2}$ o $\log(-2)$. Para resolver este tipo

de expresiones se definió la unidad imaginaria $\sqrt{-1}$, que se representa por i . De este modo podemos escribir que:

$$2 + \sqrt{-9} = 2 + 3\sqrt{-1} = 2 + 3i, \text{ que se representa como } (2, 3).$$

Puesto que un número complejo (a, b) es un par ordenado de números reales, puede representarse geométricamente mediante un punto en el plano; dicho de otra forma, mediante un vector. De aquí se deduce que: $a+bi$, número complejo en forma binómica, es equivalente a $m(\cos \alpha + i \sin \alpha)$, número complejo en forma polar, lo que indica que $a = m \cos \alpha$ y que $b = m \sin \alpha$.

El número positivo $m = \sqrt{a^2 + b^2}$ se denomina *módulo* o valor absoluto y el ángulo $\alpha = \arctg(b/a)$ recibe el nombre de *argumento*.

Operaciones aritméticas:

Suma: $(a, b) + (c, d) = (a+c, b+d)$

Diferencia: $(a, b) - (c, d) = (a-c, b-d)$

Producto: $(a, b) * (c, d) = (ac-bd, ad+bc)$

Cociente: $(a, b) / (c, d) = ((ac+bd)/(c^2+d^2), (bc-ad)/(c^2+d^2))$

Estas operaciones y otras formarán parte de la interfaz de la clase *Complejo*. Las comparaciones entre complejos estarán referidas a sus módulos.

Según la definición dada, podemos representar un complejo como un objeto que tenga dos atributos, uno para almacenar la parte real y otra para la parte imaginaria.

```
public class Complejo
{
    private double real; // parte real
    private double imag; // parte imaginaria
    // ...
}
```

La interfaz de esta clase proporcionará varios conjuntos de métodos que se pueden clasificar de la forma siguiente:

- Uno o más constructores. El complejo construido por omisión será el $(0, 0)$.
- Paso de forma polar a binómica.
- Operaciones aritméticas sumar, restar, multiplicar y dividir.

- Comparación de complejos. La igualdad y la desigualdad la realizaremos en módulo y argumento. El resto de las comparaciones tienen sentido cuando sólo se comparan los módulos.
 - Operaciones trigonométricas.
 - Operaciones logaritmo natural, exponencial, potencia y raíz cuadrada.
 - Operaciones de entrada/salida.
 - Complejo conjugado, negativo y opuesto.
 - Operaciones de asignación.
3. Se quiere escribir un programa para manipular ecuaciones algebraicas o polinómicas dependientes de una variable. Por ejemplo:

$$2x^3 - x + 8.25 \quad \text{más} \quad 5x^5 - 2x^3 + 7x^2 - 3 \quad \text{igual a} \quad 5x^5 + 7x^2 - x + 5.25$$

Cada término del polinomio será representado por una clase *CTermino* y cada polinomio por una clase *CPolinomio*.

La clase *CTermino* tendrá dos atributos privados: *coeficiente* y *exponente*, y los métodos necesarios para permitir al menos:

- Construir un término, iniciado a cero por omisión.
- Acceder al coeficiente de un término para obtener su valor.
- Acceder al exponente de un término para obtener su valor.
- Obtener la cadena de caracteres equivalente a un término con el formato siguiente: {+|-} $7x^4$.

La clase *CPolinomio* tendrá dos datos miembro privados: número de términos que tiene el polinomio (*nroTerminos*) y una matriz que referenciará los términos del polinomio (*termino*), así como los métodos necesarios para permitir al menos:

- Construir un polinomio, inicialmente con 0 términos.
- Obtener el número de términos que tiene actualmente el polinomio.
- Asignar un término a un polinomio colocándolo en orden ascendente del exponente. Si el coeficiente es nulo, no se realizará ninguna operación. Cada vez que se inserte un nuevo término, se incrementará automáticamente el tamaño del polinomio en uno. El método encargado de esta operación tendrá un parámetro de la clase *CTermino*.
- Sumar dos polinomios. El polinomio resultante quedará también ordenado en orden ascendente del exponente.
- Obtener la cadena de caracteres correspondiente a la representación de un polinomio con el formato siguiente: $+ 5x^5 - 1x^1 + 5.25$.

Algunas de las principales ventajas de la POO son la reutilización del código existente y la creación de sistemas más flexibles y adaptables. La herencia es el mecanismo que permite la reutilización del código ya que el sistema de clases es jerárquico y permite la definición de subclases que heredan las características de la clase superior. La herencia es un concepto fundamental en la POO y se aplica tanto a los tipos de datos como a las clases.

CAPÍTULO 10

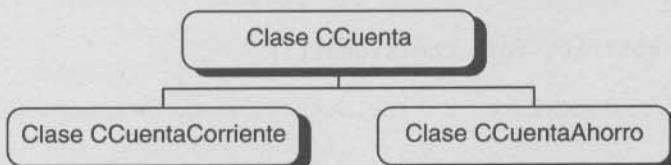
© F.J.Ceballos/RA-MA

SUBCLASES E INTERFACES

Las características fundamentales de la POO son *abstracción*, *encapsulamiento*, *herencia* y *polimorfismo*. Hasta ahora sólo hemos abordado la *abstracción* y la *encapsulación*.

Entre las características enumeradas anteriormente, hay una que destaca: la *herencia*. La *herencia* provee el mecanismo más simple para especificar una forma alternativa de acceso a una clase existente, o bien para definir una nueva clase que añada nuevas características a una clase existente. Esta nueva clase se denomina *subclase* o *clase derivada* y la clase existente, *superclase* o *clase base*.

Con la herencia todas las clases están clasificadas en una jerarquía estricta. Cada clase tiene su superclase (la clase superior en la jerarquía), y cada clase puede tener una o más subclases (las clases inferiores en la jerarquía). Las clases que están en la parte inferior en la jerarquía se dice que *heredan* de las clases que están en la parte superior en la jerarquía. Por ejemplo, la figura siguiente indica que las clases *CCuentaCorriente* y *CCuentaAhorro* heredan de la clase *CCuenta*.



Una jerarquía de clases muestra cómo los objetos se derivan de otros objetos más simples heredando su comportamiento. Los usuarios de C++ y de otros lenguajes de programación orientada a objetos están acostumbrados a ver jerarquías de clases para describir la *herencia*. Los de Java seguirán, en general, los mismos pasos.

CLASES Y MÉTODOS ABSTRACTOS

En una jerarquía de clases, una clase es tanto más especializada cuanto más alejada esté de la raíz, entendiendo por clase raíz aquella de la cual heredan directa o indirectamente el resto de las clases de la jerarquía; y al contrario, es tanto más genérica cuanto más cerca esté de la raíz. Sirva de ejemplo la clase **Object**, clase raíz de la jerarquía de clases de Java; es una clase que sólo define los atributos y el comportamiento comunes a todas las clases. Lo mismo sucederá con la clase *CCuenta* de la figura anterior, será más bien una clase genérica, correspondiendo los atributos y comportamiento más específicos a las clases *CCuentaAhorro* y *CCuentaCorriente*.

Cuando una clase se diseña para ser genérica, es casi seguro que no necesitaremos crear objetos de ella; la razón de su existencia es proporcionar los atributos y comportamientos que serán compartidos por todas sus subclases. Una clase que se comporte de la forma descrita se denomina clase *abstracta* y se define como tal calificándola explícitamente *abstracta* (**abstract**). Por ejemplo:

```
public abstract class CCuenta
{
    // Cuerpo de la clase
}
```

Una clase abstracta puede contener el mismo tipo de miembros que una clase que no lo sea, y además pueden contener métodos abstractos, que una clase no abstracta no puede contener.

¿Qué es un método abstracto? Es un método calificado **abstract** con la particularidad de que no tiene cuerpo. Por ejemplo, el siguiente código declara *comisiones* como un método abstracto:

```
public abstract class CCuenta
{
    // ...
    public abstract void comisiones();
    // ...
}
```

¿Por qué no tiene cuerpo? Porque la idea es proporcionar métodos que deban ser redefinidos en las subclases de la clase abstracta, con la intención de adaptarlos a las necesidades particulares de éstas.

A la vista de este ejemplo, puede intentar declarar la clase *CCuenta* no abstracta y comprobará que el compilador Java le muestra un mensaje indicándole que una clase sólo puede contener métodos abstractos si es abstracta.

SUBCLASES Y HERENCIA

Vuelva a echar una ojeada a la figura mostrada al principio de este capítulo. Se trata de una jerarquía de clases que puede ser analizada desde dos puntos de vista:

1. Cuando en un principio se abordó el diseño de una aplicación para administrar las cuentas de una entidad bancaria, fue suficiente con las capacidades proporcionadas por la clase *CCuenta*. Posteriormente, la evolución de los mercados bancarios, sugirió nuevas modalidades de cuentas. La mejor solución para adaptar la aplicación a esas nuevas exigencias fue definir una subclase para cada nueva modalidad, puesto que el mecanismo de herencia ponía a disposición de las subclases todo el código de su superclase, al que sólo era necesario añadir las nuevas especificaciones. Evidentemente, la *herencia* es una forma sencilla de *reutilizar el código* proporcionado por otras clases.
2. Cuando se abordó el diseño de una aplicación para administrar las cuentas de una entidad bancaria, la solución fue diseñar una clase especializada para cada una de las cuentas y agrupar el código común en una superclase de éstas.

En los dos casos planteados, la herencia es la solución para reutilizar código perteneciente a otras clases. Para ilustrar el mecanismo de herencia vamos a implementar la jerarquía de clases de la figura anterior. La idea es diseñar una aplicación para administrar las cuentas corrientes y de ahorro de los clientes de una entidad bancaria. Como ambas cuentas tienen bastantes cosas en común, hemos decidido agrupar éstas en una clase *CCuenta* de la cual posteriormente derivaremos las cuentas específicas que vayan surgiendo. Según este planteamiento, no parece que tengamos intención de crear objetos de *CCuenta*; más bien la intención es que agrupe el código común que heredarán sus subclases, razón por la cual la declararemos abstracta.

Pensemos entonces inicialmente en el diseño de la clase *CCuenta*. Después de un análisis acerca de los factores que intervienen en una cuenta en general, llegamos a la conclusión de que los atributos y métodos comunes a cualquier tipo de cuenta son los siguientes:

Atributo	Significado
<i>nombre</i>	Dato de tipo String que almacena el nombre del propietario de la cuenta.
<i>cuenta</i>	Dato de tipo String que almacena el número de la cuenta.
<i>saldo</i>	Dato de tipo double que almacena el saldo de la cuenta.
<i>tipoDeInterés</i>	Dato de la clase de tipo double que almacena el tipo de interés.

Método	Significado
<i>CCuenta</i>	Es el constructor de la clase. Inicia los datos <i>nombre</i> , <i>cuenta</i> , <i>saldo</i> y <i>tipoDeInterés</i> .
<i>asignarNombre</i>	Permite asignar el dato <i>nombre</i> .
<i>obtenerNombre</i>	Retorna el dato <i>nombre</i> .
<i>asignarCuenta</i>	Permite asignar el dato <i>cuenta</i> .
<i>obtenerCuenta</i>	Retorna el dato <i>cuenta</i> .
<i>estado</i>	Retorna el saldo de la cuenta.
<i>comisiones</i>	Es un método abstracto sin parámetros que será redefinido en las subclases. Se ejecutará los días uno de cada mes para cobrar el importe del mantenimiento de una cuenta.
<i>ingreso</i>	Es un método que tiene un parámetro <i>cantidad</i> de tipo double que añade la cantidad especificada al saldo actual de la cuenta.
<i>reintegro</i>	Es un método que tiene un parámetro cantidad de tipo double que resta la cantidad especificada del saldo actual de la cuenta.
<i>asignarTipoDeInterés</i>	Método que permite asignar el dato <i>tipoDeInterés</i> .
<i>obtenerTipoDeInterés</i>	Método que retorna el dato <i>tipoDeInterés</i> .
<i>intereses</i>	Método abstracto. Calcula los intereses producidos.

El código correspondiente a esta clase se expone a continuación:

```
//////////  
// Clase CCuenta: clase abstracta que agrupa los datos comunes a  
// cualquier tipo de cuenta bancaria.  
//  
public abstract class CCuenta  
{  
    // Atributos  
    private String nombre;  
    private String cuenta;  
    private double saldo;  
    private double tipoDeInterés;  
  
    // Métodos  
    public CCuenta() {};  
    public CCuenta(String nom, String cue, double sal, double tipo)  
    {  
        asignarNombre(nom);  
        asignarCuenta(cue);  
        ingreso(sal);  
        asignarTipoDeInterés(tipo);  
    }  
}
```

```
public void asignarNombre(String nom)
{
    if (nom.length() == 0)
    {
        System.out.println("Error: cadena vacía");
        return;
    }
    nombre = nom;
}

public String obtenerNombre()
{
    return nombre;
}

public void asignarCuenta(String cue)
{
    if (cue.length() == 0)
    {
        System.out.println("Error: cuenta no válida");
        return;
    }
    cuenta = cue;
}

public String obtenerCuenta()
{
    return cuenta;
}

public double estado()
{
    return saldo;
}

public abstract void comisiones();

public abstract double intereses();

public void ingreso(double cantidad)
{
    if (cantidad < 0)
    {
        System.out.println("Error: cantidad negativa");
        return;
    }
    saldo += cantidad;
}
```

```

public void reintegro(double cantidad)
{
    if (saldo - cantidad < 0)
    {
        System.out.println("Error: no dispone de saldo");
        return;
    }
    saldo -= cantidad;
}

public double obtenerTipoDeInterés()
{
    return tipoDeInterés;
}

public void asignarTipoDeInterés(double tipo)
{
    if (tipo < 0)
    {
        System.out.println("Error: tipo no válido");
        return;
    }
    tipoDeInterés = tipo;
}

```

Si ahora, utilizando la definición de la clase anterior intenta ejecutar una línea de código como la siguiente, obtendrá un error indicándole que la clase es abstracta.

```
CCuenta cliente01 = new CCuenta();
```

DEFINIR UNA SUBCLASE

Pensemos ahora en un tipo de cuenta específico, como es una cuenta de ahorro. Una cuenta de ahorro tiene las características aportadas por un objeto *CCuenta*, y además algunas otras; por ejemplo, un atributo que especifique el importe que hay que pagar mensualmente por el mantenimiento de la misma. Esto significa que necesitamos diseñar una nueva clase, *CCuentaAhorro*, que tenga las mismas capacidades de *CCuenta*, pero a las que hay que añadir otras que den solución a las nuevas necesidades.

Una forma de hacer esto sería definir una nueva clase *CCuentaAhorro* con los atributos y métodos de *CCuenta*, a los que añadiríamos los nuevos atributos y métodos, según muestra el esquema siguiente:

```
public class CCuentaAhorro
{
    // Atributos y métodos de CCuenta
    // Nuevos atributos y métodos de CCuentaAhorro
}
```

Esta forma de proceder puede que funcione, pero no deja de ser una mala solución; además de suponer un derroche de tiempo y esfuerzo, todo el trabajo que ya estaba realizado no ha servido para nada. Aquí es donde la herencia juega un papel importante; la utilización de esta característica evitará que recurramos a soluciones como la planteada. A través de la herencia, Java permite definir la clase *CCuentaAhorro* como una extensión de *CCuenta*. Y esto ¿cómo se hace? Definiendo una subclase de la clase existente.

Una *subclase* es un nuevo tipo de objetos definido por el usuario que tiene la propiedad de heredar los atributos y métodos de otra clase definida previamente, denominada *superclase*. La sintaxis para definir una *subclase* es la siguiente:

```
class nombre_subclase extends nombre_superclase
{
    // Cuerpo de la subclase
}
```

La palabra clave **extends** significa que se está definiendo una clase denominada *nombre_subclase* que es una extensión de otra denominada *nombre_superclase*; también se puede decir que *nombre_subclase* es una clase derivada de *nombre_superclase*.

El ejemplo mostrado a continuación define la clase *CCuentaAhorro* como una extensión de *CCuenta*.

```
public class CCuentaAhorro extends CCuenta
{
    // CCuentaAhorro ha heredado los miembros de CCuenta
    // Escriba aquí los nuevos atributos y métodos de CCuentaAhorro
}
```

Si no se especifica la cláusula **extends** con el nombre de la superclase, se entiende que la superclase es la clase **Object**. Por lo tanto, la clase *CCuenta* está derivada de la clase **Object**.

Una subclase puede serlo de una sola superclase, lo que se denomina *herencia simple* o *derivación simple*. Java, a diferencia de otros lenguajes orientados a objetos, no permite la herencia múltiple o derivación múltiple, esto es, que una subclase se derive de dos o más clases.

Una subclase puede, a su vez, ser una superclase de otra clase, dando lugar así a una *jerarquía de clases*. Por lo tanto, una clase puede ser una superclase directa de una subclase, si figura explícitamente en la definición de la subclase, o una superclase indirecta si está varios niveles arriba en la jerarquía de clases, y por lo tanto no figura explícitamente en el encabezado de la subclase.

Control de acceso a los miembros de las clases

En el capítulo dedicado a clases se expuso que para controlar el acceso a los miembros de una clase, Java provee las palabras clave **private** (privado), **protected** (protegido) y **public** (público), o bien pueden omitirse (acceso predeterminado). Lo allí estudiado se amplía ahora para las subclases. Para evitar confusiones, la tabla siguiente resume de una forma clara qué clases, o subclases, pueden acceder a los miembros de otra clase, dependiendo del control de acceso especificado:

Puede ser accedido desde:	Un miembro declarado en una clase como			
	<i>privado</i>	<i>predeterminado</i>	<i>protectoro</i>	<i>público</i>
Su misma clase	sí	sí	sí	sí
Cualquier clase o subclase del mismo paquete	no	sí	sí	sí
Cualquier clase de otro paquete	no	no	no	sí
Cualquier subclase de otro paquete	no	no	sí	sí

Qué miembros hereda una subclase

Los siguientes puntos resumen las reglas a tener en cuenta cuando se define una subclase:

1. Una subclase hereda todos los miembros de su superclase, excepto los constructores, lo que no significa que tenga acceso directo a todos los miembros. Una consecuencia inmediata de esto es que la estructura interna de datos de un objeto de una subclase, estará formada por los atributos que ella define y por los heredados de su superclase.

Una subclase no tiene acceso directo a los miembros privados (**private**) de su superclase.

Una subclase sí puede acceder directamente a los miembros públicos (**public**) y protegidos (**protected**) de su superclase; y en el caso de que pertenezca al mismo paquete de su superclase, también puede acceder a los miembros pre-determinados.

2. Una subclase puede añadir sus propios atributos y métodos. Si el nombre de alguno de estos miembros coincide con el de un miembro heredado, este último queda oculto para la subclase, que se traduce en que la subclase ya no puede acceder directamente a ese miembro. Lógicamente, lo expuesto tiene sentido siempre que nos refiramos a los miembros de la superclase a los que la subclase podía acceder, según el control de acceso aplicado.
3. Los miembros heredados por una subclase pueden, a su vez, ser heredados por más subclases de ella. A esto se le llama propagación de herencia.

Continuando con el ejemplo, diseñemos una nueva clase *CCuentaAhorro* que tenga, además de las mismas capacidades de *CCuenta*, las siguientes:

Atributo	Significado
<i>cuotaMantenimiento</i>	Dato de tipo double que almacena la comisión que cobrará la entidad bancaria por el mantenimiento de la cuenta.
Método	Significado
<i>CCuentaAhorro</i>	Es el constructor de la clase. Inicia los atributos de la misma.
<i>asignarCuotaManten</i>	Establece la cuota de mantenimiento de la cuenta.
<i>obtenerCuotaManten</i>	Devuelve la cuota de mantenimiento de la cuenta.
<i>comisiones</i>	Método que se ejecuta los días uno de cada mes para cobrar el importe correspondiente al mantenimiento de la cuenta.
<i>intereses</i>	Método que permite calcular el importe correspondiente a los intereses/mes producidos.

La definición correspondiente a esta clase se expone a continuación:

```
import java.util.*;

// Clase CCuentaAhorro: clase derivada de CCuenta
//
public class CCuentaAhorro extends CCuenta
{
    // Atributos
    private double cuotaMantenimiento;
```

```

// Métodos
public CCuentaAhorro() {} // constructor sin parámetros

public void asignarCuotaManten(double cantidad)
{
    if (cantidad < 0)
    {
        System.out.println("Error: cantidad negativa");
        return;
    }
    cuotaMantenimiento = cantidad;
}

public double obtenerCuotaManten()
{
    return cuotaMantenimiento;
}

public void comisiones()
{
    // Se aplican mensualmente por el mantenimiento de la cuenta
    GregorianCalendar fechaActual = new GregorianCalendar();
    int dia = fechaActual.get(Calendar.DAY_OF_MONTH);

    if (dia == 1) reintegro(cuotaMantenimiento);
}

public double intereses()
{
    GregorianCalendar fechaActual = new GregorianCalendar();
    int dia = fechaActual.get(Calendar.DAY_OF_MONTH);

    if (dia != 1) return 0.0;
    // Acumular los intereses por mes sólo los días 1 de cada mes
    double interesesProducidos = 0.0;
    interesesProducidos = estado() * obtenerTipoDeInterés() / 1200.0;
    ingreso(interesesProducidos);

    // Devolver el interés mensual por si fuera necesario
    return interesesProducidos;
}
/////////////////////////////////////////////////////////////////

```

CCuentaAhorro es una subclase de la superclase *CCuenta*. Observe que para definir una subclase se añade a continuación del nombre de la misma la palabra reservada **extends** y el nombre de la superclase. En la definición de la subclase se describen las características adicionales que la distinguen de la superclase.

La capacidad de la clase *CCuenta* está soportada por:

Atributos	Métodos
nombre	constructores CCuenta
cuenta	asignarNombre
saldo	obtenerNombre
tipoDeInterés	asignarCuenta
	obtenerCuenta
	estado
	comisiones
	intereses
	ingreso
	reintegro
	asignarTipoDeInterés
	obtenerTipoDeInterés

La capacidad de la clase *CCuentaAhorro*, derivada de *CCuenta*, está soportada por los miembros heredados de *CCuenta* (en cursiva y no tachados) más los suyos:

Atributos	Métodos
nombre	<i>constructores CCuenta</i>
cuenta	<i>asignarNombre</i>
saldo	<i>obtenerNombre</i>
tipoDeInterés	<i>asignarCuenta</i>
	<i>obtenerCuenta</i>
	<i>estado</i>
	<i>comisiones</i>
	<i>intereses</i>
	<i>ingreso</i>
	<i>reintegro</i>
	<i>asignarTipoDeInterés</i>
	<i>obtenerTipoDeInterés</i>
cuotaMantenimiento	constructores CCuentaAhorro
	asignarCuotaManten
	obtenerCuotaManten
	comisiones
	intereses

Observe que los constructores de la clase *CCuenta* no se heredan, puesto que cada clase define el suyo por omisión, y que los métodos *comisiones* e *intereses* quedan ocultos por los métodos del mismo nombre de la clase *CCuentaAhorro*. Un poco más adelante veremos que es posible referirse a un miembro oculto utilizando la palabra reservada *super* de Java: *super.miembro_oculto*.

Según el análisis anterior, mientras un posible objeto *CCuenta* contendría los datos *nombre*, *cuenta*, *saldo* y *tipoDeInterés*, un objeto *CCuentaAhorro* contiene los datos *nombre*, *cuenta*, *saldo*, *tipoDeInterés* y *cuotaMantenimiento*.

Escribamos ahora una pequeña aplicación basada en una clase *Test* que cree un objeto *CCuentaAhorro*:

```
public class Test
{
    public static void main(String[] args)
    {
        CCuentaAhorro cliente01 = new CCuentaAhorro();
        cliente01.asignarNombre("Un nombre");
        cliente01.asignarCuenta("Una cuenta");
        cliente01.asignarTipoDeInterés(2.5);
        cliente01.asignarCuotaManten(300);
        cliente01.ingreso(1000000);
        cliente01.reintegro(500000);
        cliente01.comisiones();

        // cliente01 no puede acceder a los miembros privados, como
        // cuenta.
    }
}
```

Partimos del hecho de que las clases *CCuenta* y *CCuentaAhorro* pertenecen al mismo paquete que la clase *Test*. Entonces, un “objeto”, como *cliente01*, de la clase *CCuentaAhorro* puede invocar a cualquiera de los métodos públicos, protegidos y predeterminados de *CCuentaAhorro* y de *CCuenta*, pero no tiene acceso a sus miembros privados. Si las clases *CCuentaAhorro* y *CCuenta* pertenecieran a otro paquete, la clase *Test* sólo tendría acceso a los miembros públicos.

Los “métodos” de una subclase no tienen acceso a los miembros privados de su superclase, pero sí lo tienen a sus miembros protegidos y públicos; y si la subclase pertenece al mismo paquete que la superclase, también tiene acceso a sus miembros predeterminados. Por ejemplo, el método *comisiones* de la clase *CCuentaAhorro* no puede acceder al atributo *saldo* de la clase *CCuenta* porque es privado, pero sí puede acceder a su método público *reintegro*.

```
public void comisiones()
{
    // Se aplican mensualmente por el mantenimiento de la cuenta
    GregorianCalendar fechaActual = new GregorianCalendar();
    int dia = fechaActual.get(Calendar.DAY_OF_MONTH);

    if (dia == 1) reintegro(cuotaMantenimiento);
}
```

Esta restricción puede sorprender, pero es así para imponer la encapsulación. Si una subclase tuviera acceso a los miembros privados de su superclase, entonces cualquiera podría acceder a los miembros privados de una clase, simplemente derivando una clase de ella. Consecuentemente, si una subclase quiere acceder a los miembros privados de su superclase, debe hacerlo a través de la interfaz pública, protegida, o predeterminada en su caso, de dicha superclase.

ATRIBUTOS CON EL MISMO NOMBRE

Como sabemos, una subclase puede acceder directamente a un atributo público, protegido, o predeterminado en su caso, de su superclase. Qué sucede si definimos en la subclase uno de estos atributos, con el mismo nombre que tiene en la superclase. Por ejemplo, supongamos que una clase *ClaseA* define un atributo identificado por *atributo_x*, que después redefinimos en una subclase *ClaseB*:

```
class ClaseA
{
    public int atributo_x = 1;

    public int método_x()
    {
        return atributo_x * 10;
    }

    public int método_y()
    {
        return atributo_x + 100;
    }
}

class ClaseB extends ClaseA
{
    protected int atributo_x = 2;

    public int método_x()
    {
        return atributo_x * -10;
    }
}
```

Ahora, la definición del atributo *atributo_x* en la subclase oculta la definición del atributo con el mismo nombre en la superclase. Por lo tanto, la última línea de código del ejemplo siguiente devolverá el valor de *atributo_x* de *ClaseB*. Si este atributo no hubiera sido definido en la subclase, entonces el valor devuelto sería el valor de *atributo_x* de la superclase. Se puede observar que el tipo de control de acceso puede modificarse en cualquier sentido.

```

public class Test
{
    public static void main(String[] args)
    {
        ClaseB objClaseB = new ClaseB();

        System.out.println(objClaseB.atributo_x); // escribe 2
        System.out.println(objClaseB.método_y()); // escribe 101
        System.out.println(objClaseB.método_x()); // escribe -20
    }
}

```

Ahora bien, ¿cómo procederíamos si el método referenciado por el *método_x* de la clase *ClaseB* tuviera que acceder obligatoriamente al dato *atributo_x* de la superclase? La solución es sencilla: utilizar para ese atributo nombres diferentes en la superclase y en la subclase. No obstante, aun habiendo utilizado el mismo nombre, tenemos una alternativa de acceso: utilizar la palabra reservada **super**. Por ejemplo:

```

public int método_x()
{
    return super.atributo_x * -10;
}

```

Como se puede ver, podemos referirnos al dato *atributo_x* de la superclase con la expresión:

`super.atributo_x`

Asimismo, podríamos referirnos al dato *atributo_x* de la subclase con la expresión:

`this.atributo_x`

En cambio, la expresión siguiente hace referencia al dato *atributo_x* de la *ClaseA*:

`((ClaseA)this).atributo_x`

La técnica de realizar una conversión explícita u obligada es la que tendremos que utilizar si necesitamos referirnos a un miembro oculto perteneciente a una clase por encima de la superclase (una clase base indirecta).

REDEFINIR MÉTODOS DE LA SUPERCLASE

Cuando se invoca a un método en respuesta a un mensaje recibido por un objeto, Java busca su definición en la clase del objeto. La definición del método que allí se encuentra puede pertenecer a la propia clase o puede haber sido heredada de alguna de sus superclases (esto último equivale a decir que si no la encuentra, Java sigue buscando hacia arriba en la jerarquía de clases hasta que la localice).

Sin embargo, puede haber ocasiones en que deseemos que un objeto de una subclase responda al mismo método heredado de su superclase pero con un comportamiento diferente. Esto implica redefinir en la subclase el método heredado de su superclase.

Redefinir un método heredado significa volverlo a escribir en la subclase con el mismo nombre, la misma lista de parámetros y el mismo tipo del valor retornado que tenía en la superclase; su cuerpo será adaptado a las necesidades de la subclase. Esto es lo que se ha hecho con el *método_x* del ejemplo expuesto en el apartado anterior.

Se puede observar que este método ha sido redefinido en la *ClaseB* para que realice unos cálculos diferentes a los que realizaba en la *ClaseA*.

En el método **main** de la clase *Test* del ejemplo anterior, se creó un objeto *objClaseB* y se invocó a su *método_y*. Como la clase del objeto, *ClaseB*, no define este método, Java ejecuta el heredado. Asimismo, se invocó a su *método_x*; en este caso, existe una definición para este método, que es la que se ejecuta.

Cuando en una subclase se redefine un método de una superclase, se oculta el método de la superclase, pero no las sobrecargas que existan del mismo en dicha superclase. Si el método se redefine en la subclase con distinto tipo o número de parámetros, el método de la superclase no se oculta, sino que se comporta como una sobrecarga de ese método. Por ejemplo, el *método_x* tal cual lo hemos redefinido en la subclase oculta al método del mismo nombre de la superclase. Pero si lo hubiéramos definido con distinto número de parámetros, por ejemplo con uno, según se muestra a continuación, sería una sobrecarga.

```
public int método_x(int a) // método de ClaseB
{
    return atributo_x * -a;
}
```

En el caso de que el método heredado por la subclase sea abstracto, como sucede en nuestro ejemplo acerca de las cuentas bancarias, es obligatorio redefinirlo, de lo contrario la subclase debería ser declarada también abstracta.

A diferencia de lo que ocurría con los atributos redefinidos, el control de acceso de un método que se redefine no puede modificarse en cualquier sentido. Como regla, no se puede redefinir un método en una subclase y hacer que su control de acceso sea más restrictivo que el original. El orden de los tipos de control de acceso de más a menos restrictivo es así: **private**, **predeterminado**, **protected** y **public**. Aplicando esta regla, un método que en la superclase sea **protected**, en la subclase podrá ser redefinido como **protected** o **public**; si es **public** sólo podrá ser redefinido como **public**. Si es **private** no tiene sentido hablar de redefinición, porque no puede ser accedido nada más que desde su propia clase.

Para acceder a un método de la superclase que ha sido redefinido en la subclase, igual que se expuso para los atributos, tendremos que utilizar la palabra reservada **super**. Por ejemplo, suponga que añadimos el siguiente método a la *ClaseB*:

```
public int método_z()
{
    atributo_x = super.atributo_x + 3;
    return super.método_x() + atributo_x;
}
```

Como se puede observar, podemos referirnos al *método_x* de la superclase con la expresión:

```
super.método_x()
```

Es importante resaltar que **super** puede ser utilizado sólo desde dentro de la clase que proporciona los miembros redefinidos.

Asimismo, como ya vimos cuando se expuso **this**, podríamos referirnos al *método_x* de la subclase así:

```
this.método_x()
```

En cambio, una expresión como la siguiente es válida para el compilador, pero, de acuerdo con lo que aprendió en el apartado anterior, no producirá los resultados que quizás usted esperaba.

```
// Método de la ClaseB
public int método_z()
{
    // ...
    return ((ClaseA)this).método_x() + atributo_x;
}
```

En el ejemplo anterior, **this** lógicamente es una referencia a un objeto de la *ClaseB*; esta referencia es convertida explícitamente a un objeto de la *ClaseA*; pero, independientemente de esto, en Java, el método invocado, cuando se trate de un método redefinido, siempre pertenece a la clase del objeto no de la referencia; por lo tanto, el *método_x* invocado será el de la *ClaseB*.

CONSTRUCTORES DE LAS SUBCLASES

Sabemos que cuando se crea un objeto de una clase se invoca a su constructor. También sabemos que los constructores de la superclase no son heredados por sus subclases. En cambio, cuando se crea un objeto de una subclase, se invoca a su constructor, que a su vez invoca al constructor sin parámetros de la superclase, que a su vez invoca al constructor de su superclase, y así sucesivamente.

Lo anteriormente expuesto se traduce en que primero se ejecutan los constructores de las superclases de arriba a abajo en la jerarquía de clases y finalmente el de la subclase. Esto sucede así, porque una subclase contiene todos los atributos de su superclase, y todos tienen que ser iniciados, razón por la que el constructor de la subclase tiene que llamar implícita o explícitamente al de la superclase.

Sin embargo, cuando se hayan definido constructores con parámetros tanto en las subclases como en las superclases, tal vez se desee construir un objeto de la subclase iniciándolo con unos valores determinados. En este caso, la definición ya conocida para los constructores de una clase cualquiera se extiende ahora para permitir al constructor de la subclase invocar explícitamente al constructor de la superclase. Esto se hace utilizando la palabra reservada **super**:

```
nombre_subclase( lista de parámetros )
{
    super( lista de parámetros );
    // cuerpo del constructor de la subclase
}
```

En la definición genérica anterior correspondiente a un constructor con parámetros de una subclase, se observa, por una parte, la utilización de la palabra reservada **super** para invocar al constructor de la superclase, y por otra, el cuerpo del constructor de la subclase. Cuando desde un constructor de una subclase se invoque al constructor de su superclase, esta línea tiene que ser la primera.

Se puede observar que la sintaxis y los requerimientos son análogos a los utilizados con **this** cuando se llama a otro constructor de la misma clase.

Si la superclase no tiene un constructor de forma explícita o tiene uno que no requiere parámetros, no se necesita invocarlo explícitamente, ya que Java lo invocará automáticamente mediante **super()** sin argumentos. Por el contrario, si es necesario invocarlo cuando se trate de un constructor con parámetros, para poder así pasar los argumentos necesarios en la llamada.

Por ejemplo, aplicando la teoría expuesta, vamos a añadir a la clase *CCuentaAhorro* un constructor con parámetros. ¿Cuántos parámetros debe tener este constructor para iniciar todos los atributos del un objeto *CCuentaAhorro*? Pues tantos como atributos heredados y propios tenga la clase; en nuestro caso un objeto *CCuentaAhorro* contiene los atributos *nombre*, *cuenta*, *saldo*, *tipoDeInterés* y *cuotaMantenimiento*. Según esto el constructor podría ser así:

```
public CCuentaAhorro(String nom, String cue, double sal,
                      double tipo, double mant)
{
    super(nom, cue, sal, tipo); // invoca al constructor CCuenta
    asignarCuotaManten(mant); // inicia cuotaMantenimiento
}
```

La primera línea del método constructor anterior llama al constructor de *CCuenta*, superclase de *CCuentaAhorro*. Lógicamente, la clase *CCuenta* debe tener un constructor con cuatro parámetros del tipo de los argumentos especificados. La segunda línea invoca al método *asignarCuotaManten* para iniciar el atributo *cuotaMantenimiento* de *CCuentaAhorro*.

Evidentemente, si sólo se desea iniciar algunos de los atributos de un objeto, hay que escribir los constructores adecuados tanto en la subclase como en la superclase.

De acuerdo con los métodos constructores definidos en la clase *CCuentaAhorro*, son declaraciones válidas las siguientes:

```
public class Test
{
    public static void main(String[] args)
    {
        CCuentaAhorro cliente01 = new CCuentaAhorro();
        CCuentaAhorro cliente02 = new CCuentaAhorro("Un nombre",
                                                      "Una cuenta", 1000000, 3.5, 300);
        // ...
    }
}
```

En este ejemplo, la sentencia primera requiere en *CCuentaAhorro* un constructor sin parámetros y en *CCuenta* otro. En cambio, la segunda sentencia re-

quiere en *CCuentaAhorro* un constructor con parámetros y en *CCuenta* otro que se pueda invocar como se indica a continuación, con el fin de iniciar los atributos definidos en la superclase, con los valores pasados como argumentos.

```
super(nombre, cuenta, saldo, tipoInterés);
```

Según lo expuesto, cuando se crea un objeto de una subclase, por ejemplo *cliente01* o *cliente02*, primero se construye la porción del objeto correspondiente a su superclase y a continuación la porción del objeto correspondiente a su subclase. Esto es una forma lógica de operar, ya que permite al constructor de la subclase hacer referencia a los atributos de su superclase que ya han sido iniciados.

Según lo expuesto, los objetos de una subclase son construidos de abajo hacia arriba; esto es, la pila de llamadas relativas a los constructores de las clases involucradas crece hasta llegar a la clase raíz en la jerarquía de clases; en este instante, comienza a ejecutarse el constructor de esta superclase: primero se construyen sus atributos ejecutando, cuando sea necesario, los constructores de los mismos, y después, se pasa a ejecutar el cuerpo del constructor de dicha superclase; y a continuación se ejecuta el cuerpo del constructor de la subclase. Este orden se aplica recursivamente por cada constructor de cada una de las clases.

DESTRUCTORES DE LAS SUBCLASES

Acabamos de ver que en Java, el constructor de una subclase invoca automáticamente al constructor sin parámetros de su superclase. En cambio, con los destructores (métodos **finalize**) no ocurre lo mismo.

Por ejemplo, si definimos en una subclase un método **finalize** para liberar los recursos asignados por dicha clase, debemos redefinir el método **finalize** en la superclase para liberar también los recursos asignados por ella. Pero si el método **finalize** de la subclase no invoca explícitamente al método **finalize** de la superclase, este último nunca será ejecutado y los recursos asignados por la superclase no serán liberados. ¿Cuándo debemos invocar al método **finalize** de la superclase? El mejor lugar para hacerlo es en la última línea del método **finalize** de la subclase, porque como la parte del objeto de la subclase se ha construido una vez que estaba construida la parte del objeto de la superclase, en más de una ocasión los vínculos existentes entre una y otra parte exigirán deshacer lo construido, justo en el orden inverso.

Para ver prácticamente la forma de implementar los destructores, volvamos al ejemplo anteriormente expuesto con la *ClaseA* y la *ClaseB*, y añadamos un destructor a cada una de ellas que supuestamente hace algo.

```
class ClaseA
{
    public int atributo_x = 1;

    public int método_x()
    {
        return atributo_x * 10;
    }

    public int método_y()
    {
        return atributo_x + 100;
    }

    protected void finalize() throws Throwable // destructor
    {
        System.out.println("Recursos de ClaseA liberados");
    }
}

class ClaseB extends ClaseA
{
    protected int atributo_x = 2;

    public int método_x()
    {
        return atributo_x * -10;
    }

    public int método_z()
    {
        atributo_x = super.atributo_x + 3;
        return super.método_x() + atributo_x;
    }

    protected void finalize() throws Throwable // destructor
    {
        System.out.println("Recursos de ClaseB liberados");
        super.finalize();
    }
}

public class Test
{
    public static void main(String[] args)
    {
        ClaseB objClaseB = new ClaseB();
        // ...
        objClaseB = null;
```

```

// Ejecutar el recolector de basura
Runtime runtime = Runtime.getRuntime();
runtime.gc();
runtime.runFinalization();
}
}

```

Para ver cómo son invocados los destructores, el método **main** de la clase *Test* crea un objeto de la *ClaseB* referenciado por *objClaseB* y cuando finaliza el trabajo con el mismo asigna a la variable *objClaseB* el valor **null** con la intención de enviar el objeto referenciado por ella a la basura. Finalmente fuerza al recolector de basura a que recoja la basura, lo que provocará la ejecución de los destructores: primero el de la *ClaseB* y después el de la *ClaseA*.

Puesto que Java proporciona para cada clase que definamos un método **finalize** heredado de la clase **Object**, una programación segura aconseja redefinir este método en cada una de las subclases que escribamos, aunque no haga nada; simplemente con la intención de invocar al método **finalize** de la superclase, por si alguna versión futura de la misma incluye un método **finalize**.

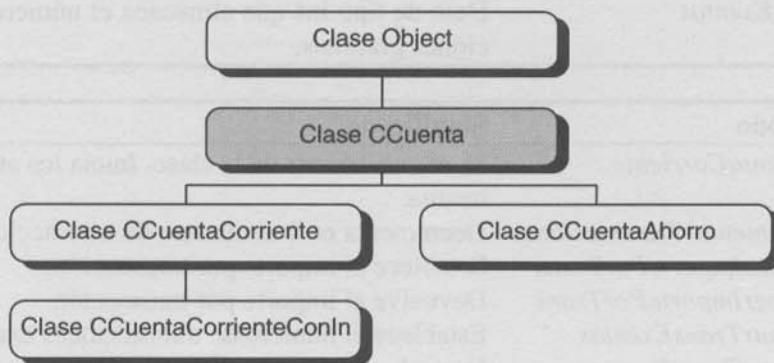
```

protected void finalize() throws Throwable // destructor
{
    // Ninguna operación
    super.finalize(); // invocar al método finalize de la superclase
}

```

JERARQUÍA DE CLASES

Una subclase puede asimismo ser una superclase de otra clase, y así sucesivamente. En la siguiente figura se puede ver esto con claridad:



El conjunto de clases así definido da lugar a una *jerarquía de clases*. Cuando cada subclase lo es de una sola superclase, como ocurre en Java, la estructura jerárquica recibe el nombre de *árbol de clases*.

La raíz del árbol es la clase que representa el tipo más general, y las clases terminales en el árbol (nodos hoja) representan los tipos más especializados.

Las reglas que podemos aplicar para manipular la subclase *CCuentaCorriente* de la superclase *CCuenta* o la subclase *CCuentaCorrienteConIn* de la superclase *CCuentaCorriente* son las mismas que hemos aplicado anteriormente para la subclase *CCuentaAhorro* de la superclase *CCuenta*, y lo mismo diremos para cualquier otra subclase que deseemos añadir. Esto quiere decir que para implementar una subclase como *CCuentaCorrienteConIn*, nos es suficiente con conocer a fondo su superclase *CCuentaCorriente* sin importarnos *CCuenta*.

Observe que la clase *CCuenta* actúa como superclase de más de una clase, concretamente de las clases *CCuentaAhorro* y *CCuentaCorriente*.

Como ejemplo, vamos a completar la jerarquía de clases expuesta con las clases que faltan: *CCuentaCorriente* y *CCuentaCorrienteConIn*.

La clase *CCuentaCorriente* es una nueva clase que hereda de la clase *CCuenta*. Por lo tanto, tendrá todos los miembros de su superclase, a los que añadiremos los siguientes:

Atributo	Significado
<i>transacciones</i>	Dato de tipo int que almacena el número de transacciones efectuadas sobre esa cuenta.
<i>importePorTrans</i>	Dato de tipo double que almacena el importe que la entidad bancaria cobrará por cada transacción.
<i>transExentas</i>	Dato de tipo int que almacena el número de transacciones gratuitas.

Método	Significado
<i>CCuentaCorriente</i>	Es el constructor de la clase. Inicia los atributos de la misma.
<i>decrementarTransacciones</i>	Decrementa en 1 el número de transacciones.
<i>asignarImportePorTrans</i>	Establece el importe por transacción.
<i>obtenerImportePorTrans</i>	Devuelve el importe por transacción.
<i>asignarTransExentas</i>	Establece el número de transacciones exentas.
<i>obtenerTransExentas</i>	Devuelve el número de transacciones exentas.
<i>ingreso</i>	Añade la cantidad especificada al saldo actual de la cuenta e incrementa el número de transacciones.

<i>reintegro</i>	Resta la cantidad especificada del saldo actual de la cuenta e incrementa el número de transacciones.
<i>comisiones</i>	Se ejecuta los días uno de cada mes para cobrar el importe de las transacciones efectuadas que no estén exentas y pone el número de transacciones a cero.
<i>intereses</i>	Se ejecuta los días uno de cada mes para calcular el importe correspondiente a los intereses/mes producidos y añadirlo al saldo. Hasta 3000 euros al 0.5%. El resto al interés establecido.

Aplicando la teoría expuesta hasta ahora y procediendo de forma similar a como lo hicimos para construir la subclase *CCuentaAhorro*, la definición de la clase *CCuentaCorriente* es la siguiente:

```

import java.util.*;

///////////////////////////////
// Clase CCuentaCorriente: clase derivada de CCuenta
//
public class CCuentaCorriente extends CCuenta
{
    // Atributos
    private int transacciones;
    private double importePorTrans;
    private int transExentas;

    // Métodos
    public CCuentaCorriente() {} // constructor sin parámetros

    public CCuentaCorriente(String nom, String cue, double sal,
                           double tipo, double imptrans, int transex)
    {
        super(nom, cue, sal, tipo); // invoca al constructor CCuenta
        transacciones = 0;           // inicia transacciones
        asignarImportePorTrans(imptrans); // inicia importePorTrans
        asignarTransExentas(transex);   // inicia transExentas
    }

    public void decrementarTransacciones()
    {
        transacciones--;
    }

    public void asignarImportePorTrans(double imptrans)
    {
        if (imptrans < 0)
        {
            System.out.println("Error: cantidad negativa");
        }
    }
}

```

```
    }
    return;
}
importePorTrans = imprtrans;
}

public double obtenerImportePorTrans()
{
    return importePorTrans;
}

public void asignarTransExentas(int transex)
{
    if (transex < 0)
    {
        System.out.println("Error: cantidad negativa");
        return;
    }
    transExentas = transex;
}

public int obtenerTransExentas()
{
    return transExentas;
}

public void ingreso(double cantidad)
{
    super.ingreso(cantidad);
    transacciones++;
}

public void reintegro(double cantidad)
{
    super.reintegro(cantidad);
    transacciones++;
}

public void comisiones()
{
    // Se aplican mensualmente por el mantenimiento de la cuenta
    GregorianCalendar fechaActual = new GregorianCalendar();
    int dia = fechaActual.get(Calendar.DAY_OF_MONTH);
    if (dia == 1)
    {
        int n = transacciones - transExentas;
        if (n > 0) reintegro(n * importePorTrans);
        transacciones = 0;
    }
}
```

```

public double intereses()
{
    GregorianCalendar fechaActual = new GregorianCalendar();
    int día = fechaActual.get(Calendar.DAY_OF_MONTH);

    if (día != 1) return 0.0;

    // Acumular los intereses por mes sólo los días 1 de cada mes
    double interesesProducidos = 0.0;
    // Hasta 3000 euros al 0.5%. El resto al interés establecido.
    if (estado() <= 3000)
        interesesProducidos = estado() * 0.5 / 1200.0;
    else
    {
        interesesProducidos = 3000 * 0.5 / 1200.0 +
            (estado() - 3000) * obtenerTipoDeInterés() / 1200.0;
    }
    ingreso(interesesProducidos);
    // Este ingreso no debe incrementar las transacciones
    decrementarTransacciones();

    return interesesProducidos;
}
}
////////////////////////////////////////////////////////////////

```

Observe que el constructor de la clase *CCuentaCorriente* tiene los parámetros necesarios para iniciar sus datos miembro, excepto *transacciones* que inicialmente vale 0, y los heredados de su superclase. El cuerpo del constructor consta de la llamada al constructor de su superclase y de las llamadas a los métodos de la propia clase que permiten iniciar de forma segura los atributos de la misma. También se ha implementado un constructor sin parámetros.

Procediendo de forma similar a como lo hemos hecho para las clases *CCuentaAhorro* y *CCuentaCorriente*, construimos a continuación la clase *CCuentaCorrienteConIn* (cuenta corriente con intereses) derivada de *CCuentaCorriente*.

Supongamos que este tipo de cuenta se ha pensado para que acumule intereses de forma distinta a los otros tipos de cuenta, pero para obtener una rentabilidad mayor respecto a *CCuentaCorriente*.

Digamos que se trata de una cuenta de tipo *CCuentaCorriente* que precisa un saldo mínimo de 3000 euros para que pueda acumular intereses. Según esto, *CCuentaCorrienteConIn*, además de los miembros heredados, sólo precisa implementar sus constructores y variar el método *intereses*:

Método	Significado
<i>CCuentaCorriente</i>	Es el constructor de la clase. Inicia los atributos de la misma.
<i>intereses</i>	Permite calcular el importe/mes correspondiente a los intereses producidos. Precisa un saldo mínimo de 3000 euros.

La definición correspondiente a esta clase se expone a continuación:

```

import java.util.*;

///////////////////////////////
// Clase CCuentaCorrienteConIn: clase derivada de CCuentaCorriente
//
public class CCuentaCorrienteConIn extends CCuentaCorriente
{
    // Métodos
    public CCuentaCorrienteConIn() {} // constructor sin parámetros

    public CCuentaCorrienteConIn(String nom, String cue, double sal,
                                  double tipo, double imprans, int transex)
    {
        // Invocar al constructor de la superclase
        super(nom, cue, sal, tipo, imprans, transex);
    }

    public double intereses()
    {
        GregorianCalendar fechaActual = new GregorianCalendar();
        int dia = fechaActual.get(Calendar.DAY_OF_MONTH);

        if (dia != 1 || estado() < 3000) return 0.0;

        // Acumular interés mensual sólo los días 1 de cada mes
        double interesesProducidos = 0.0;
        interesesProducidos = estado() * obtenerTipoDeInterés() / 1200.0;
        ingreso(interesesProducidos);
        // Este ingreso no debe incrementar las transacciones
        decrementarTransacciones();

        // Devolver el interés mensual por si fuera necesario
        return interesesProducidos;
    }
}
/////////////////////////////

```

La clase *CCuenta* es la “superclase directa” (o simplemente superclase) de *CCuentaAhorro* y de *CCuentaCorriente*, y es una “superclase indirecta” para *CCuentaCorrienteConIn*. Mientras que una subclase tiene una única superclase

directa, puede tener varias superclases indirectas: todas las que haya en el camino para llegar desde su superclase hasta la clase raíz; esto es importante porque lo que una subclase hereda de su superclase, será heredado a su vez por una subclase de ella, y así sucesivamente.

Una subclase que redefina un método heredado sólo tiene acceso a su propia versión y a la publicada por su superclase directa. Por ejemplo, las clases *CCuenta* y *CCuentaCorriente* incluyen cada una su versión del método *ingreso*; y la subclase *CCuentaCorrienteConIn* hereda el método *ingreso* de *CCuentaCorriente*. Entonces, *CCuentaCorrienteConIn*, además de a su propia versión, sólo puede acceder a la versión de su superclase directa por medio de la palabra **super** (en este caso ambas versiones son la misma), pero no puede acceder a la versión de su superclase indirecta *CCuenta* (**super.super** no es una expresión admitida por el compilador Java).

Según lo expuesto las líneas de código:

```
ingreso(interesesProducidos);
decrementarTransacciones();
```

del método *intereses* de la clase *CCuentaCorriente* podrían ser sustituidas por la indicada a continuación, puesto que el método *ingreso* de *CCuenta* no actúa sobre las transacciones:

```
super.ingreso(interesesProducidos);
```

En cambio, en el método *intereses* de la clase *CCuentaCorrienteConIn* no podemos proceder de la misma forma porque desde esta clase no se puede acceder a la versión de *ingreso* de *CCuenta*.

A continuación se presenta una aplicación con algunos ejemplos de operaciones con objetos de las clases pertenecientes a la jerarquía construida:

```
public class Test
{
    public static void main(String[] args)
    {
        CCuentaAhorro cliente01 = new CCuentaAhorro(
            "Un nombre", "Una cuenta", 10000, 3.5, 30);

        System.out.println(cliente01.obtenerNombre());
        System.out.println(cliente01.obtenerCuenta());
        System.out.println(cliente01.estado());
        System.out.println(cliente01.obtenerTipoDeInterés());
        System.out.println(cliente01.intereses());
```

```

CCuentaCorrienteConIn cliente02 = new CCuentaCorrienteConIn();
cliente02.asignarNombre("cliente 02");
cliente02.asignarCuenta("1234567890");
cliente02.asignarTipoDeInterés(3.0);
cliente02.asignarTransExentas(0);
cliente02.asignarImportePorTrans(1.0);

cliente02.ingreso(20000);
cliente02.reintegro(10000);
cliente02.intereses();
cliente02.comisiones();
System.out.println(cliente02.obtenerNombre());
System.out.println(cliente02.obtenerCuenta());
System.out.println(cliente02.estado());
}
}
}

```

En la aplicación anterior se puede observar cómo el método **main** construye dos objetos: *cliente01* de la clase *CCuentaAhorro* y *cliente02* de la clase *CCuentaCorrienteConIn*.

Para construir *cliente01* se ha utilizado el constructor *CCuentaAhorro* con argumentos. Una vez construido, observe que responde a una serie de mensajes ejecutando los métodos del mismo nombre, unos heredados de su superclase, como *obtenerNombre*, y otros propios, como *intereses*.

En cambio, para construir *cliente02* se ha utilizado el constructor *CCuentaCorrienteConIn* sin argumentos. Una vez construido, puede también observar que responde a una serie de mensajes ejecutando los métodos del mismo nombre, unos heredados de su superclase directa, como *reintegro*, otros heredados de su superclase indirecta, como *asignarNombre*, y otros propios, como *intereses*.

Finalmente, indicar que aunque en ninguna clase de nuestra jerarquía han intervenido miembros **static**, su comportamiento en cuanto a la herencia se refiere es el mismo que el de los otros miembros, pero teniendo presente que son miembros de la clase; y si es necesario, cuando se trate de métodos, también pueden ser redefinidos, aunque, en este caso, el nombre de la clase indicará la versión del método que se invocará. Una advertencia, si definiera, por ejemplo, en *CCuenta* el atributo *tipoDeInterés static*, lógicamente se mantendría una única copia que utilizarían tanto los objetos de *CCuenta* como los de sus subclases.

REFERENCIAS A OBJETOS DE UNA SUBCLASE

Las referencias a objetos de una subclase pueden ser declaradas y manipuladas de la misma forma que las referencias a objetos de una clase cualquiera, tal y como

ya expusimos en el capítulo 4. Veamos algunos ejemplos basados en la jerarquía de clases que acabamos de construir:

```
public class Test
{
    public static void main(String[] args)
    {
        CCuentaCorriente cliente01;
        cliente01 = new CCuentaCorriente("cliente01", "1234567890",
                                         10000, 3.5, 30);
        // ...
    }
}
```

Este ejemplo declara una variable *cliente01* de la subclase *CCuentaCorriente* de *CCuenta*. Después crea un objeto de esa subclase y almacena su referencia en la variable *cliente01*. Una vez que disponemos de la referencia a un objeto podemos trabajar con él como lo hemos venido haciendo hasta ahora. Por ejemplo:

```
String cuenta = cliente01.obtenerCuenta();
double saldo = cliente01.estado();
```

Conversiones implícitas

El ejemplo anterior no aporta nada que nos sorprenda; operaciones como éas ya han sido expuestas anteriormente. Pero, qué pasaría si a la variable *cliente01* le asignamos la referencia a un objeto de la subclase *CCuentaCorrienteConIn* de *CCuentaCorriente*. Por ejemplo:

```
public class Test
{
    public static void main(String[] args)
    {
        CCuentaCorriente cliente01;
        cliente01 = new CCuentaCorrienteConIn("cliente01", "1234567890",
                                              10000, 3.5, 1.0, 6);
        String cuenta = cliente01.obtenerCuenta();
        double saldo = cliente01.estado();
        // ...
    }
}
```

Si ejecutamos este ejemplo, comprobaremos que los resultados obtenidos son los mismos que obtuvimos con el ejemplo anterior. Esto es así porque Java permite convertir implícitamente una referencia a un objeto de una subclase en una referencia a su superclase directa o indirecta. Veamos otro ejemplo:

```

CCuenta cliente;
CCuentaCorriente cliente01 =
    new CCuentaCorriente("cliente01", "1234567891",
                           10000, 3.5, 1.0, 6);
CCuentaCorrienteConIn cliente02 =
    new CCuentaCorrienteConIn("cliente02", "1234567892",
                               20000, 2.0, 1.0, 6);
cliente = cliente01;
String nombre = cliente.obtenerNombre();
// ...

cliente = cliente02;
String cuenta = cliente.obtenerCuenta();
// ...

```

El ejemplo anterior declara una referencia *cliente* de la clase *CCuenta*, la cual utilizamos después para referenciar indistintamente a un objeto *cliente01* de la clase *CCuentaCorriente*, o a un objeto *cliente02* de la clase *CCuentaCorrienteConIn*.

Cuando accedemos a un objeto por medio de una variable no del tipo del objeto, sino del tipo de alguna de sus superclases (directa o indirectas) según muestra el ejemplo anterior, es el tipo de la variable el que determina qué mensajes puede recibir el objeto referenciado; dicho de otra forma, es este tipo el que determina qué métodos pueden ser invocados por el objeto referenciado. ¿Cuáles son esos métodos? Pues los correspondientes al tipo de la variable que utilizamos para hacer referencia al objeto, no los de la clase del objeto.

Resumiendo: cuando accedemos a un objeto de una subclase por medio de una referencia a su superclase, ese objeto sólo puede ser manipulado por los métodos de su superclase. Por ejemplo:

```

CCuenta cliente;
CCuentaCorriente cliente01 =
    new CCuentaCorriente("cliente01", "1234567891",
                           10000, 3.5, 1.0, 6);
CCuentaCorrienteConIn cliente02 =
    new CCuentaCorrienteConIn("cliente02", "1234567892",
                               20000, 2.0, 1.0, 6);
cliente = cliente01;
cliente.asignarImportePorTrans(1.0); // error: no es un método de CCuenta
// ...

cliente = cliente02;
cliente.asignarTransExentas(10); // error: no es un método de CCuenta
// ...

```

Este último ejemplo define las mismas referencias y objetos que el anterior. Pero ahora observamos que un intento de acceder al método *asignarImportePorTrans* ocasiona un error. Esto es porque el tipo de la variable *cliente*, que es *CCuenta*, determina que el objeto referenciado por ella sólo puede recibir mensajes de la clase de dicha variable; dicho de otra forma, sólo puede ser manipulado por métodos de la clase *CCuenta* (propias y heredadas). Lo mismo diríamos respecto al mensaje *asignarTransExentas* enviado al objeto inicialmente referenciado por *cliente02* y finalmente, también por *cliente*.

En cambio, cuando se invoca a un método que está definido en la superclase y redefinido en sus subclases, la versión que se ejecuta depende de la clase del objeto referenciado, no del tipo de la variable que lo referencia. Por ejemplo:

```
CCuenta cliente;
CCuentaCorriente cliente01 =
    new CCuentaCorriente("cliente01", "1234567891",
                           10000, 3.5, 1.0, 6);
CCuentaCorrienteConIn cliente02 =
    new CCuentaCorrienteConIn("cliente02", "1234567892",
                               20000, 2.0, 1.0, 6);
double intereses;
cliente = cliente01;
intereses = cliente.intereses(); // CCuentaCorriente.intereses()
// ...

cliente = cliente02;
intereses = cliente.intereses(); // CCuentaCorrienteConIn.intereses()
// ...
```

Este ejemplo declara *cliente* de la clase *CCuenta*, la cual utilizamos después para referenciar indistintamente a un objeto *cliente01* de la clase *CCuentaCorriente*, o a un objeto *cliente02* de la clase *CCuentaCorrienteConIn*. Por otra parte, el método *intereses* está definido en la superclase *CCuenta* y redefinido en sus subclases *CCuentaCorriente* y *CCuentaCorrienteConIn*. Por lo tanto, la expresión *cliente.intereses()* invocará a *CCuentaCorriente.intereses()* si *cliente* señala a un objeto *CCuentaCorriente*, e invocará a *CCuentaCorrienteConIn.intereses()* si *cliente* señala a un objeto *CCuentaCorrienteConIn*.

Conversiones explícitas

La conversión contraria, de una referencia a un objeto de la superclase a una referencia a su subclase, no se puede hacer, aunque se fuerce a ello utilizando una construcción *cast*, excepto cuando el objeto al que se tiene acceso a través de la referencia a la superclase es un objeto de la subclase. Por ejemplo:

```

CCuentaCorriente cliente01 =
    new CCuentaCorriente("cliente01", "1234567891",
                           10000, 3.5, 1.0, 6);
CCuentaCorrienteConIn cliente;
cliente = cliente01; // error de compilación: conversión implícita
                     // no permitida
// La siguiente línea durante la ejecución lanza una excepción
// de tipo ClassCastException, debido a que la conversión
// explícita requerida no se permite
cliente = (CCuentaCorrienteConIn)cliente01;
// La línea anterior sería válida si cliente01 referenciara a un
// objeto de la clase de cliente, esto es, CCuentaCorrienteConIn.

```

POLIMORFISMO

La utilización de subclases y de métodos definidos en una clase y redefinidos en sus clases derivadas es frecuentemente denominada *programación orientada a objetos*. En cambio, la facultad de llamar a una variedad de métodos utilizando exactamente el mismo medio de acceso, proporcionada por los métodos redefinidos en las subclases, es a veces denominada *polimorfismo*.

La palabra “polimorfismo” significa “la facultad de asumir muchas formas”, refiriéndose a la facultad de llamar a muchos métodos diferentes utilizando una única sentencia.

Recuerde que cuando se invoca a un método que está definido en la superclase y redefinido en sus subclases, la versión que se ejecuta depende de la clase del objeto referenciado, no del tipo de la variable que lo referencia.

Asimismo, sabemos que una referencia a una subclase puede ser convertida implícitamente por Java en una referencia a su superclase directa o indirecta. Esto significa que es posible referirse a un objeto de una subclase utilizando una variable del tipo de su superclase.

Según lo expuesto, y en un intento de buscar una codificación más genérica, pensemos en una matriz de referencias en la que cada elemento señale a un objeto de alguna de las subclases de la jerarquía construida anteriormente. ¿De qué tipo deben ser los elementos de la matriz? Según el párrafo anterior deben de ser de la clase *CCuenta*; de esta forma ellos podrán almacenar indistintamente referencias a objetos de cualquiera de las subclases. Por ejemplo:

```

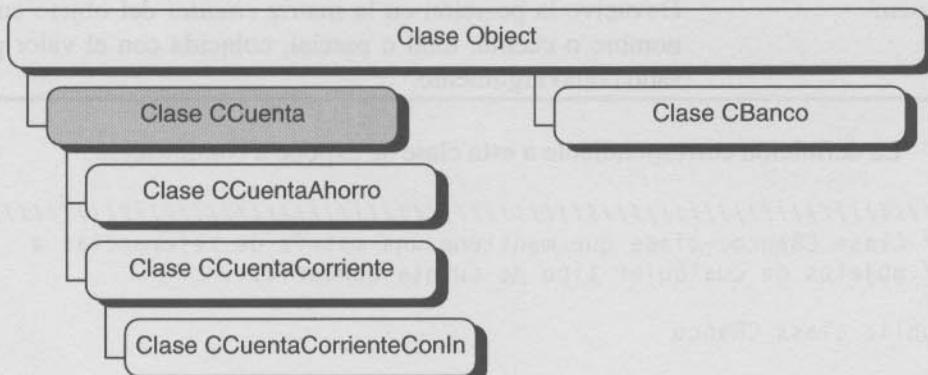
public class Test
{
    public static void main(String[] args)
    {

```

```
CCuenta[] cliente = new CCuenta[100];
// Crear objetos y guardar sus referencias en la matriz
cliente[0] = new CCuentaAhorro("cliente00", "3000123450",
                               10000, 2.5, 30);
cliente[1] = new CCuentaCorriente("cliente01", "6000123450",
                                  10000, 2.0, 1.0, 6);
cliente[2] = new CCuentaCorrienteConIn("cliente02",
                                         "4000123450", 10000, 3.5, 1.0, 6);
for (int i = 0; cliente[i] != null; i++)
{
    System.out.print(cliente[i].obtenerNombre() + ": ");
    System.out.println(cliente[i].intereses());
}
```

Este ejemplo define una matriz *cliente* de tipo *CCuenta* con 100 elementos que Java inicia con el valor **null**. Después crea un objeto de una de las subclases y almacena su referencia en el primer elemento de la matriz; aquí Java realizará una conversión implícita del tipo de la referencia devuelta por **new** al tipo *CCuenta*. Este proceso se repetirá para cada objeto nuevo que deseemos crear (en nuestro caso tres veces). Finalmente, utilizando un bucle mostramos el nombre del cliente y los intereses que le corresponderán por mes. Pregunta: ¿en cuál de las dos líneas de este bucle se aplica la definición de polimorfismo? Lógicamente en la última porque, según lo estudiado hasta ahora, invoca a las distintas definiciones del método *intereses* utilizando el mismo medio de acceso: una referencia a *CCuenta*.

Como ejemplo, vamos a escribir un programa que cree un objeto que represente a una entidad bancaria con un cierto número de clientes. Este objeto estará definido por una clase que denominaremos *CBanco* y los clientes serán objetos de alguna de las clases de la jerarquía construida en los apartados anteriores.



La estructura de datos que represente el banco tiene que ser capaz de almacenar objetos *CCuentaAhorro*, *CCuentaCorriente* y *CCuentaCorrienteConIn*. Sabiendo que cualquier referencia a un objeto de una subclase puede convertirse implícitamente en una referencia a un objeto de su superclase, la estructura idónea es una matriz de referencias a la superclase *CCuenta*. Esta matriz será dinámica; esto es, aumentará en un elemento cuando se añada un objeto de alguna de las subclases y disminuirá en uno cuando se elimine; inicialmente tendrá 0 elementos. Según esto, la clase *CBanco*, que no pertenece a nuestra jerarquía, tendrá los atributos y métodos que se exponen a continuación:

Atributo	Significado
<i>clientes</i>	Matriz de referencias de tipo <i>CCuenta</i> .
<i>nElementos</i>	Número de elementos de la matriz.
Método	Significado
<i>CBanco</i>	Es el constructor de la clase. Inicia la matriz <i>clientes</i> con cero elementos.
<i>unElementoMás</i>	Añade un elemento vacío (null) al final de la matriz, incrementando su longitud en 1.
<i>unElementoMenos</i>	Elimina un elemento cuyo valor sea null , decrementando su longitud en 1.
<i>insertarCliente</i>	Asigna un objeto de alguna de las subclases de <i>CCuenta</i> al elemento <i>i</i> de la matriz <i>clientes</i> .
<i>clienteEn</i>	Devuelve el objeto que está en la posición <i>i</i> de la matriz <i>clientes</i> .
<i>longitud</i>	Devuelve la longitud de la matriz.
<i>eliminar</i>	Elimina el objeto que coincide con el número de cuenta pasado como argumento, poniendo el elemento correspondiente de la matriz a valor null .
<i>buscar</i>	Devuelve la posición en la matriz <i>clientes</i> del objeto cuyo nombre o cuenta, total o parcial, coincide con el valor pasado como argumento.

La definición correspondiente a esta clase se expone a continuación:

```
///////////////////////////////
// Clase CBanco: clase que mantiene una matriz de referencias a
// objetos de cualquier tipo de cuenta bancaria.
//
public class CBanco
{
    private CCuenta[] clientes; // matriz de objetos
    private int nElementos; // número de elementos de la matriz
```

```
public CBanco()
{
    // Crear una matriz vacía
    nElementos = 0;
    clientes = new CCuenta[nElementos];
}

private void unElementoMás(CCuenta[] clientesActuales)
{
    nElementos = clientesActuales.length;
    // Crear una matriz con un elemento más
    clientes = new CCuenta[nElementos + 1];
    // Copiar los clientes que hay actualmente
    for ( int i = 0; i < nElementos; i++ )
        clientes[i] = clientesActuales[i];
    nElementos++;
}

private void unElementoMenos(CCuenta[] clientesActuales)
{
    if (clientesActuales.length == 0) return;
    int k = 0;
    nElementos = clientesActuales.length;
    // Crear una matriz con un elemento menos
    clientes = new CCuenta[nElementos - 1];
    // Copiar los clientes no nulos que hay actualmente
    for ( int i = 0; i < nElementos; i++ )
        if (clientesActuales[i] != null)
            clientes[k++] = clientesActuales[i];
    nElementos--;
}

public void insertarCliente( int i, CCuenta objeto )
{
    // Asignar al elemento i de la matriz, un nuevo objeto
    if (i >= 0 && i < nElementos)
        clientes[i] = objeto;
    else
        System.out.println("Índice fuera de límites");
}

public CCuenta clienteEn( int i )
{
    // Devolver la referencia al objeto i de la matriz
    if (i >= 0 && i < nElementos)
        return clientes[i];
    else
    {
        System.out.println("Índice fuera de límites");
        return null;
    }
}
```

```
    }

}

public int longitud() { return nElementos; }

public void añadir(CCuenta obj)
{
    // Añadir un objeto a la matriz
    unElementoMás(clientes);
    insertarCliente( nElementos - 1, obj );
}

public boolean eliminar(String cuenta)
{
    // Buscar la cuenta y eliminar el objeto
    for ( int i = 0; i < nElementos; i++ )
        if (cuenta.compareTo(clientes[i].obtenerCuenta()) == 0)
    {
        clientes[i] = null; // enviar el objeto a la basura
        unElementoMenos(clientes);
        return true;
    }
    return false;
}

public int buscar(String str, int pos)
{
    // Buscar un objeto y devolver su posición
    String nom, cuen;
    if (str == null) return -1;
    if (pos < 0) pos = 0;
    for ( int i = pos; i < nElementos; i++ )
    {
        // Buscar por el nombre
        nom = clientes[i].obtenerNombre();
        if (nom == null) continue;
        // ¿str está contenida en nom?
        if (nom.indexOf(str) > -1)
            return i;
        // Buscar por la cuenta
        cuen = clientes[i].obtenerCuenta();
        if (cuen == null) continue;
        // ¿str está contenida en cuen?
        if (cuen.indexOf(str) > -1)
            return i;
    }
    return -1;
}
//////////
```

Analizando la clase *CBanco*, observamos que su constructor inicia la matriz *clientes* con 0 elementos; que añadir un objeto (un cliente) a la matriz se hace en dos pasos: uno, incrementar la matriz en un elemento, y dos, asignar la referencia al objeto al nuevo elemento de la matriz; que eliminar un objeto de la matriz también se hace en dos pasos: uno, poner a **null** el elemento de la matriz que referencia al objeto que se desea eliminar (el objeto se envía a la basura para que sea recogido por el recolector de basura), y dos, quitar ese elemento de la matriz decrementando su tamaño en 1.

Notar que la operación de incrementar o decrementar en un elemento la matriz de referencias *clientes*, asigna un nuevo espacio de memoria; el necesario para el nuevo número de elementos. Entonces ¿qué sucede con el espacio que antes estaba referenciado por *clientes*? Pues que queda sin referenciar, condición suficiente para que sea enviado a la basura y recogido por el recolector de basura.

Para finalizar, queda escribir una aplicación que utilizando la clase *CBanco*, construya la entidad bancaria objetivo del ejemplo propuesto. Esta aplicación presentará un menú como el indicado a continuación:

1. Saldo
2. Buscar siguiente
3. Ingreso
4. Reintegro
5. Añadir
6. Eliminar
7. Mantenimiento
8. Salir

Opción:

La operación elegida será identificada por una sentencia **switch** y procesada de acuerdo al esquema presentado a continuación:

```
public class Test
{
    public static CCuenta leerDatos(int op) { ... }

    public static int menú() { ... }

    public static void main(String[] args)
    {
        // Crear un objeto banco vacío (con cero elementos)
        CBanco banco = new CBanco();

        do
        {
            opción = menú();
```

```
switch (opción)
{
    case 1: // saldo
        // Buscar un elemento por el nombre o por la cuenta.
        // La subcadena de búsqueda será obtenida del teclado.
        pos = banco.buscar(cadenabuscar, 0);
        // Si se encuentra, mostrar nombre, cuenta y saldo
        break;
    case 2: // buscar siguiente
        // Buscar el siguiente elemento que contenga la subcadena
        // utilizada en la última búsqueda (case 1).
        pos = banco.buscar(cadenabuscar, pos + 1);
        // Si se encuentra, mostrar nombre, cuenta y saldo
        break;
    case 3: // ingreso
    case 4: // reintegro
        // Ingresar o reintegrar una cantidad en la cuenta
        // especificada. Ambos datos se solicitarán del teclado.
        pos = banco.buscar(cuenta, 0);
        if (opción == 3)
            banco.clienteEn(pos).ingreso(cantidad);
        else
            banco.clienteEn(pos).reintegro(cantidad);
        break;
    case 5: // añadir
        // Añadir un nuevo cliente. El objeto correspondiente
        // será devuelto por el método static leerDatos de esta
        // aplicación, que obtendrá los datos desde el teclado.
        banco.añadir(leerDatos(tipo_objeto));
        break;
    case 6: // eliminar
        // Eliminar el cliente que coincide con la cuenta
        // tecleada.
        banco.eliminar(cuenta);
        break;
    case 7: // mantenimiento
        // Cobrar comisiones e ingresar intereses
        for (pos = 0; pos < banco.longitud(); pos++)
        {
            banco.clienteEn(pos).comisiones();
            banco.clienteEn(pos).intereses();
        }
        break;
    case 8: // salir
        banco = null;
    }
}
while(opción != 8);
```

El listado completo de la aplicación *Test* se muestra a continuación. Se puede observar que la clase aplicación utiliza tres métodos estáticos: *leerDatos*, *menú* y el método **main**.

El método *leerDatos* recibe como parámetro un valor 1, 2 ó 3 dependiendo del tipo de objeto que se desee crear: *CCuentaAhorro*, *CCuentaCorriente*, o *CCuentaCorrienteConIn*. Lee los atributos correspondientes al tipo de cuenta elegido e invoca al constructor adecuado. El método devuelve una referencia al nuevo objeto construido. Este método será invocado cada vez que se elija la opción *añadir* una nueva cuenta.

El método *menú* visualiza el menú anteriormente mostrado, y devuelve el entero correspondiente a la opción elegida.

El método **main** crea el objeto *banco* e invoca repetidamente al método *menú* para permitir elegir la operación programada que se desee realizar en ese instante sobre el cliente correspondiente a la cuenta o nombre especificado.

```
import java.io.*;
/////////////////////////////////////////////////////////////////
// Aplicación para trabajar con la clase CBanco y la jerarquía
// de clases derivadas de CCuenta
//
public class Test
{
    // Para la entrada de datos se utiliza Leer.class
    public static CCuenta leerDatos(int op)
    {
        CCuenta obj = null;
        String nombre, cuenta;
        double saldo, tipoi, mant;
        System.out.print("Nombre.....: ");
        nombre = Leer.dato();
        System.out.print("Cuenta.....: ");
        cuenta = Leer.dato();
        System.out.print("Saldo.....: ");
        saldo = Leer.datoDouble();
        System.out.print("Tipo de interés.....: ");
        tipoi = Leer.datoDouble();
        if (op == 1)
        {
            System.out.print("Mantenimiento.....: ");
            mant = Leer.datoDouble();
            obj = new CCuentaAhorro(nombre, cuenta, saldo, tipoi, mant);
        }
        else
        {
            int transex;
```

```
    double imptrans;
    System.out.print("Importe por transacción: ");
    imptrans = Leer.datoDouble();
    System.out.print("Transacciones exentas..: ");
    transex = Leer.datoInt();

    if (op == 2)
        obj = new CCuentaCorriente(nombre, cuenta, saldo, tipoi,
                                      imptrans, transex);
    else
        obj = new CCuentaCorrienteConIn(nombre, cuenta, saldo,
                                         tipoi, imptrans, transex);
    }
    return obj;
}

public static int menú()
{
    System.out.print("\n\n");
    System.out.println("1. Saldo");
    System.out.println("2. Buscar siguiente");
    System.out.println("3. Ingreso");
    System.out.println("4. Reintegro");
    System.out.println("5. Añadir");
    System.out.println("6. Eliminar");
    System.out.println("7. Mantenimiento");
    System.out.println("8. Salir");
    System.out.println();
    System.out.print("    Opción: ");
    int op;
    do
        op = Leer.datoInt();
    while (op < 1 || op > 8);
    return op;
}

public static void main(String[] args)
{
    // Definir una referencia al flujo estándar de salida: flujoS
    PrintStream flujoS = System.out;

    // Crear un objeto banco vacío (con cero elementos)
    CBanco banco = new CBanco();

    int opción = 0, pos = -1;
    String cadenabuscar = null;
    String nombre, cuenta;
    double cantidad;
    boolean eliminado = false;
```

```
do
{
    opción = menú();
    switch (opción)
    {
        case 1: // saldo
            flujoS.print("Nombre o cuenta, total o parcial ");
            cadenaBuscar = Leer.dato();
            pos = banco.buscar(cadenaBuscar, 0);
            if (pos == -1)
                if (banco.longitud() != 0)
                    flujoS.println("búsqueda fallida");
                else
                    flujoS.println("no hay clientes");
            else
            {
                flujoS.println(banco.clienteEn(pos).obtenerNombre());
                flujoS.println(banco.clienteEn(pos).obtenerCuenta());
                flujoS.println(banco.clienteEn(pos).estado());
            }
            break;
        case 2: // buscar siguiente
            pos = banco.buscar(cadenaBuscar, pos + 1);
            if (pos == -1)
                if (banco.longitud() != 0)
                    flujoS.println("búsqueda fallida");
                else
                    flujoS.println("no hay clientes");
            else
            {
                flujoS.println(banco.clienteEn(pos).obtenerNombre());
                flujoS.println(banco.clienteEn(pos).obtenerCuenta());
                flujoS.println(banco.clienteEn(pos).estado());
            }
            break;
        case 3: // ingreso
        case 4: // reintegro
            flujoS.print("Cuenta: "); cuenta = Leer.dato();
            pos = banco.buscar(cuenta, 0);
            if (pos == -1)
                if (banco.longitud() != 0)
                    flujoS.println("búsqueda fallida");
                else
                    flujoS.println("no hay clientes");
            else
            {
                flujoS.print("Cantidad: "); cantidad = Leer.datoDouble();
                if (opción == 3)
                    banco.clienteEn(pos).ingreso(cantidad);
                else

```

```

        banco.clienteEn(pos).reintegro(cantidad);
    }
    break;
case 5: // añadir
    flujoS.print("Tipo de cuenta: 1-(CA), 2-(CC), 3-CCI ");
    do
        opción = Leer.datoInt();
    while (opción < 1 || opción > 3);
    banco.añadir(leerDatos(opción));
    break;
case 6: // eliminar
    flujoS.print("Cuenta: "); cuenta = Leer.dato();
    eliminado = banco.eliminar(cuenta);
    if (eliminado)
        flujoS.println("registro eliminado");
    else
        if (banco.longitud() != 0)
            flujoS.println("cuenta no encontrada");
        else
            flujoS.println("no hay clientes");
    break;
case 7: // mantenimiento
    for (pos = 0; pos < banco.longitud(); pos++)
    {
        banco.clienteEn(pos).comisiones();
        banco.clienteEn(pos).intereses();
    }
    break;
case 8: // salir
    banco = null;
}
}
while(opción != 8);
}
/////////////////////////////////////////////////////////////////

```

Se puede observar que el mantenimiento de las cuentas de los clientes (case 7) resulta sencillo gracias a la aplicación de la definición de polimorfismo. Esto es, el método *comisiones* o *intereses* que se invoca para cada cliente depende del tipo del objeto referenciado por el elemento accedido de la matriz *clientes* de *banco*.

MÉTODOS EN LÍNEA

Cuando el compilador Java conoce con exactitud qué método tiene que llamar para responder al mensaje que se ha programado que un objeto reciba en un instante determinado, puede tomar la iniciativa de reemplazar la llamada al método por el cuerpo del mismo. Se dice entonces que el método está en línea. El que se pro-

duzca esta circunstancia, por ejemplo, porque el método es corto, redundará en tiempos de ejecución más bajos ya que se evita que el intérprete Java tenga que llamar al método. En principio, en Java, todos los métodos de una clase pueden ser métodos en línea.

¿Cuándo un método no podrá pasar a ser un método en línea? Cuando el compilador no sepa con exactitud a qué versión del método tiene que invocar. Veamos; si como en el ejemplo anterior, tenemos una matriz de referencias a objetos de las subclases *CCuentaAhorro*, *CCuentaCorriente* o *CCuentaCorrienteConIn*, ¿cómo sabe el compilador a qué método *interés*, por ejemplo, tiene que llamar? El compilador no puede saber esto. Cuando esto sucede, el compilador produce código que permitirá al intérprete Java consultar durante la ejecución qué método tiene que invocar. Como el intérprete Java sí sabe a qué objeto, *CCuentaAhorro*, *CCuentaCorriente* o *CCuentaCorrienteConIn*, se refiere cada uno de los elementos de la matriz, el código añadido por el compilador será suficiente para determinar qué método invocar para cada uno de los objetos.

La consulta dinámica acerca del método que hay que invocar es rápida, pero no tan rápida como invocar a un método directamente. Afortunadamente no hay muchos casos en los que Java necesite usar la consulta dinámica. Por ejemplo, los métodos finales (**final**), los estáticos (**static**) y los privados (**private**) pueden ser invocados directamente; y si son cortos son candidatos a ser métodos en línea. Si un método es final, el compilador sabe que ese método no puede ser redefinido, por lo tanto existe una sola versión; si es estático es invocado anteponiendo el nombre de su clase; y si es privado no puede ser invocado por un método que no sea de su clase. Por lo tanto, en ninguno de los tres casos habrá que tomar una decisión acerca de a qué método hay que llamar.

INTERFACES

De forma genérica una interfaz se define así: un dispositivo o un sistema utilizado por entidades inconexas para interactuar. Según esta definición un control remoto es una interfaz, el idioma inglés es una interfaz, etc. Análogamente, una interfaz Java es un dispositivo que permite interactuar a objetos no relacionados entre sí. Las interfaces Java en realidad definen un conjunto de mensajes que se puede aplicar a muchas clases de objetos, a los que cada una de ellas debe responder de forma adecuada. Por eso, una interfaz recibe también el nombre de protocolo.

Definir una interfaz

Una interfaz consta de dos partes: el *nombre de la interfaz* precedido por la palabra reservada **interface**, y el *cuerpo de la interfaz* encerrado entre llaves. Esto es:

```
[public] interface nombre_interfaz extends superinterfaces
{
    cuerpo de la interfaz
}
```

El modificador de acceso **public** indica que la interfaz puede ser utilizada por cualquier clase de cualquier paquete. Si no se especifica, entonces sólo estará accesible para las clases definidas en el mismo paquete que la interfaz. Una interfaz puede incluirse en un paquete exactamente igual que una clase.

El cuerpo de la interfaz puede incluir declaraciones de constantes y declaraciones de métodos (no sus definiciones).

La palabra clave **extends** significa que se está definiendo una interfaz que es una extensión de otras; también se puede decir que es una interfaz derivada de otras; estas otras se especifican a continuación de **extends** separadas por comas. Como habrá observado, a diferencia de las clases, una interfaz puede derivarse de más de una superinterfaz. Una interfaz así definida hereda todas las constantes y métodos de sus superinterfaces, excepto las constantes y métodos que queden ocultos porque se redefinan.

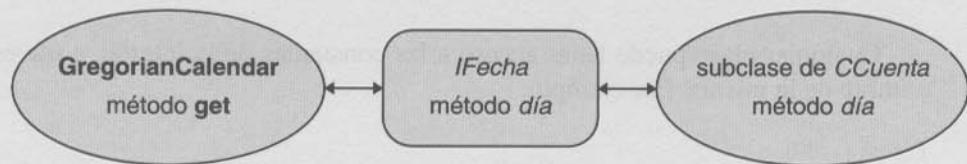
El nombre de una interfaz se puede utilizar en cualquier lugar donde se pueda utilizar el nombre de una clase.

Un ejemplo: la interfaz IFecha

En la jerarquía de clases implementada anteriormente en este mismo capítulo, nosotros declaramos las clases *CCuentaAhorro*, *CCuentaCorriente* y *CCuentaCorrienteConIn* como parte de un conjunto de clases para administrar distintos tipos de cuentas bancarias. Todas estas clases tienen varios métodos en común; así que para facilitar, no sólo el diseño, sino el trabajo con matrices de objetos de dichas clases, nosotros implementamos una superclase genérica, *CCuenta*, que encapsula los atributos y los métodos comunes a todas esas clases; incluso, alguno de esos métodos, como *comisiones* e *intereses*, no tenía sentido definirlos en la superclase porque debían ser después particularizados para cada una de las subclases. Esto nos condujo a definir esos métodos como abstractos, lo que implicó definir la superclase también abstracta.

Las interfaces, al igual que las clases y métodos abstractos, proporcionan plantillas de comportamiento que se espera sean implementadas por otras clases. Esto es, una interfaz Java declara un conjunto de métodos, pero no los define (sólo aporta los prototipos de los métodos). También puede incluir definiciones de constantes.

Para comprender con claridad las interfaces vamos a realizar un ejemplo de una interfaz *IFecha* que va a ser utilizada para que dos clases (**GregorianCalendar** y una de las subclases de *CCuenta*) interactúen entre sí.



La clase **GregorianCalendar** es un proveedor de servicios; en nuestro ejemplo, notificará el día a los objetos derivados de *CCuenta* cuando intenten ejecutar sus métodos *comisiones* o *intereses*. Para ello, como se muestra a continuación, proporciona el método *get* que devuelve el tipo de dato (día, mes, etc.) solicitado:

```

public class GregorianCalendar extends Calendar
{
    // ...
    public final int get(int tipo_de_dato) { ... }
    // ...
}
    
```

Según hemos planteando el problema, cualquier objeto derivado de *CCuenta* que quiera utilizar un objeto **GregorianCalendar** debe implementar el método *día* proporcionado por la interfaz *IFecha*. Este método es el medio utilizado por el objeto **GregorianCalendar** para notificar al objeto derivado de *CCuenta* el día actual. Según lo expuesto la interfaz *IFecha* puede tener el aspecto siguiente:

```

import java.util.*;
// Interfaz IFecha: métodos y constantes para obtener
// el día, mes y año
public interface IFecha
{
    public final static int DIA_DEL_MES = Calendar.DAY_OF_MONTH;
    public final static int MES_DEL_AÑO = Calendar.MONTH;
    public final static int AÑO = Calendar.YEAR;

    public abstract int día();
    public abstract int mes();
    public abstract int año();
}
    
```

Se puede observar que una interfaz sólo declara los métodos, no los define, y además puede definir constantes. También, cada una de las declaraciones y defi-

niciones finaliza con un punto y coma (;). Todos los métodos declarados en una interfaz son implícitamente públicos y abstractos (**public** y **abstract**); y todas las constantes son implícitamente públicas, finales y estáticas (**public**, **final** y **static**). En ambos casos, el uso de estos modificadores es sólo una cuestión de estilo.

Cualquier clase puede tener acceso a las constantes de la interfaz a través del nombre de la misma. Por ejemplo:

```
IFecha.AÑO
```

En cambio, una clase que implemente la interfaz puede tratar las constantes como si las hubiese heredado; esto es, accediendo directamente a su nombre.

Utilizar una interfaz

Para utilizar una interfaz hay que añadir el nombre de la misma precedido por la palabra clave **implements** a la definición de la clase. La palabra clave **implements** sigue a la palabra clave **extends**, si existe.

Siguiendo con el ejemplo iniciado en el apartado anterior, una subclase de *CCuenta* como *CCuentaAhorro* que utilice la interfaz *IFecha* debe definirse así:

```
import java.util.*;
// Clase CCuentaAhorro: clase derivada de CCuenta
//
public class CCuentaAhorro extends CCuenta implements IFecha
{
    // ...
    public void comisiones()
    {
        // Se aplican mensualmente por el mantenimiento de la cuenta
        if (día() == 1) reintegro(cuotaMantenimiento);
    }

    public double intereses()
    {
        if (día() != 1) return 0.0;
        // Acumular los intereses por mes sólo los días 1 de cada mes
        double interesesProducidos = 0.0;
        interesesProducidos = estado() * obtenerTipoDeInterés() / 1200.0;
        ingreso(interesesProducidos);
        // Devolver el interés mensual por si fuera necesario
        return interesesProducidos;
    }
}
```

```

// Implementación de los métodos de la interfaz IFecha
public int día()
{
    GregorianCalendar fechaActual = new GregorianCalendar();
    return fechaActual.get(DIA_DEL_MES);
}
public int mes() { return 0; } // no se necesita
public int año() { return 0; } // no se necesita
}
/////////////////////////////////////////////////////////////////

```

Como una interfaz sólo aporta declaraciones de métodos abstractos, es nuestra obligación definir todos los métodos en cada una de las clases que utilice la interfaz. No podemos elegir y definir sólo aquellos métodos que necesitemos. De no hacerlo, Java obligaría a que la clase fuera abstracta. Se puede observar también cómo el acceso a las constantes definidas en la interfaz es directo.

Si una clase implementa una interfaz, todas sus subclases heredarán los nuevos métodos que se hayan implementado en la superclase, así como las constantes definidas por la interfaz. Por ejemplo, modifiquemos la clase *CCuentaCorriente* para que utilice también la interfaz *IFecha*:

```

*import java.util.*;
/////////////////////////////////////////////////////////////////
// Clase CCuentaCorriente: clase derivada de CCuenta
//
public class CCuentaCorriente extends CCuenta implements IFecha
{
    // ...
    public void comisiones()
    {
        // Se aplican mensualmente por el mantenimiento de la cuenta
        if (día() == 1)
        {
            // ...
        }
    }

    public double intereses()
    {
        if (día() != 1) return 0.0;
        // ...
    }

    // Implementación de los métodos de la interfaz IFecha
    public int día()
    {
        GregorianCalendar fechaActual = new GregorianCalendar();
    }
}

```

```

        return fechaActual.get(DIA_DEL_MES);
    }
    public int mes() { return 0; } // no se necesita
    public int año() { return 0; } // no se necesita
}
/////////////////////////////////////////////////////////////////

```

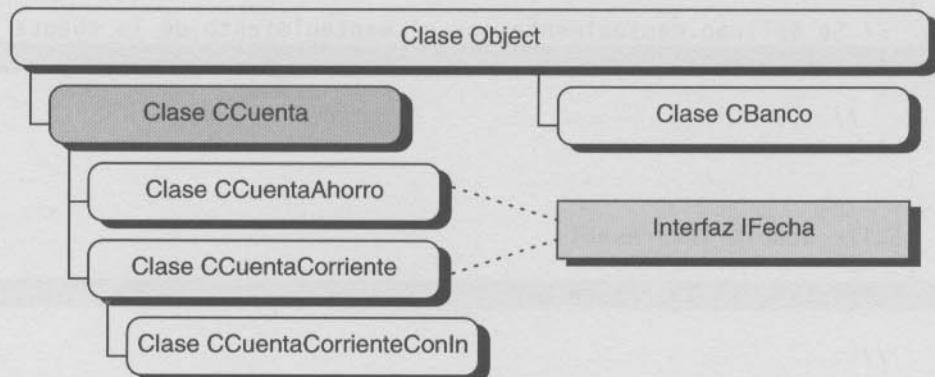
Como la clase *CCuentaCorriente* implementa la interfaz *IFecha*, su subclase *CCuentaCorrienteConIn* heredará los nuevos métodos y constantes. Por lo tanto, no es necesario agregar a la definición de esta clase la palabra clave **implements** más el nombre de la interfaz.

```

/////////////////////////////////////////////////////////////////
// Clase CCuentaCorrienteConIn: clase derivada de CCuentaCorriente
//
public class CCuentaCorrienteConIn extends CCuentaCorriente
{
    // ...
    public double intereses()
    {
        if *(día() != 1 || estado() < 3000) return 0.0;
        // ...
    }
}
/////////////////////////////////////////////////////////////////

```

Una vez realizadas en nuestra jerarquía de clases las modificaciones propuestas como consecuencia de haber añadido la interfaz *IFecha*, el resultado obtenido desde un punto de vista gráfico es el siguiente:



Probablemente habrá pensado que hubiéramos obtenido el mismo resultado implementando la interfaz *IFecha* en la superclase *CCuenta*. Pues, si es así, tiene

razón. El hecho de haber implementado la interfaz en las subclases ha sido puramente didáctico.

Clase abstracta frente a interfaz

Llegado a este punto, se preguntará ¿en qué difiere una interfaz de una clase abstracta? Puesto que una interfaz es simplemente una lista de constantes y métodos abstractos, ¿sería equivalente la clase *IFecha* siguiente, a la interfaz *IFecha*?

```
import java.util.*;
// Clase IFecha: métodos y constantes para obtener
// el día, mes y año
public abstract class IFecha
{
    public final static int DIA_DEL_MES = Calendar.DAY_OF_MONTH;
    public final static int MES_DEL_AÑO = Calendar.MONTH;
    public final static int AÑO = Calendar.YEAR;

    public abstract int día();
    public abstract int mes();
    public abstract int año();
}
```

La respuesta a la pregunta anterior es no. Si *IFecha* es una clase abstracta, entonces todas las subclases de *CCuenta*, como *CCuentaAhorro*, que quisieran utilizar su funcionalidad para interactuar con **GregorianCalendar** tendrían que derivarse de ella. Pero sucede que las subclases a las que nos referimos ya tienen una superclase y no pueden tener otra, ya que Java no permite la herencia múltiple de clases; sí permite que una interfaz se derive de múltiples interfaces. Por lo tanto, en casos como el presentado hay que utilizar una interfaz.

Lo anterior es una explicación práctica. Una explicación conceptual puede ser que **GregorianCalendar** no debe forzar a sus usuarios a establecer una relación entre clases. Esto es, no importa la clase; lo único que importa es implementar uno o más métodos específicos. Al fin y al cabo, una interfaz no es más que un protocolo que una clase implementa cuando necesita utilizarlo.

Evidentemente nuestro problema en concreto tiene una solución, que es derivar la clase *CCuenta* de la clase abstracta *IFecha* e implementar en *CCuenta* los métodos proporcionados por *IFecha*. Pero nuestro objetivo no es dar solución a este problema, sino presentar ejemplos adecuados acerca de lo que se quiere explicar.

Utilizar una interfaz como un tipo

Una interfaz es un nuevo tipo de datos; un tipo referenciado. Por lo tanto, el nombre de una interfaz se puede utilizar en cualquier lugar donde pueda aparecer el nombre de cualquier otro tipo de datos.

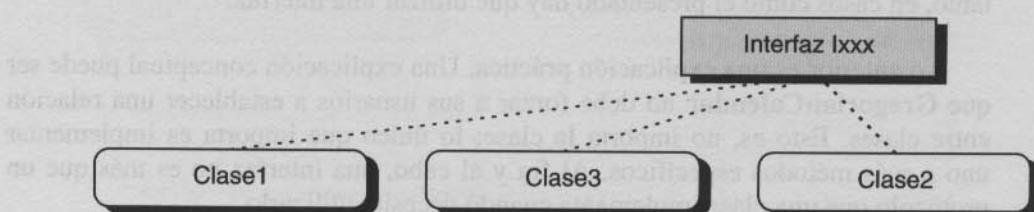
Por ejemplo, se puede declarar una matriz *clientes* que sea de tipo *IFecha* y asignar a cada elemento un objeto de algunas de las subclases de *CCuenta*:

```
IFecha[] clientes = new IFecha[3];
CCuentaAhorro cliente0 = new CCuentaAhorro();
clientes[0] = cliente0;
// ...
((CCuentaAhorro)clientes[0]).asignarNombre("cliente0");
System.out.println(cliente0.obtenerNombre());
// ...
```

Una variable del tipo de una interfaz espera referenciar un objeto que tenga implementada dicha interfaz, de lo contrario el compilador Java mostrará un error. En el ejemplo anterior la variable *clientes[0]* de tipo *IFecha* hace referencia a un objeto *CCuentaAhorro* que implementa esa interfaz.

Asimismo, se puede observar que es posible convertir implícitamente referencias a objetos que implementan una interfaz en referencias a esa interfaz y vice-versa, pero en este caso, explícitamente. Lógicamente, con lo que sabemos, podemos deducir que con una referencia a la interfaz sólo se tiene acceso a los métodos y constantes declarados en dicha interfaz.

El siguiente ejemplo muestra cómo tres clases no relacionadas, *Clase1*, *Clase2* y *Clase3*, por el hecho de implementar la misma interfaz *Ixxx*, permite definir una matriz de objetos de esas clases y aplicar la definición de polimorfismo.



```
public interface Ixxx
{
    public abstract void m(); // método m
    public abstract void p(); // método p
}
```

```

public class Clase1 implements Ixxx
{
    public void m()
    {
        System.out.println("método m de Clase1");
    }
    public void p() {}
}

public class Clase2 implements Ixxx
{
    public void m()
    {
        System.out.println("método m de Clase2");
    }
    public void p() {}
}

public class Clase3 implements Ixxx
{
    public void m()
    {
        System.out.println("método m de Clase3");
    }
    public void p() {}
}

public class Test
{
    public static void main (String[] args)
    {
        Ixxx[] objs = new Ixxx[3]; // matriz de referencias a objetos
        objs[0] = new Clase1();
        objs[1] = new Clase2();
        objs[2] = new Clase3();
        for( int i = 0; i < objs.length; i++)
            objs[i].m(); // invoca al método m del objeto Clase1, Clase2
                           // o Clase3 referenciado por objs[i]
    }
}

```

Interfaces frente a herencia múltiple

A menudo se piensa en las interfaces como en una alternativa a la herencia múltiple. Pero la realidad es que ambos conceptos, interfaz y herencia múltiple, son bastantes diferentes, a pesar de que las interfaces pueden resolver problemas similares. En particular:

- Desde una interfaz, una clase sólo hereda constantes.
- Desde una interfaz, una clase no puede heredar definiciones de métodos.
- La jerarquía de interfaces es independiente de la jerarquía de clases. De hecho, varias clases pueden implementar la misma interfaz y no pertenecer a la misma jerarquía de clases. En cambio, cuando se habla de herencia múltiple, todas las clases pertenecen a la misma jerarquía.

Para qué sirve una interfaz

Después de todo lo expuesto es posible que aún no esté claro cuál es el sentido de utilizar interfaces. Si analizamos el ejercicio realizado anteriormente basado en la jerarquía de clases *CCuenta* y en la interfaz *IFecha*, seguro que llegaremos a alguna conclusión similar a la siguiente: puesto que los métodos *día*, *mes* y *año* pertenecen a la subclase que los implementa y las constantes no son ningún obstáculo, ¿para qué queremos la interfaz? Pensando así, para nada.

Una interfaz se utiliza para definir un protocolo de conducta que puede ser implementado por cualquier clase en una jerarquía de clases. La utilidad que esto pueda tener puede resumirse en los puntos siguientes:

- Captar similitudes entre clases no relacionadas sin forzar entre ellas una relación artificial. Una acción de este tipo permitiría incluso, definir una matriz de objetos de esas clases y aplicar, si fuera necesario, la definición de polimorfismo.
- Declarar métodos que una o más clases deben implementar en determinadas situaciones.

Suponga que se ha diseñado una clase de objetos que puede tener un comportamiento especial siempre que implemente unos determinados métodos. Por ejemplo, cuando aprenda sobre *applets* y subprocessos comprobará que usar un subprocesso en un *applet* implica que la clase de éste implemente la interfaz *Runnable*.

- Publicar la interfaz de programación de una clase sin descubrir cómo está implementada.

En este caso, otros desarrolladores recibirían la clase compilada y la interfaz correspondiente.

Implementar múltiples interfaces

Una clase puede implementar una o más interfaces. Por ejemplo:

```
public class miClase implements interfaz1, interfaz2, interfaz3
{
    // ...
}
```

Cuando una clase implemente múltiples interfaces puede suceder que dos o más interfaces diferentes implementen el mismo método. Si esto ocurre, proceda de alguna de las formas indicadas a continuación:

- Si los métodos tienen el mismo prototipo, basta con definir uno en la clase.
- Si los métodos difieren en el número o tipo de sus parámetros, estamos en el caso de una sobrecarga del método; implemente todas las sobrecargas.
- Si los métodos sólo difieren en el tipo del valor returned, no existe sobre carga y el compilador produce un error, ya que dos métodos pertenecientes a la misma clase no pueden diferir sólo en el tipo del resultado.

CLASES ANIDADAS

Una *clase anidada* es una clase que es un miembro de otra clase. Por ejemplo, en el código mostrado a continuación, *CFecha* es una clase anidada:

```
public class CPersona
{
    // Miembros de CPersona
    private class CFecha
    {
        // Miembros de CFecha
    }
    // Otros miembros de CPersona
}
```

Una clase se debe definir dentro de otra sólo cuando tenga sentido en el contexto de la clase que la incluye o cuando depende de la función que desempeña la clase que la incluye. Por ejemplo una ventana puede definir su propio cursor; en este caso, la ventana puede ser un objeto de una clase y el cursor de una clase anidada.

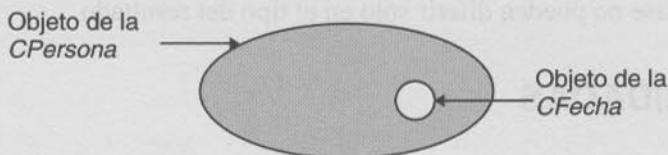
Una clase anidada es un miembro más de la clase que la contiene. En el ejemplo anterior la clase *CFecha* es un miembro más de *CPersona* y como tal se le aplican las mismas reglas que para el resto de los miembros. Según esto, *CFecha* tendrá acceso al resto de los miembros de *CPersona* independientemente de su modificador de acceso (decir *CFecha*, implica a los miembros de *CFecha*); *CFecha* puede ser pública, privada o protegida; puede ser estática; etc.

Recuerde: un miembro estático (**static**) es un miembro de la clase y uno no estático es un miembro del objeto; y como ocurría con los métodos estáticos, una clase anidada estática no puede referirse directamente a un miembro del objeto, sólo puede hacerlo a través de un objeto de su clase.

Clases internas

Cuando un miembro de una clase es una clase anidada y no es **static** recibe el nombre de *clase interna* por tratarse de un miembro del objeto. Esto significa que cuando se cree un objeto de la clase externa también se creará uno de la interna. Por ejemplo, según la definición anterior de *CPersona* y *CFecha*, el siguiente código crea un objeto *obj* de la *CPersona* que incluye un objeto de la *CFecha*.

```
CPersona obj = new CPersona();
```



Si una *clase interna* está asociada con un objeto, lógicamente no puede tener miembros **static**.

Un objeto de una clase interna puede existir sólo dentro de un objeto de su clase externa. Asimismo, puesto que se trata de un miembro de su clase externa, tiene acceso directo al resto de los miembros de esa clase. Por ejemplo, una fecha puede ser un miembro de los datos relativos a la identificación de una persona; entonces, la persona puede ser representada por una clase *CPersona* y la fecha por una clase *CFecha*, como puede observar en el código mostrado a continuación:

```
public class CPersona
{
    private String nombre;
    private CFecha fechaNacimiento;

    private class CFecha
    {
        private int dia, mes, año;
        private CFecha(int dd, int mm, int aa)
        {
            dia = dd; mes = mm; año = aa;
        }
    }

    public CPersona() {}
```

```

public CPersona(String nom, int dd, int mm, int aa)
{
    nombre = nom;
    fechaNacimiento = new CFecha(dd, mm, aa);
}

public String obtenerNombre() { return nombre; }

public String obtenerFechaNa()
{
    return fechaNacimiento.día + "/" +
        fechaNacimiento.mes + "/" +
        fechaNacimiento.año;
}
}

```

El siguiente ejemplo crea un objeto *CPersona* invocando al constructor de esta clase, el cual, a su vez, invocará al constructor de *CFecha* para crear el objeto *fechaNacimiento* con la fecha pasada como argumento.

```

public class Test
{
    public static void main(String[] args)
    {
        CPersona unaPersona = new CPersona("Su nombre", 22, 2, 2002);
        System.out.println(unaPersona.obtenerNombre());
        System.out.println(unaPersona.obtenerFechaNa());
    }
}

```

Cuando compile la aplicación *Test* anterior, podrá observar que Java genera, además del fichero *Test.class* y de *CPersona.class*, el fichero *CPersona\$CFecha.class* correspondiente a la clase interna. Por lo tanto, cuando quiera instalar la aplicación en otra máquina no olvide que cada clase interna tiene su propio fichero de clase, y que deben incluirse junto con los de las clases de nivel superior. Resumiendo, para poder ejecutar la aplicación *Test* del ejemplo anterior son necesarios los ficheros: *Test.class*, *CPersona.class* y *CPersona\$CFecha.class*.

Clases definidas dentro de un método

Java permite definir una clase dentro de un método. Por ejemplo, el método *meto-Clase2* de la *Clase2* mostrada a continuación incluye la definición de una *Clase3*:

```

public class Clase1
{
    public static void main (String[] args)
    {

```

```

        Clase2 obj = new Clase2();
        obj.metClase2(1, 2);
    }
}

public class Clase2
{
    public void metClase2(int x, final int y)
    {
        int i = x + y;
        final int c = x + y;
        class Clase3
        {
            // int a = x; // error: x no es final
            int b = y;
            void metClase3()
            {
                // System.out.println(b + i); // error: i no es final
                System.out.println(b + c);
            }
        }
        Clase3 obj = new Clase3();
        obj.metClase3();
    }
}

```

Una clase definida dentro de un método tiene unas reglas de acceso bastante restrictivas. Un método de una clase definida dentro de otro método sólo tiene acceso a sus variables locales o parámetros formales declarados **final**. En el ejemplo, *metClase2* tiene dos parámetros formales, *x* e *y*, y dos variables locales, *i* y *c*. Se puede observar que el método *metClase3* de la clase *Clase3* definida dentro de *metClase2* sólo tiene acceso a las variables locales y parámetros de este que han sido declarados **final**.

Clases anónimas

Algunas veces podemos escribir una clase dentro de un método sin necesidad de identificarla. Una clase definida de esta forma recibe el nombre de *clase anónima*.

Por otra parte, para crear con **new** un objeto de una clase necesitamos conocer el nombre de la misma. Entonces ¿para qué se utiliza una clase anónima?

```

interface Interfaz
{
    public abstract void p();
    public abstract void m();
}

```

```

class Clase2
{
    private int i;

    // ...
    public Interfaz metClase2() // método de la clase 2
    {
        return new Clase3();
    }

    class Clase3 implements Interfaz
    {
        public void p() { System.out.println("método p"); }
        public void m() { System.out.println("método m"); }
    }
}

public class Clase1
{
    public static void main (String[] args)
    {
        Clase2 obj = new Clase2();
        Interfaz iobj = obj.metClase2(); // devuelve un objeto Clase3
        iobj.m();
    }
}

```

En este ejemplo se puede observar que la *Clase2* define una clase interna *Clase3* y un método *metClase2* que devuelve una referencia del tipo *Interfaz* a un objeto *Clase3* que implementa dicha interfaz. El método **main** de *Clase1* pone a prueba las capacidades de *Clase2*.

Una alternativa al ejemplo planteado es definir anónima la clase utilizada dentro del método (en el ejemplo *Clase3*). Por ser anónima, su definición debería ocupar el lugar donde se utiliza su nombre para crear un objeto de la misma. Esto es, la definición y la creación del objeto se harían bajo la siguiente sintaxis:

new Ixxx () { ... }

donde *Ixxx* es el nombre de la interfaz que implementa la clase. Por ejemplo:

```

public Interfaz metClase2()
{
    return new Interfaz()
    {
        public void p() { System.out.println("método p"); }
        public void m() { System.out.println("método m"); }
    };
}

```

Se puede observar que la llamada a **new** va seguida de la definición de la clase y que no se utiliza el nombre de la clase, sino el nombre de la interfaz que la clase implementa; notar que en este caso no interviene la palabra clave **implements** y que la sentencia **return** finaliza con punto y coma.

Resumiendo, una clase anónima es una forma de evitar el que tengamos que pensar en un nombre trivial para una clase pequeña en código, utilizada en un lugar concreto. Evidentemente, el precio que se paga es que no podremos crear objetos de esta clase fuera del lugar donde haya sido definida.

EJERCICIOS RESUELTOS

Se quiere escribir un programa para manipular ecuaciones algebraicas o polinómicas dependientes de las variables x e y . Por ejemplo:

$$2x^3y - xy^3 + 8.25 \text{ más } 5x^5y - 2x^3y + 7x^2 - 3 \text{ igual a } 5x^5y + 7x^2 - xy^3 + 5.25$$

Cada término del polinomio será representado por una clase *CTermino* y cada polinomio por una clase *CPolinomio*.

Quizás se pregunte qué sentido tiene realizar este ejercicio, si no trata con subclases. La respuesta es sencilla, este ejercicio es la antesala al ejercicio que a continuación se propone.

La clase *CTermino* puede escribirse así:

```
import java.math.*;
/////////////////////////////////////////////////////////////////
// Clase CTermino: expresión de la forma a.x^n.y^m
//                   a es el coeficiente de tipo double.
//                   n y m son los exponentes enteros de x e y.
//
public class CTermino {
    private double coeficiente = 0.0; // coeficiente
    private int exponenteDeX = 1;      // exponente de x
    private int exponenteDeY = 1;      // exponente de y

    public CTermino() {}

    public CTermino( double coef, int expx, int expy ) // constructor
    {
        coeficiente = coef;
        exponenteDeX = expx;
        exponenteDeY = expy;
    }
}
```

```

public CTermino(CTermino t) // constructor copia
{
    coeficiente = t.coeficiente;
    exponenteDeX = t.exponenteDeX;
    exponenteDeY = t.exponenteDeY;
}
public CTermino copiar(CTermino t) // asignación
{
    coeficiente = t.coeficiente;
    exponenteDeX = t.exponenteDeX;
    exponenteDeY = t.exponenteDeY;
    return this;
}
public void asignarCoeficiente(double coef) {coeficiente = coef;}
public double obtenerCoeficiente() {return coeficiente;}
public void asignarExponenteDeX(int expx) {exponenteDeX = expx;}
public int obtenerExponenteDeX() {return exponenteDeX;}
public void asignarExponenteDeY(int expy) {exponenteDeY = expy;}
public int obtenerExponenteDeY() {return exponenteDeY;}
public void mostrarTermino()
{
    if (coeficiente == 0) return;
    // Signo
    String sterm = (coeficiente < 0) ? " - " : " + ";
    // Coeficiente
    if (Math.abs(coeficiente) != 1)
        sterm = sterm + Math.abs(coeficiente);
    // Potencia de x
    if (exponenteDeX > 1 || exponenteDeX < 0)
        sterm = sterm + "x^" + exponenteDeX;
    else if (exponenteDeX == 1)
        sterm = sterm + "x";
    // Potencia de y
    if (exponenteDeY > 1 || exponenteDeY < 0)
        sterm = sterm + "y^" + exponenteDeY;
    else if (exponenteDeY == 1)
        sterm = sterm + "y";
    // Mostrar término
    System.out.print(sterm);
}
}

```

La clase *CTermino* representa un término del polinomio, el cual queda perfectamente definido cuando se conoce su coeficiente, el grado de la variable *x* y el grado de la variable *y*: *coeficiente*, *exponenteDeX* y *exponenteDeY*.

Para acceder a los atributos de un término se han implementado las funciones típicas de asignar y obtener el valor almacenado en el atributo que se trate en cada

caso. Otros métodos implementados son: un constructor sin argumentos y otro con argumentos para permitir construir un objeto *CTermino* a partir de unos valores determinados; un constructor copia para poder construir un nuevo término a partir de otro existente; un método *copiar* para poder copiar un término en otro existente; y un método *mostrarTermino* para visualizar un término en la pantalla.

Es evidente que extender esta clase a términos de polinomios dependientes de más de dos variables no entraña ninguna dificultad; es cuestión de añadir más datos miembro y las funciones de acceso correspondientes.

Siguiendo con el desarrollo, el esqueleto de la clase *CPolinomio* puede escribirse así:

```
import java.math.*;
///////////////////////////////
// Clase CPolinomio. Un objeto CPolinomio consta de uno o más
//           objetos CTermino.
//
public class CPolinomio
{

    private CTermino[] términos; // matriz de objetos
    private int nElementos; // número de elementos de la matriz

    public CPolinomio()
    {
        // Crear una matriz vacía
        nElementos = 0;
        términos = new CTermino[nElementos];
    }

    private void unElementoMás(CTermino[] términosAct) { ... }
    private void unElementoMenos(CTermino[] términosAct) { ... }
    public void insertarTermino(CTermino obj) { ... }
    public boolean eliminarTermino(int i) { ... }
    public CTermino términoEn(int i) { ... }
    public int longitud() { return nElementos; }
    public CPolinomio copiar(CPolinomio p) { ... }
    public CPolinomio sumar(CPolinomio pB) { ... }
    public void mostrarPolinomio() { ... }
    public double valorPolonomio(double x, double y) { ... }
}
///////////////////////////////
```

Como se puede observar, la clase *CPolinomio* tiene dos atributos: *términos* que es una matriz de referencias a objetos *CTermino* y *nElementos* que es un entero que especifica el número de términos del polinomio.

Para crear un polinomio escribiremos una sentencia análoga a la siguiente:

```
CPolinomio polinomioA = new CPolinomio();
```

Esta sentencia, invoca al constructor *CPolinomio* e inicia un polinomio con 0 elementos, según se puede observar en la clase *CPolinomio*.

Para incrementar la matriz de referencias del polinomio en un elemento, iniciado con el valor **null**, escribiremos el método *unElementoMás* y para eliminar un elemento previamente establecido a **null**, el método *unElementoMenos*. Ambos métodos se muestran a continuación:

```
private void unElementoMás(CTermino[] términosAct)
{
    nElementos = términosAct.length;
    // Crear una matriz con un elemento más
    términos = new CTermino[nElementos + 1];
    // Copiar los términos que hay actualmente
    for ( int i = 0; i < nElementos; i++ )
        términos[i] = términosAct[i];
    nElementos++;
}

private void unElementoMenos(CTermino[] términosAct)
{
    if (términosAct.length == 0) return;
    int k = 0;
    nElementos = términosAct.length;
    // Crear una matriz con un elementos menos
    términos = new CTermino[nElementos - 1];
    // Copiar los términos no nulos que hay actualmente
    for ( int i = 0; i < nElementos; i++ )
        if (términosAct[i] != null)
            términos[k++] = términosAct[i];
    nElementos--;
}
```

Para añadir un nuevo término en el polinomio escribiremos el método *insertarTermino*, que permite insertar el término pasado como argumento, en orden ascendente del exponente de x ; y a exponentes iguales de x , en orden ascendente de y . Este método primeramente verifica si el coeficiente del término a insertar es 0, en cuya caso finaliza sin realizar ninguna inserción. Si el coeficiente es distinto de 0, verifica si el término en xy a insertar ya existe, en cuyo caso simplemente suma al coeficiente existente el del término pasado como argumento; si el resultado de esta suma es cero, invoca además al método *eliminarTermino* para quitar ese término. Si el término no existe, entonces lo inserta en el lugar adecuado. Para realizar esta operación, primero llama al método *unElementoMás* (añade un elemento

vacio al final) y después busca el lugar donde debe ser insertado el nuevo término; si ese lugar no es el último, hace un hueco moviendo un lugar en esta dirección los términos que hay desde ahí hasta el final.

El algoritmo utilizado para saber el lugar que le corresponde al término que se inserta en orden ascendente primero por x y después por y , es muy sencillo: a cada unidad del exponente de x le damos un peso k y a cada unidad del exponente de y un peso de 1; la suma de ambas cantidades nos da el valor utilizado para efectuar la ordenación requerida. El valor de k es la potencia de 10 que sea igual o mayor que el mayor de los exponentes de x e y del término a insertar.

```
public void insertarTermino(CTermino obj)
{
    // Insertar un nuevo término en orden ascendente del
    // exponente de x; y a igual exponente de x, en orden
    // ascendente del exponente de y.
    if ( obj.obtenerCoeficiente() == 0 ) return;
    int k = 10, i;
    int expX = obj.obtenerExponenteDeX();
    int expY = obj.obtenerExponenteDeY();
    // Si el término en xy existe, sumar los coeficientes
    for ( i = nElementos - 1; i >= 0; i-- )
    {
        if ( expX == términos[i].obtenerExponenteDeX() &&
            expY == términos[i].obtenerExponenteDeY() )
        {
            double coef = términos[i].obtenerCoeficiente() +
                obj.obtenerCoeficiente();
            if ( coef != 0 )
                términos[i].asignarCoeficiente(coef);
            else
                eliminarTermino(i);
            return;
        }
    }
    // Si el término en xy no existe, insertarlo.
    while (Math.abs(expX) > k || Math.abs(expY) > k) k = k*10;
    // Se añade un elemento vacío
    unElementoMás(términos);
    i = nElementos - 2; // i = nElementos - 1 vale null
    while ( i >= 0 && (expX * k + expY <
                           términos[i].obtenerExponenteDeX() * k +
                           términos[i].obtenerExponenteDeY()) )
    {
        términos[i+1] = términos[i];
        i--;
    }
    términos[i+1] = obj;
}
```

Para eliminar un término, escribiremos el método *eliminarTermino* que recibe como argumento el índice del elemento de la matriz *términos* del objeto *CPolinomio* que hace referencia al mismo. Utilizando ese índice, se envía el objeto *CTermino* a la basura poniendo a **null** el elemento que lo referencia y se llama al método *unElementoMenos* para quitar dicho elemento de la matriz y decrementar en 1 el número de elementos de la misma. El método devuelve **true** si la operación se realiza satisfactoriamente y **false** en caso contrario.

```
public boolean eliminarTermino(int i)
{
    // Eliminar el objeto que está en la posición i
    if (i >= 0 && i < nElementos)
    {
        términos[i] = null; // enviar el objeto a la basura
        unElementoMenos(términos);
        return true;
    }
    return false;
}
```

Para obtener el término *i* del polinomio implementamos el método *términoEn*. Este método recibe como argumento el índice *i* del término del polinomio que se desea recuperar y devuelve una referencia al mismo.

```
public CTermino términoEn(int i)
{
    // Devolver la referencia al objeto i de la matriz
    if (i >= 0 && i < nElementos)
        return términos[i];
    else
    {
        System.out.println("Índice fuera de límites");
        return null;
    }
}
```

El método *longitud* devuelve el número de términos del polinomio.

Para poder copiar un polinomio en otro escribiremos el método *copiar* especificado a continuación:

```
public CPolinomio copiar(CPolinomio p) // asignación
{
    // Copiar el origen en el nuevo destino
    nElementos = p.nElementos;
    términos = new CTermino[nElementos];
    for (int i = 0; i < nElementos; i++)
        términos[i] = new CTermino(p.términos[i]);
```

```

    return this;
}
}

```

Observe que primero se construye un nueva matriz de referencias del mismo tamaño que la matriz origen y después se asigna a cada uno de sus elementos, el correspondiente duplicado del objeto *CTermino* (invocando al constructor copia). Si no duplicáramos los objetos *CTermino*, esto es, si hiciéramos:

```
términos[i] = p.términos[i]
```

los dos polinomios, origen y destino, harían referencia a los mismos términos; con lo cual, las modificaciones realizadas en uno de ellos repercutirían también de la misma forma en el otro.

El siguiente método permite sumar dos polinomios. La idea básica es construir un tercer polinomio que contenga los términos de los otros dos, pero sumando los coeficientes de los términos que se repitan en ambos. Los términos en el polinomio resultante también quedarán ordenados ascendenteamente, por el mismo criterio que se expuso anteriormente. Una vez finalizada la suma, se eliminarán los términos que hayan resultado nulos (coeficiente 0). Un ejemplo de cómo invocar a este método puede ser el siguiente:

```
pR = pA.sumar(pB);
```

El proceso de sumar consiste en:

- Partiendo de los polinomios *pA* y *pB* que se quieren sumar, obtener un término de cada uno de ellos.
- Comparar los dos términos (uno de cada polinomio) según el criterio explicado cuando se expuso el método *insertarTermino*, y almacenar el menor en el polinomio *pR*.
- Obtener el siguiente término del polinomio al que pertenecía el término almacenado en *pR*, y volver al punto b).
- Cuando no queden más elementos en uno de los dos polinomios de partida, se copian directamente en *pR* todos los elementos que queden en el otro polinomio.

```

public CPolinomio sumar(CPolinomio pB)
{
    // pR = pA.sumar(pB). pA es this y pR el resultado.
    int ipa = 0, ipb = 0, k = 0;
    int na = nElementos, nb = pB.nElementos;
}

```

```

double coefA, coefB;
int expXA, expYA, expXB, expYB;
CPolinomio pR = new CPolinomio(); // polinomio resultante
// Sumar pA con pB
while ( ipa < na && ipb < nb )
{
    coefA = términos[ipa].obtenerCoeficiente();
    expXA = términos[ipa].obtenerExponenteDeX();
    expYA = términos[ipa].obtenerExponenteDeY();
    coefB = pB.términos[ipb].obtenerCoeficiente();
    expXB = pB.términos[ipb].obtenerExponenteDeX();
    expYB = pB.términos[ipb].obtenerExponenteDeY();
    k = 10;
    while (Math.abs(expXA) > k || Math.abs(expYA) > k) k = k*10;

    if ( expXA == expXB && expYA == expYB )
    {
        pR.insertarTermino(new CTermino(coefA+coefB, expXA, expYA));
        ipa++; ipb++;
    }
    else if (expXA * k + expYA < expXB * k + expYB)
    {
        pR.insertarTermino(new CTermino(coefA, expXA, expYA));
        ipa++;
    }
    else
    {
        pR.insertarTermino(new CTermino(coefB, expXB, expYB));
        ipb++;
    }
}
// Términos restantes en el pA
while ( ipa < na )
{
    coefA = términos[ipa].obtenerCoeficiente();
    expXA = términos[ipa].obtenerExponenteDeX();
    expYA = términos[ipa].obtenerExponenteDeY();
    pR.insertarTermino(new CTermino(coefA, expXA, expYA));
    ipa++;
}
// Términos restantes en el pB
while ( ipb < nb )
{
    coefB = pB.términos[ipb].obtenerCoeficiente();
    expXB = pB.términos[ipb].obtenerExponenteDeX();
    expYB = pB.términos[ipb].obtenerExponenteDeY();
    pR.insertarTermino(new CTermino(coefB, expXB, expYB));
    ipb++;
}

```

```

// Quitar los términos con coeficiente 0
k = 0;
while ( k < pR.nElementos )
{
    if (pR.términos[k].obtenerCoeficiente() == 0)
    {
        pR.eliminarTermino(k);
        pR.nElementos--;
    }
    else
        k++;
}
return pR;
}

```

El siguiente método permite mostrar todos los términos del polinomio pasado como argumento.

```

public void mostrarPolinomio()
{
    int i = nElementos;

    while ( i-- != 0 )
        términos[i].mostrarTermino();
}

```

Este otro método que se expone a continuación, devuelve el valor del polinomio para los valores de x e y pasados como argumentos.

```

public double valorPolonomio(double x, double y)
{
    double v = 0;

    for ( int i = 0; i < nElementos; i++ )
        v += términos[i].obtenerCoeficiente() *
            Math.pow(x, términos[i].obtenerExponenteDeX()) *
            Math.pow(y, términos[i].obtenerExponenteDeY());
    return v;
}

```

El siguiente programa, utilizando las clases *CTermino* y *CPolinomio*, lee dos polinomios, crea un tercero suma de los dos anteriores y visualiza el polinomio resultante, así como su valor para x e y igual a 1.

```

// Suma de polinomios dependientes de dos variables.
// Esta aplicación utiliza la clase Leer.
//
public class Test
{

```

```

public static CTermino leerTermino()
{
    CTermino ptx = null;
    double coef;
    int expx, expy;
    System.out.print("Coeficiente:      ");
    coef = Leer.datoDouble();
    System.out.print("Exponente en X: ");
    expx = Leer.datoInt();
    System.out.print("Exponente en Y: ");
    expy = Leer.datoInt();
    System.out.println();
    if ( coef == 0 && expx == 0 && expy == 0 ) return null;
    ptx = new CTermino( coef, expx, expy );
    return ptx;
}

public static void main(String[] args)
{
    // Definir los polinomios a sumar
    CPolinomio polinomioA = new CPolinomio();
    CPolinomio polinomioB = new CPolinomio();
    // Declarar una referencia al polinomio resultante
    CPolinomio polinomioR;
    // Declarar una referencia a un término cualquiera
    CTermino ptx = null; // puntero a un término
    // Leer los términos del primer sumando
    System.out.print("Términos del polinomio A "
        + "(para finalizar introduzca 0 para el\n"
        + "coeficiente y para los exponentes).\n\n");
    ptx = leerTermino();
    while ( ptx != null )
    {
        polinomioA.insertarTermino( ptx );
        ptx = leerTermino();
    }
    // Leer los términos del segundo sumando
    System.out.println("Términos del polinomio B "
        + "(para finalizar introduzca 0 para el\n"
        + "coeficiente y para los exponentes).\n\n");
    ptx = leerTermino();
    while ( ptx != null )
    {
        polinomioB.insertarTermino( ptx );
        ptx = leerTermino();
    }
    // Sumar los dos polinomios leídos
    polinomioR = polinomioA.sumar(polinomioB);

    // Visualizar el primer sumando
}

```

```

        System.out.print("Polinomio A: ");
        polinomioA.mostrarPolinomio();
        System.out.println();
        // Visualizar el segundo sumando
        System.out.print("Polinomio B: ");
        polinomioB.mostrarPolinomio();
        System.out.println();
        // Visualizar el polinomio suma
        System.out.print("Polinomio R: ");
        polinomioR.mostrarPolinomio();
        System.out.println();

        // Visualizar el valor del polinomio suma para x = 1 e y = 1
        System.out.println("Para x = 1 e y = 1, el valor es: " +
                           polinomioR.valorPolonomio(1, 1));
    }
}

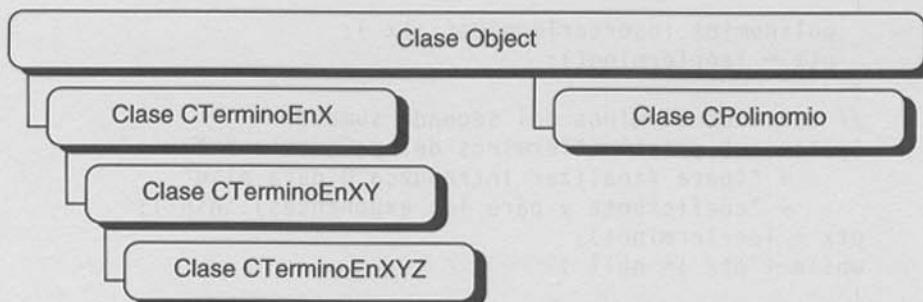
```

EJERCICIOS PROPUESTOS

Se quiere escribir un programa para manipular ecuaciones algebraicas o polinómicas dependientes de las variables x , y , z . Por ejemplo:

$$2x^3y - xyz^3 + 8.25 \text{ más } 5x^5y - 2x^3y + 7x^2 - 3 \text{ igual a } 5x^5y + 7x^2 - xyz^3 + 5.25$$

Cada término del polinomio será representado por una clase *CTerminoEnX*, *CTerminoEnXY* o *CTerminoEnXYZ* y cada polinomio por una clase *CPolinomio*. La clase *CTerminoEnXY* se derivará de *CTerminoEnX* y la clase *CTerminoEnXYZ* de *CTerminoEnXY*:



De acuerdo con el enunciado y apoyándose en el ejercicio anteriormente resuelto, construya las clases a las que hemos hecho referencia para que soporten al menos la misma funcionalidad que vio allí y realice un programa similar al anterior, para probar las clases construidas.

CAPÍTULO 11

© F.J.Ceballos/RA-MA

EXCEPCIONES

El lenguaje Java incorpora soporte para manejar situaciones anómalas, conocidas como “excepciones”, que pueden ocurrir durante la ejecución de un programa. Con el sistema de manipulación de excepciones de Java, un programa puede comunicar eventos inesperados a un contexto de ejecución más capacitado para responder a tales eventos anormales. Estas excepciones son manejadas por código fuera del flujo normal de control del programa.

Las excepciones proporcionan una manera limpia de verificar errores; esto es, sin abarrotar el código básico de una aplicación utilizando sistemáticamente los códigos de retorno de los métodos en sentencias **if** y **switch** para controlar los posibles errores que se puedan dar. Veamos con un ejemplo, a qué nos estamos refiriendo:

```
int códigoDeError = 0;
códigoDeError = leerFichero(nombre);
if ( códigoDeError != 0 )
{
    // Ocurrió un error al leer el fichero
    switch( códigoDeError )
    {
        case 1:
            // No se encontró el fichero
            // ...
            break;
        case 2:
            // El fichero está corrupto
            // ...
            break;
        case 3:
            // El dispositivo no está listo
            // ...
```

```

        break;
    default:
        // Otro error
        // ...
    }
}
else
{
    // Procesar los datos leídos del fichero
}

```

El código del ejemplo anterior trata de leer un fichero almacenado en el disco invocando al método *leerFichero*. Este método devuelve un valor 0 si se ejecuta satisfactoriamente y un valor distinto de 0 en otro caso. Para analizar este hecho se ha utilizado una sentencia **if**. En el caso de que se produzca un error, una sentencia **switch** se encargará de verificar qué es lo que ha ocurrido y tratar de resolverlo de la mejor forma posible. Lo que se persigue es que el programa no sea abortado inesperadamente por el sistema, sino diseñar una continuación o terminación normal dentro de lo ocurrido.

Observar el código que ha sido necesario escribir para tratar un posible error de no poder leer un fichero del disco. Pensemos ¿cuántos errores más podrían abortar nuestra aplicación? Para que esto no suceda ¿se imagina la complejidad del código escrito una vez añadido todo el necesario para tratar cada uno de ellos? El manejo de excepciones ofrece una forma de separar explícitamente el código que maneja los errores, del código básico de una aplicación, haciéndola más legible, lo que desemboca en un buen estilo de programación. Por ejemplo:

```

try
{
    // Código de la aplicación
}
catch(clase_de_excepción e)
{
    // Código de tratamiento de esta excepción
}
catch(otra_clase_de_excepción e)
{
    // Código de tratamiento para otra clase de excepción
}

```

Básicamente, el esquema anterior dice que si el código de la aplicación no puede realizar alguna operación, se espera lance una excepción que será tratada por el código de tratamiento especificado para esa clase de excepción, o en su defecto por Java.

A lo largo de este capítulo comprobará que: el manejo de excepciones reduce la complejidad de la programación; los métodos que invocan a otros no necesitan comprobar valores de retorno; si el método invocado finaliza de forma normal, el que llamó está seguro de que no ocurrió ninguna situación anómala; etc.

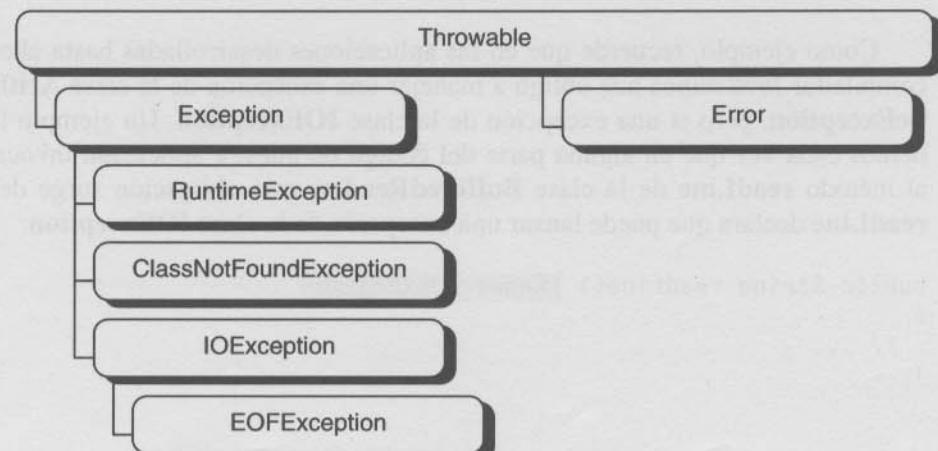
EXCEPCIONES DE JAVA

Durante el estudio de los capítulos anteriores, seguro que se habrá encontrado con excepciones como las siguientes:

<i>Clase de excepción</i>	<i>Significado</i>
ArithmaticException	Una condición aritmética excepcional ha ocurrido. Por ejemplo, una división por 0.
ArrayIndexOutOfBoundsException	Una matriz fue accedida con un índice ilegal (fuera de los límites permitidos).
NullPointerException	Se intentó utilizar null donde se requería un objeto.
NumberFormatException	Se intentó convertir una cadena con un formato inapropiado en un número.

¿Qué es lo que ocurrió entonces cuando durante la ejecución de su programa se lanzó una excepción? Seguramente el programa dejó de funcionar y Java visualizó algún mensaje acerca de lo ocurrido. Si no es esto lo que deseamos, tendremos que aprender a manipular las excepciones.

Las excepciones en Java son objetos de clases derivadas de la clase **Throwable** definida en el paquete **java.lang**. Por ejemplo, cuando se lanza una excepción **ArithmaticException**, automáticamente Java crea un objeto de esta clase. La figura siguiente muestra algunas de las clases de la jerarquía de excepciones:



Un objeto **Error** se crea cuando ha ocurrido un problema serio. Normalmente se lanza una excepción de este tipo, cuando durante la ejecución ocurre un error que involucra a la máquina virtual de Java, por lo que una aplicación normal no suele manipular este tipo de excepción.

La clase **Excepción** cubre las excepciones que una aplicación normal puede manipular. Tiene varias subclases entre las que destacan: **RuntimeException** e **IOException**.

La clase **RuntimeException** cubre las excepciones ocurridas al ejecutar operaciones sobre los datos que manipula la aplicación y que residen en memoria; se trata de excepciones que se lanzan en tiempo de ejecución, en contraposición a las que se lanzarían por causas no dependientes de la máquina virtual de Java, como sucedería cuando no se pudiera leer de un fichero del disco. Son ejemplos de excepciones de este tipo: **ArithmeiticException** o **NullPointerException**. Este grupo de excepciones pertenece al paquete **java.lang**.

La clase **IOException** cubre las excepciones ocurridas al ejecutar una operación de entrada o salida. Este grupo de excepciones pertenece al paquete **java.io**.

Las excepciones de tiempo de ejecución son *excepciones implícitas* y se corresponden con las subclases de **RuntimeException** y **Error**. Se dice que son implícitas porque son lanzadas por la máquina virtual de Java y por lo tanto, los métodos implementados en las aplicaciones no tienen que declarar que las lanzan, y aunque lo hicieran, cualquier otro método que los invoque no está obligado a manejarlas. El resto de las excepciones, como las que se corresponden con las subclases de **IOException**, son *excepciones explícitas*; esto significa que, si se quieren manipular, los métodos implementados en las aplicaciones tienen que declarar que las lanzan y en este caso, cualquier otro método que los invoque está obligado a manejarlas.

Como ejemplo, recuerde que en las aplicaciones desarrolladas hasta ahora el compilador Java nunca nos obligó a manejar una excepción de la clase **ArithmeiticException**, pero sí una excepción de la clase **IOException**. Un ejemplo lo tenemos cada vez que en alguna parte del código de nuestra aplicación invocamos al método **readLine** de la clase **BufferedReader**; esta obligación surge de que **readLine** declara que puede lanzar una excepción de la clase **IOException**:

```
public String readLine() throws IOException
{
    // ...
}
```

No se preocupe si no le quedó todo claro; a continuación aprenderá con detalle cómo atrapar, crear y lanzar excepciones, además de otras cosas.

MANEJAR EXCEPCIONES

Cuando un método se encuentra con una anomalía que no puede resolver, lo lógico es que *lance (throw)* una excepción, esperando que quien lo llamó directa o indirectamente la *atrapa (catch)* y maneje la anomalía. Incluso él mismo podría atrapar y manipular dicha excepción. Si la excepción no se atrapa, el programa finalizará automáticamente.

Por ejemplo, ¿recuerda la clase *Leer* desarrollada en el capítulo 5? Según puede observar a continuación, el método *dato* de esta clase invoca a *readLine* con el propósito de devolver un objeto **String** correspondiente a la cadena leída. Según se ha explicado anteriormente, *readLine* puede lanzar una excepción de la clase **IOException**. Para manejarla hay que atraparla, para lo cual se utiliza un bloque **catch**, y para poder atraparla hay que encerrar el código que puede lanzarla en un bloque **try**.

```
import java.io.*;
public class Leer
{
    public static String dato()
    {
        String sdato = "";
        try
        {
            // Definir un flujo de caracteres de entrada: flujoE
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader flujoE = new BufferedReader(isr);

            // Leer. La entrada finaliza al pulsar la tecla Entrar
            sdato = flujoE.readLine();
        }
        catch(IOException e)
        {
            System.err.println("Error: " + e.getMessage());
        }

        return sdato; // devolver el dato tecleado
    }
    // ...
}
```

Las palabras **try** y **catch** trabajan conjuntamente y pueden traducirse así: “poner a prueba un fragmento de código por si lanzara una excepción; si se ejecuta satisfactoriamente, continuar con la ejecución del programa; si no, atrapar la excepción lanzada y manejárla”.

Lanzar una excepción

Lanzar una excepción equivale a crear un objeto de la clase de la excepción para manipularlo fuera del flujo normal de ejecución del programa. Para lanzar una excepción se utiliza la palabra reservada **throw** y para crear un objeto, **new**. Por ejemplo, volviendo al método *datos* de la clase *Leer* expuesta anteriormente, si ocurre un error cuando se ejecute el método **readLine** se supone que éste ejecutará una sentencia similar a la siguiente:

```
if (error) throw new IOException();
```

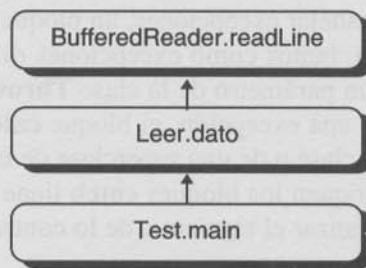
Esta sentencia lanza una excepción de la clase **IOException** lo que implica crear un objeto de esta clase. Un objeto de éstos contiene información acerca de la excepción, incluyendo su tipo y el estado del sistema cuando el error ocurrió.

Atrapar una excepción

Una vez lanzada la excepción, el sistema es responsable de encontrar a alguien que la atrape con el objetivo de manipularla. El conjunto de esos “alguien” es el conjunto de métodos especificados en la pila de llamadas hasta que ocurrió el error. Por ejemplo, consideremos la siguiente aplicación, que invoca al método *dato* de la clase *Leer* con la intención de leer un dato:

```
public class Test
{
    public static void main(String[] args)
    {
        String str;
        System.out.print("Dato: ");
        str = Leer.dato();
        // ...
    }
}
```

Cuando se ejecute esta aplicación y se invoque al método *dato*, la pila de llamadas crecerá como se observa en la figura siguiente:



Si al ejecutarse el método **readLine** ocurriera un error, según hemos visto anteriormente, éste lanzaría una excepción de la clase **IOException** que interrumpiría el flujo normal de ejecución. Después, el sistema buscaría en la pila de llamadas hacia abajo y comenzando por el propio método que produjo el error, uno que implemente un manejador que pueda atrapar esta excepción. Si el sistema descendiendo por la pila de llamadas no encontrara este manejador, el programa terminaría.

Para implementar un manejador para una clase de excepción hay que hacer las dos cosas que se indican a continuación:

1. Encerrar el código que puede lanzar la excepción en un bloque **try**. En la clase *Leer* presentada anteriormente, el método *dato* tiene un bloque **try** que encierra la llamada al método **readLine**, además de a otras sentencias:

```

try
{
    // ...
    sdato = flujoE.readLine();
}
  
```

2. Escribir un bloque **catch** capaz de atrapar la excepción lanzada. En la clase *Leer* presentada anteriormente, el método *dato* tiene un bloque **catch** capaz de atrapar excepciones de la clase **IOException** y de sus subclases:

```

catch(IOException e)
{
    System.err.println("Error: " + e.getMessage());
}
  
```

En este manejador se observa un parámetro *e* que referencia al objeto que se creó cuando se lanzó la excepción atrapada. Para manipularla, además de escribir el código que consideremos adecuado, disponemos de la funcionalidad proporcionada por la clase **IOException**, y a la que podremos acceder mediante el objeto *e*. Por ejemplo, el método **getMessage** devuelve una cadena con información acerca de la excepción ocurrida.

Cuando se trata de manejar excepciones, un bloque **try** puede estar seguido de uno o más bloques **catch**, tantos como excepciones diferentes tengamos que manejar. Cada **catch** tiene un parámetro de la clase **Throwable** o de alguna subclase de ésta. Cuando se lance una excepción, el bloque **catch** que la atrape será aquel cuyo parámetro sea de la clase o de una superclase de la excepción. Debido a esto, el orden en el que se coloquen los bloques **catch** tiene que ser tal, que cualquiera de ellos debe permitir alcanzar el siguiente, de lo contrario el compilador produciría un error.

Por ejemplo, si el primer bloque **catch** especifica un parámetro de la clase **Throwable**, ningún otro bloque que le siga podría alcanzarse; esto es, cualquier excepción lanzada sería atrapada por ese primer bloque, ya que cualquier referencia a una subclase puede ser convertida implícitamente por Java en una referencia a su superclase directa o indirecta.

En cambio, en el ejemplo siguiente, una excepción de la clase **EOFException** será atrapada por el primer bloque **catch**; una excepción de la clase **IOException** será atrapada por el bloque segundo; una excepción de la clase **FileNotFoundException**, subclase de **IOException**, será atrapada también por el bloque segundo; y una excepción de la clase **ClassNotFoundException**, subclase de **Exception**, será atrapada por el bloque tercero.

```

try
{
    // ...
}
catch(EOFException e)
{
    // Manejar esta clase de excepción
}
catch(IOException e)
{
    // Manejar esta clase de excepción o de alguna de sus subclases,
    // excepto EOFException
}
catch(Exception e)
{
    // Manejar esta clase de excepción o de alguna de sus subclases,
    // excepto EOFException e IOException
}

```

Un manejador de excepción, **catch**, sólo se puede utilizar justamente a continuación de un bloque **try** o de otro manejador de excepción (bloque **catch**). Las palabras clave **try** y **catch**, por definición, van seguidas de un bloque que encierra el código relativo a cada una de ellas, razón por la cual es obligatorio utilizar llaves: {}.

BLOQUE DE FINALIZACIÓN

Si no se trata de manejar excepciones, sino de realizar alguna acción por obligación (por ejemplo, liberar algún recurso externo que se haya adquirido, cerrar un fichero, etc.) ponga el código adecuado dentro de un bloque **finally** después del bloque **try** o de un bloque **catch**. El bloque **finally** deberá ser siempre el último.

La ejecución del bloque **finally** queda garantizada independientemente de que finalice o no la ejecución del bloque **try**. Quiere esto decir que aunque se abandone la ejecución del bloque **try** porque, por ejemplo, se ejecute una sentencia **return**, el bloque **finally** se ejecuta.

Para aclarar lo expuesto analicemos el siguiente ejemplo. Se trata de un método para escribir en un fichero datos procedentes de una matriz. ¿Recuerda la aplicación *Test* que escribimos en el capítulo anterior que operaba sobre un objeto *CBanco*? Pues, suponga ahora que queremos añadir al menú que presentaba esta aplicación, una opción más que permita escribir en un fichero en disco una lista con los nombres de los clientes del banco (en el capítulo siguiente aprenderemos a trabajar con ficheros). Para ello, añadiremos a la clase *Test* el método que se expone a continuación. Dicho método se ejecutará cuando se seleccione esa opción:

```
public static void escribirDatos(CBanco banco, String fich)
{
    PrintWriter fcli = null;
    CCuenta cliente;
    CListaClientes lista = new CListaClientes(banco.longitud());
    try
    {
        for (int i = 0; i < banco.longitud(); i++)
        {
            cliente = banco.clienteEn(i);
            lista.añadir(cliente.obtenerNombre(), i);
        }
        // Abrir el fichero para escribir. Se crea el flujo fcli;
        fcli = new PrintWriter(new FileWriter(fich));
        lista.escribir(fcli);
    }
    catch (IOException e)
    {
        System.out.println(e.getMessage());
    }
    finally
    {
        // Cerrar el fichero
        if (fcli != null) fcli.close();
    }
}
```

El objeto *lista* de la clase *CListaClientes* encapsula una matriz que almacenará la lista de los clientes del banco. El método *añadir* puede lanzar una excepción **ArrayIndexOutOfBoundsException** si el índice utilizado para acceder a los elementos de la matriz encapsulada en el objeto *CListaClientes* tiene un valor fuera de los límites establecidos. Los métodos *PrintWriter* y *escribir* pueden lanzar una excepción **IOException** si el fichero no puede abrirse, o bien ocurre un error de escritura. Las excepciones implícitas como **ArrayIndexOutOfBoundsException** no estamos obligados a manejarlas, pero las explícitas como **IOException** sí. Por eso se ha añadido un manejador para este tipo de excepción. Finalmente se ha añadido un bloque **finally** para garantizar que, ocurra lo que ocurra, el fichero será cerrado cuando se abandone el método *escribirDatos*.

¿Es realmente necesario el bloque **finally**? En el ejemplo anterior, el método *escribirDatos* no proporciona un manejador de excepciones por si ocurre un error durante la ejecución del método *añadir*. Entonces, si este error sucede ¿cómo cerraríamos el fichero? La respuesta es: el bloque **finally** es lo último que se ejecuta antes de abandonar el método *escribirDatos*, lo que ocurrirá cuando:

1. El bloque **try** finalice de ejecutarse satisfactoriamente.
2. Se lance una excepción **IOException**.
3. Se lance una excepción **ArrayIndexOutOfBoundsException**.

En el primer caso, después del bloque **try** se ejecutará el bloque **finally** y se saldrá del método *escribirDatos*. En el segundo caso, se ejecutará el bloque **catch** proporcionado por *escribirDatos*, después el bloque **finally** y se saldrá del método. Y en el tercer caso, se ejecutará el bloque **catch** proporcionado por el sistema, después el bloque **finally** y se saldrá del método.

Si hubiéramos incluido un manejador para el caso 3, podríamos cerrar el fichero en los bloques **try** y **catch** y prescindir del bloque **finally**, pero a costa de duplicar código y hacer menos legible el programa.

Los bloques **try** y **finally** pueden también utilizarse conjuntamente, sin que sea necesario incluir un bloque **catch**.

DECLARAR EXCEPCIONES

Java requiere que cualquier método que pueda lanzar una excepción la declare o la atrape. Por ejemplo ¿veamos qué ocurre si en el método *escribirDatos* presentado en el ejemplo anterior eliminamos el bloque **catch**? Observaremos que cuando Java compile este método indicará mediante un mensaje de error que la excepción **IOException** no está interceptada ni declarada por el método *escri-*

birDatos. Y ¿por qué sucede esto? Porque el método **FileWriter** (se trata de un constructor) invocado por *escribirDatos* está definido en la biblioteca de Java así:

```
public FileWriter(String nombre_fichero) throws IOException
{
    // Cuerpo del método
}
```

La palabra reservada **throws** permite a un método declarar las lista de excepciones (nombres de las clases de excepción separados por comas) que puede lanzar. Esto tiene dos lecturas:

1. Dar información a los usuarios de la clase que proporciona este método sobre las cosas anormales que puede hacer el método.
2. Escribir un método que lance una o más excepciones que no sean atrapadas por el propio método, sino por los métodos que lo llamen; al fin y al cabo, lo único que estamos haciendo cuando procedemos de esta forma es no anticiparnos a las necesidades que pueda tener el usuario en cuanto al tratamiento de la excepción se refiere.

Siguiendo con lo expuesto, si no deseáramos que el método *escribirDatos* atrapara las excepciones debidas a las anomalías que pudieran ocurrir dentro de él, tendríamos que escribirlo así:

```
public static void escribirDatos(CBanco banco, String fich) throws IOException
{
    PrintWriter fcli = null;
    CCuenta cliente;
    CListaClientes lista = new CListaClientes(banco.longitud());
    try
    {
        for (int i = 0; i < banco.longitud(); i++)
        {
            cliente = banco.clienteEn(i);
            lista.añadir(cliente.obtenerNombre(), i);
        }
        // Abrir el fichero para escribir. Se crea el flujo fcli;
        fcli = new PrintWriter(new FileWriter(fich));
        lista.escribir(fcli);
    }
    finally
    {
        // Cerrar el fichero
        if (fcli != null) fcli.close();
    }
}
```

Se puede observar que ahora no hay un bloque **catch** que intercepte la excepción **IOException**. Esta forma de proceder obligará a cualquier método que invoque a *escribirDatos* a atrapar la excepción. Por ejemplo:

```
public static void main(String[] args)
{
    // ...
    case 8: // escribir
    try
    {
        flujoS.print("Fichero: "); nombre = Leer.dato();
        escribirDatos(banco, nombre);
    }
    catch (IOException e)
    {
        System.out.println(e.getMessage());
    }
    break;
    // ...
}
```

En el caso de tener que redefinir el método en una subclase, la declaración de cuántas excepciones que puede lanzar puede ser inferior, nunca superior; incluso puede no lanzar ninguna.

CREAR Y LANZAR EXCEPCIONES

En alguna ocasión puede que necesitemos crear nuestras propias excepciones, a pesar de que en la biblioteca de clases de Java hay una gran cantidad de ellas que podemos utilizar sin más. En cualquier caso, todos los tipos de excepción se corresponden con una clase derivada de **Throwable**, clase raíz de la jerarquía de clases de excepciones de Java. Más aún, el paquete **java.lang** proporciona dos subclases de **Throwable** que agrupan las excepciones que se pueden lanzar, como consecuencia de los errores que pueden ocurrir en un programa, en dos clases: **Error** y **Exception**. Los errores que ocurren en la mayoría de los programas se corresponden con excepciones de alguna de las subclases de **Exception**, razón por la que esta clase será la superclase directa o indirecta de las nuevas clases de excepción que creemos, quedando la clase **Error** reservada para el tratamiento de los errores que se puedan producir en la máquina virtual de Java.

En general, crearemos un nuevo tipo de excepción cuando queramos manejar un determinado tipo de error no contemplado por las excepciones proporcionadas por la biblioteca de Java. Por ejemplo, para crear un tipo de excepción *EValorNoValido*, con la intención de manejar un error “valor no válido”, podemos diseñar una clase como la siguiente:

```

public class EValorNoValido extends Exception
{
    public EValorNoValido() {}
    public EValorNoValido(String mensaje)
    {
        super(mensaje);
    }
    // ...
}

```

Según se observa en este ejemplo, la superclase de la nueva clase de excepción *EValorNoValido* es **Exception**, e implementa dos constructores: uno sin parámetros y otro con un parámetro de tipo **String**; esto es lo más habitual. El parámetro de tipo **String** es el mensaje que devolverá el método **getMessage** heredado de la clase **Throwable** a través de **Exception**. Para ello el constructor *EValorNoValido* debe invocar al constructor de la superclase y pasar como argumento dicha cadena, la cual será almacenada como un miembro de datos de la clase **Throwable**.

La clase de excepción *EValorNoValido* relacionada con el error “valor no válido” ya está creada. Lógicamente, siempre que se implemente una clase de excepción es porque durante el desarrollo de una clase, por ejemplo *CMiClase*, se ha observado que su código para determinados valores durante la ejecución, puede presentar una anomalía de la que los usuarios de esa clase deben ser informados para que la puedan tratar.

¿Qué aspecto tiene *CMiClase*? Según lo expuesto, el código que implementa esta clase ante determinados valores produce un error. Habrá entonces que añadir el código que chequee si se producen esos valores y en caso afirmativo lanzar la excepción programada para este caso. Por ejemplo:

```

public class CMiClase
{
    // ...
    public void m(int a)
    {
        // ...
        if (a == 0)
            throw new EValorNoValido("Error: valor cero");
        // ...
    }
    // ...
}

```

Lanzar, una excepción equivale a crear un objeto de ese tipo de excepción. En el ejemplo anterior se observa que la circunstancia que provoca el error es que el

parámetro *a* del método *m* de *CMiClase* sea 0; en este caso, el método *m* lanza (**throw**) una excepción de la clase *EValorNoValido* creando un objeto de esta clase. Para crear (**new**) ese objeto se invoca al constructor *EValorNoValido* pasando como argumento, en este caso, la cadena "Error: valor cero".

Si un método *m* lanza una excepción debe declararlo, para que los usuarios de *CMiClase*, que es quien proporciona el método *m*, estén informados sobre las cosas anormales que puede hacer dicho método.

```
public class CMiClase
{
    // ...
    public void m(int a) throws EValorNoValido
    {
        // ...
    }
    // ...
}
```

Otra alternativa es que el propio método que lanza la excepción la atrape, como puede observar en el ejemplo siguiente. Lo que sucede es que escribir un método que lance una o más excepciones y él mismo las atrape es anticiparnos a las necesidades que pueda tener el usuario de la clase que proporciona ese método, en cuanto al tratamiento de la excepción se refiere.

```
public class CMiClase
{
    // ...
    public void m(int a)
    {
        // ...
        try
        {
            if (a == 0)
                throw new EValorNoValido("Error: valor cero");
        }
        catch (EValorNoValido e)
        {
            System.out.println(e.getMessage());
        }
        // ...
    }
    // ...
}
```

Combinar ambas formas (declarar la excepción y además atraparla) no sirve de nada, porque si un método lanza una excepción y la atrapa, en el supuesto de

que en la pila de llamadas quedarán otras que pudieran atraparla, no serán tenidas en cuenta; esto es, sólo se ejecuta el manejador del método por el que haya pasado el flujo de control más recientemente.

En este instante tenemos una clase, *CMiClase*, cuyo método *m* declara una excepción de tipo *EValorNoValido*. Un método de cualquier otra clase que utilice el método *m* de esta clase debe detectar esa posible anomalía, de lo contrario el compilador Java mostrará un error. Dicho método expresará esa necesidad encerrando el código que puede *intentar* (*try*) producir tal anomalía en un bloque *try* con un manejador para esa excepción. Por ejemplo:

```
public class Test
{
    public static void main(String[] args)
    {
        int x = 0;
        CMiClase obj = new CMiClase();
        try
        {
            obj.m(x);
        }
        catch (EValorNoValido e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println("Continúa la ejecución");
    }
}
```

Tenga en cuenta que un manejador tiene alcance a las variables locales del método donde se ha definido pero no a las variables locales al bloque *try* o a otros bloques *catch*.

Cuando un método utilizando *throw* lanza una excepción, crea un objeto de la clase de excepción especificada, que interrumpe el flujo de ejecución del programa y vuelve por la pila de llamadas hasta encontrar uno que sepa atrapar la excepción (que contenga un bloque *catch* con un argumento de la clase de la excepción o de alguna de sus superclases). La ejecución del programa se transfiere entonces, directamente al método que atrapó la excepción para que ejecute el manejador. Si el manejador, una vez ejecutado, permite que la aplicación continúe, la ejecución se transfiere a la primera línea ejecutable que haya a continuación del último manejador del bloque *try*. Según esto, cuando se ejecute el método *main* del ejemplo anterior se obtendrá el siguiente resultado:

```
Error: valor cero
Continúa la ejecución
```

Si un método lanza una excepción y en la vuelta por la pila de llamadas no se encuentra uno que la atrape, el programa finalizará. En cambio, si se encuentra un manejador para esa excepción, se ejecuta. En el supuesto de que en la pila de llamadas quedaran otros métodos que pudieran atraparla, no serán tenidos en cuenta; esto es, sólo se tiene en cuenta el manejador del método por el que haya pasado el flujo de control más recientemente.

A su vez, si el método contiene una lista de manejadores sólo se ejecutará el correspondiente a la excepción lanzada; esto es, el comportamiento es el mismo que el de una sentencia **switch**, pero con la diferencia de que los **case** necesitan sentencias **break** y los **catch** no.

Según hemos visto, una excepción se atrapa en un bloque **catch** que declare un argumento de su clase o superclase; pero como lo que se lanza es un objeto, puesto que **throw** especifica a continuación una llamada al constructor de la clase de excepción, si necesitáramos transmitir información adicional desde el punto de lanzamiento al manejador, lo podemos hacer a través de argumentos en el constructor.

Una excepción se considera manejada desde el momento en que se entra en su manejador, así que cualquier otra excepción lanzada desde el cuerpo de éste, deberá ser atrapada por algún otro método cuya llamada se encuentre en la pila de llamadas; si la excepción no es atrapada, el programa finaliza. Esto explica por qué el siguiente código no provoca un bucle infinito:

```
public void otroMétodo() throws EValorNoValido
{
    // ...
    try
    {
        // ...
    }
    catch(EValorNoValido e)
    {
        // ...
        throw EValorNoValido();
    }
    // ...
}
```

Según lo expuesto anteriormente, una vez atrapada una excepción, el manejador puede decidir volver a lanzarla para que sea procesada por otro manejador. Esto implica que el método declare que puede lanzar ese tipo de excepción. En el ejemplo anterior se puede observar que *otroMétodo* declara que puede lanzar una excepción de la clase *EValorNoValido*.

CUÁNDO UTILIZAR EXCEPCIONES Y CUÁNDO NO

No todas los programas necesitan responder lanzando una excepción a cualquier situación anómala que se produzca. Por ejemplo, si partiendo de unos datos de entrada estamos haciendo una serie de cálculos más o menos complejos con la única finalidad de observar unos resultados, quizás la respuesta más adecuada a un error sea interrumpir sin más el programa, no antes de haber lanzado un mensaje apropiado y haber liberado los recursos adquiridos que aún no hayan sido liberados. Otro ejemplo, podemos utilizar la clase de excepción **ArrayIndexOutOfBoundsException** para manejar el error que se produce cuando se rebasan los límites de una matriz, pero es más fácil utilizar el miembro **length** de la matriz para prevenir que esto no suceda.

En cambio, si estamos construyendo una biblioteca estamos obligados a evitar todos los errores que se puedan producir cuando su código sea ejecutado por cualquier programa que la utilice.

Por último, no todas las excepciones tienen que servir para manipular errores. Puede también manejar excepciones que no sean errores.

EJERCICIOS RESUELTOS

1. Añadir a la aplicación realizada en el capítulo 9 sobre el mantenimiento de una lista de teléfonos, el código necesario para manejar la excepción **OutOfMemoryError** que Java lanza cuando un método intenta realizar una asignación dinámica de memoria y no hay disponible un bloque de memoria del tamaño requerido.

La clase de excepción **OutOfMemoryError** pertenece a la jerarquía cuya clase raíz es **Error**. Anteriormente se expuso que la jerarquía derivada de la clase **Error** estaba reservada para el tratamiento de los errores que se puedan producir en la máquina virtual de Java. El error de falta de memoria para asignación es un error típico que puede surgir en un programa que necesite reservar repetidas veces bloques de memoria de un tamaño considerable.

Como ejemplo de tratamiento de este error vamos a añadir a la clase *CListaTfnos* de la aplicación “lista de teléfonos” realizada en el capítulo 9 un manejador para esta clase de excepción. Cargue esta aplicación, visualice la clase *CListaTfnos* y compruebe los métodos que utilizan **new** para reservar memoria; concretamente, cuando se crea un objeto *CListaTfnos* y cada vez que se necesita aumentar o disminuir el tamaño de la matriz de objetos *CPersona*. Para dar solución al problema propuesto, añadiremos un nuevo método a la clase *CListaTfnos* denominado *asignarMemoria*. Este método tendrá un parámetro de tipo entero

que se corresponderá con el número de elementos de la matriz para los que deseamos asignar memoria y devolverá una referencia al nuevo bloque de memoria asignado. Si no hay un bloque de memoria del tamaño solicitado, el manejador de la excepción **OutOfMemoryError** visualizará el mensaje de error predeterminado y devolverá la referencia al bloque de memoria existente.

```

public class CListaTfnos
{
    private CPersona[] listaTeléfonos; // matriz de objetos
    private int nElementos; // número de elementos de la matriz

    private CPersona[] asignarMemoria(int nElementos)
    {
        try
        {
            return new CPersona[nElementos];
        }
        catch (OutOfMemoryError e)
        {
            System.out.println(e.getMessage());
            return listaTeléfonos;
        }
    }

    public CListaTfnos()
    {
        // Crear una lista vacía
        nElementos = 0;
        listaTeléfonos = asignarMemoria(nElementos);
    }

    private void unElementoMás(CPersona[] listaActual)
    {
        nElementos = listaActual.length;
        listaTeléfonos = asignarMemoria(nElementos + 1);
        // Copiar la lista actual
        for ( int i = 0; i < nElementos; i++ )
            listaTeléfonos[i] = listaActual[i];
        nElementos++;
    }

    private void unElementoMenos(CPersona[] listaActual)
    {
        if (listaActual.length == 0) return;
        int k = 0;
        nElementos = listaActual.length;
        listaTeléfonos = asignarMemoria(nElementos - 1);
        // Copiar la lista actual
        for ( int i = 0; i < nElementos; i++ )
    }
}

```

```

        if (listaActual[i] != null)
            listaTeléfonos[k++] = listaActual[i];
        nElementos--;
    }
    // ...
}

```

2. En el capítulo 5 implementamos una clase *Leer* con los miembros siguientes:

Método	Significado
<i>dato</i>	Devuelve un objeto String correspondiente a la cadena tecleada.
<i>datoShort</i>	Devuelve el dato de tipo short tecleado, o el valor Short.MIN_VALUE si el dato tecleado no se corresponde con un short .
<i>datoInt</i>	Devuelve el dato de tipo int tecleado, o el valor Integer.MIN_VALUE si el dato tecleado no se corresponde con un int .
<i>datoLong</i>	Devuelve el dato de tipo long tecleado, o el valor Long.MIN_VALUE si el dato tecleado no se corresponde con un long .
<i>datoFloat</i>	Devuelve el dato de tipo float tecleado, o el valor Float.NaN si el dato tecleado no se corresponde con un float .
<i>datoDouble</i>	Devuelve el dato de tipo double tecleado, o el valor Double.NaN si el dato tecleado no se corresponde con un double .

Al principio de este capítulo explicamos el manejador de excepciones que incluye el método *datos*. Como ejercicio se trata ahora de explicar y modificar los manejadores de excepciones incluidos en el resto de los métodos.

El método *datoShort* fue implementado de la forma siguiente:

```

public static short datoShort()
{
    try
    {
        return Short.parseShort(dato());
    }
    catch(NumberFormatException e)
    {
        return Short.MIN_VALUE; // valor más pequeño
    }
}

```

Este método devuelve el valor returned a su vez por el método `parseShort` de la clase `Short`, resultado de convertir la cadena de caracteres devuelta por `dato`. Pero ¿qué ocurre si la cadena de caracteres devuelta por `dato` no se corresponde con un `short`? Pues que al ejecutarse el método `parseShort`, Java lanza una excepción `NumberFormatException` que es atrapada por el manejador, que devuelve el valor `Short.MIN_VALUE`.

Una alternativa al manejador anterior podría ser otro que ante un dato no válido (por ejemplo: xxx; 3.5; 3,5; etc.) solicitará teclear un dato correcto. Haremos una excepción para el carácter fin de fichero (`Ctrl+Z`). En este caso, `dato` devolverá `null` (porque `readLine` devuelve `null`) señal para que nuestro método devuelva un valor que sirva para identificar que se pulsó `Ctrl+Z`; conviene, si es posible, que este valor no pertenezca al conjunto de valores válidos que se quiera leer. De esta forma, podremos utilizar `Ctrl+Z` como marca para finalizar una entrada masiva de datos. Según esto, podemos reescribir el método `datoShort` así:

```
public static short datoShort()
{
    try
    {
        String sdato = dato();
        if (sdato == null)
        {
            System.out.println();
            return Short.MIN_VALUE;
        }
        else
            return Short.parseShort(sdato);
    }
    catch(NumberFormatException e)
    {
        System.out.print("Ese dato no es válido. Teclee otro: ");
        return datoShort();
    }
}
```

Como ejemplo, el siguiente código, utilizando el método anterior, permite introducir datos de tipo `int` hasta pulsar las teclas `Ctrl+Z`:

```
int eof = Integer.MIN_VALUE, i = 0;
int[] a = new int[100];

System.out.println("Introducir datos. Finalizar con Ctrl+Z");
System.out.print("Dato int: ");
while (i < 100 && (a[i] = Leer.datoInt()) != eof)
{
    i++;
}
```

```
    System.out.print("Dato int: ");
}
} // El control saliendo del bucle para la siguiente iteración.
```

Análogamente podríamos escribir un manejador para el método *datoFloat*:

```
public static float datoFloat()
{
    try
    {
        String sdato = dato();
        if (sdato == null)
        {
            System.out.println();
            return Float.NaN; // No es un Número; valor float.
        }
        else
        {
            Float f = new Float(sdato);
            return f.floatValue();
        }
    }
    catch(NumberFormatException e)
    {
        System.out.print("Ese dato no es válido. Teclee otro: ");
        return datoFloat();
    }
}
```

Si tenemos en cuenta que *Float(dato())* lanza la excepción **NullPointerException** cuando el método *dato* devuelve **null**, el método anterior podría escribirse también así:

```
public static float datoFloat()
{
    try
    {
        Float f = new Float(dato());
        return f.floatValue();
    }
    catch(NumberFormatException e)
    {
        System.out.print("Ese dato no es válido. Teclee otro: ");
        return datoFloat();
    }
    catch(NullPointerException e)
    {
        return Float.NaN; // No es un Número; valor float.
    }
}
```

Como ejemplo, el siguiente código utilizando el método anterior permite introducir datos de tipo **float** hasta pulsar las teclas *Ctrl+Z*:

```
boolean eof = true;
float[] a = new float[100];
int i = 0;

System.out.println("Introducir datos. Finalizar con Ctrl+Z");
System.out.print("Dato float: ");
while (i < 100 && Float.isNaN(a[i]) = Leer.datoFloat()) != eof)
{
    i++;
    System.out.print("Dato float: ");
}
```

La variable *eof* se ha utilizado simplemente por motivos didácticos. Quiere esto decir que la sentencia **while** podría escribirse también así:

```
while (i < 100 && !Float.isNaN(a[i]) = Leer.datoFloat())
{
    i++;
    System.out.print("Dato float: ");
}
```

EJERCICIOS PROPUESTOS

1. Implementar los manejadores para el resto de los métodos de la clase *Leer* de forma análoga a como se ha hecho en el ejercicio anterior.
2. La clase *CCuenta* que implementamos en el capítulo 10, tiene un método *reintegro* que muestra un mensaje “Error: no dispone de saldo” cuando se intenta retirar una cantidad y no hay suficiente saldo. Modifique esta clase para que el método *reintegro* lance una excepción *ESaldoInsuficiente*.

La clase *ESaldoInsuficiente* tendrá dos atributos, uno de la clase *CCuenta* para hacer referencia a la cuenta que causó el problema, y otro de tipo **double** para almacenar la cantidad solicitada. Asimismo tendrá un constructor y el método *mensaje*. El constructor *ESaldoInsuficiente* tendrá dos parámetros que harán referencia a la cuenta causante del problema y a la cantidad solicitada. El método *mensaje* no tiene argumentos, generará un mensaje de error basado en la información almacenada en los atributos y devolverá un objeto **String** con ese mensaje.

Cuando haya finalizado pruebe la jerarquía de la clase *CCuenta* junto con la clase *CBanco* que también implementamos en ese capítulo.

CAPÍTULO 12

© F.J.Ceballos/RA-MA

TRABAJAR CON FICHEROS

Todos los programas realizados hasta ahora obtenían los datos necesarios para su ejecución de la entrada estándar y visualizaban los resultados en la salida estándar. Por otra parte, una aplicación podrá retener los datos que manipula en su espacio de memoria, sólo mientras esté en ejecución; es decir, cualquier dato introducido se perderá cuando la aplicación finalice.

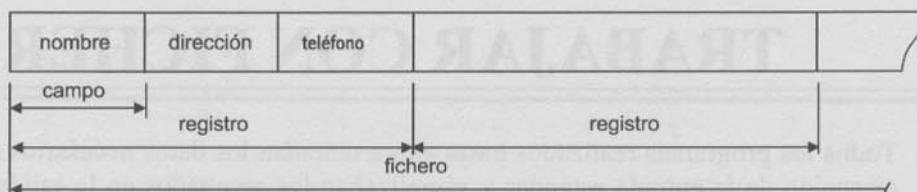
Por ejemplo, si hemos realizado un programa con la intención de construir una agenda, lo ejecutamos y almacenamos los datos *nombre*, *apellidos* y *teléfono* de cada uno de los componentes de la agenda en una matriz, los datos estarán disponibles mientras el programa esté en ejecución. Si finalizamos la ejecución del programa y lo ejecutamos de nuevo, tendremos que volver a introducir de nuevo todos los datos.

La solución para hacer que los datos persistan de una ejecución para otra es almacenarlos en un fichero en el disco en vez de en una matriz en memoria. Entonces, cada vez que se ejecute la aplicación que trabaja con esos datos, podrá leer del fichero los que necesite y manipularlos. Nosotros procedemos de forma análoga en muchos aspectos de la vida ordinaria; almacenamos los datos en fichas y guardamos el conjunto de fichas en lo que generalmente denominamos fichero o archivo.



Desde el punto de vista informático, un fichero o archivo es una colección de información que almacenamos en un soporte magnético para poderla manipular en cualquier momento. Esta información se almacena como un conjunto de registros, conteniendo todos ellos, generalmente, los mismos campos. Cada campo almacena un dato de un tipo predefinido o de un tipo definido por el usuario. El registro más simple estaría formado por un carácter.

Por ejemplo, si quisiéramos almacenar en un fichero los datos relativos a la agenda de teléfonos a la que nos hemos referido anteriormente, podríamos diseñar cada registro con los campos *nombre*, *dirección* y *teléfono*. Según esto y desde un punto de vista gráfico, puede imaginarse la estructura del fichero así:



Cada campo almacenará el dato correspondiente. El conjunto de campos descritos forma lo que hemos denominado registro, y el conjunto de todos los registros forman un fichero que almacenaremos, por ejemplo, en el disco bajo un nombre.

Por lo tanto, para manipular un fichero que identificamos por un nombre, son tres las operaciones que tenemos que realizar: abrir el fichero, escribir o leer registros del fichero y cerrar el fichero. En la vida ordinaria hacemos lo mismo, abrimos el cajón que contiene las fichas (fichero), cogemos una ficha (registro) para leer datos o escribir datos y, finalizado el trabajo con la ficha, la dejamos en su sitio y cerramos el cajón de fichas (fichero).

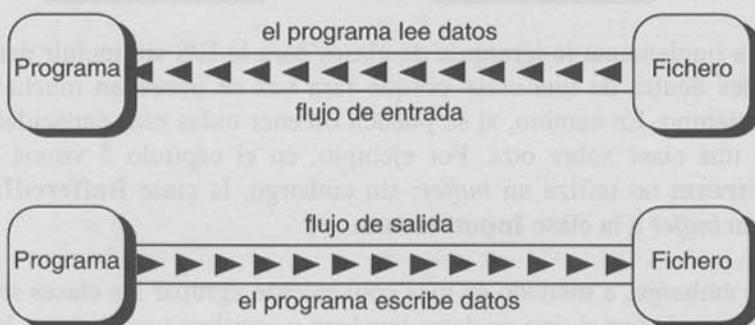
En programación orientada a objetos, hablaremos de objetos más que de registros, y de sus atributos más que de campos.

Podemos agrupar los ficheros en dos tipos: ficheros de la aplicación (son los ficheros *.java*, *.class*, etc. que forman la aplicación) y ficheros de datos (son los que proveen de datos a la aplicación). A su vez, Java ofrece dos tipos diferentes de acceso a los ficheros de datos: secuencial y aleatorio.

Para dar soporte al trabajo con ficheros, la biblioteca de Java proporciona varias clases de entrada/salida (E/S) que permiten leer y escribir datos a, y desde, ficheros y dispositivos (en el capítulo 5 trabajamos con algunas de ellas).

VISIÓN GENERAL DE LOS FLUJOS DE E/S

La comunicación entre el programa y el origen o el destino de cierta información, se realiza mediante un *flujo* de información (en inglés *stream*) que no es más que un objeto que hace de intermediario entre el programa, y el origen o el destino de la información. Esto es, el programa leerá o escribirá en el *flujo* sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que se traduce en una facilidad más a la hora de escribir programas.

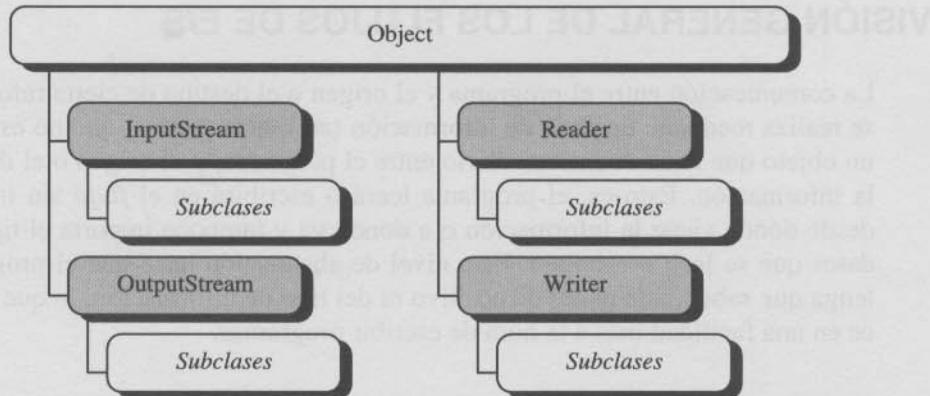


Entonces, para que un programa pueda obtener información desde un fichero tiene que abrir un flujo y leer la información en él almacenada. Análogamente, para que un programa puede enviar información a un fichero tiene que abrir un flujo y escribir la información en el mismo.

Los algoritmos para leer y escribir datos son siempre más o menos los mismos:

Leer	Escribir
<i>Abrir un flujo desde un fichero</i>	<i>Abrir un flujo hacia un fichero</i>
<i>Mientras haya información</i>	<i>Mientras haya información</i>
<i> Leer información</i>	<i> Escribir información</i>
<i>Cerrar el flujo</i>	<i>Cerrar el flujo</i>

El paquete **java.io** de la biblioteca estándar de Java, contiene una colección de clases que soportan estos algoritmos para leer y escribir. Estas clases se dividen en dos grupos distintos, según se muestra en la figura siguiente. El grupo de la izquierda ha sido diseñado para trabajar con datos de tipo **byte** (8 bits) y el de la derecha con datos de tipo **char** (16 bits). Ambos grupos presentan clases análogas que tienen interfaces casi idénticas, por lo que se utilizan de la misma manera.



Java implementa la jerarquía de clases para la E/S sin incluir demasiadas capacidades dentro de una clase porque rara vez se necesitan muchas de ellas al mismo tiempo. En cambio, sí se pueden obtener todas esas capacidades superponiendo una clase sobre otra. Por ejemplo, en el capítulo 5 vimos que la clase **InputStream** no utiliza un *buffer*; sin embargo, la clase **BufferedInputStream** añade un *buffer* a la clase **InputStream**.

Sin embargo, a menudo es más conveniente agrupar las clases según su finalidad en vez de por el tipo de datos que leen o escriben (caracteres o bytes). Desde este punto de vista distinguimos flujos que simplemente permiten leer y escribir datos y flujos que, además, procesan la información leída o escrita.

Flujos que no procesan los datos de E/S

La tabla siguiente lista las subclases que permiten definir flujos para leer o escribir información en un medio sin realizar ningún proceso añadido:

Medio	Flujo de caracteres	Flujo de bytes
Memoria	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
	StringReader StringWriter	StringBufferInputStream
Fichero	FileReader FileWriter	FileInputStream FileOutputStream
Tubería	PipedReader PipedWriter	PipedInputStream PipedOutputStream

Un programa que cree un flujo de alguna de estas clases podrá leer o escribir información en algunos de los medios especificados: una matriz en memoria, un fichero en el disco o una tubería. Una tubería es un flujo que permite comunicar dos subprocessos para transferencia de información entre uno y otro. Veremos esto con más detalle en el capítulo dedicado a hilos.

El siguiente ejemplo muestra cómo utilizar las clases **CharArrayReader** y **CharArrayWriter**. El resto de las clases expuestas en la tabla anterior se utilizan de forma análoga.

```
import java.io.*;
public class Test
{
    public static void main(String[] args)
    {
        char[] m1 = new char[80];
        char[] m2 = new char[80];
        int car, i = 0;

        // Almacenar datos en la matriz m1
        for (car = 'a'; car <= 'z'; car++)
            m1[i++] = (char)car;

        // Abrir un flujo, flujoE, desde la matriz m1
        CharArrayReader flujoE = new CharArrayReader(m1);
        // Abrir un flujo, flujoS, hacia una matriz temporal
        CharArrayWriter flujoS = new CharArrayWriter();
        try
        {
            // Leer de flujoE y escribir en flujoS
            while ((car = flujoE.read()) != -1)
                flujoS.write(car);

            // Copiar en m2 los datos enviados al flujoS
            m2 = flujoS.toCharArray();
            System.out.println(m2);
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
        }
        finally
        {
            // Cerrar los flujos
            flujoE.close();
            flujoS.close();
        }
    }
}
```

Qué hace este programa:

1. Almacena datos en una matriz *m1*.
2. Abre un flujo de entrada desde la matriz *m1*. Entonces, el programa puede leer datos de este flujo de forma similar a como lo hace del flujo estándar de entrada.
3. Abre un flujo de salida hacia una matriz temporal. Como el tamaño de la matriz no ha sido especificado, éste se ajustará a la cantidad de información que se envíe al flujo. De esta forma, el programa puede escribir datos en este flujo de forma similar a como lo hace en el flujo estándar de salida.
4. Lee datos del flujo abierto desde *m1*, *flujoE*, y los escribe en el flujo de salida, *flujoS*.
5. Copia los datos enviados a *flujoS* en una matriz *m2* y la muestra.

Flujos que procesan los datos de E/S

La tabla siguiente lista las subclases que permiten definir flujos para leer o escribir información en un medio, además de realizar alguna operación como añadir un *buffer*, un filtro, realizar una conversión, etc.:

<i>Operación</i>	<i>Flujo de caracteres</i>	<i>Flujo de bytes</i>
Establecer un buffer	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Establecer un filtro	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Conversión (bytes - cars.)	InputStreamReader OutputStreamWriter	
Concatenación		SequenceInputStream
Seriación		ObjectInputStream ObjectOutputStream
Conversión (datos)		DataInputStream DataOutputStream
Contar	LineNumberReader	LineNumberInputStream
Mirar anticipadamente	PushbackReader	PushbackInputStream
Escribir	PrintWriter	PrintStream

Un programa que cree un flujo de alguna de estas clases podrá leer o escribir información además de ejecutar la operación para la que ha sido diseñado. Por ejemplo, un flujo de la clase **PushbackReader** (derivada de **FilterReader** que a su vez se deriva **Reader**) es útil cuando, por ejemplo, un analizador necesita mirar el siguiente carácter en la entrada con el fin de determinar qué hacer a continuación; para ello, el analizador leerá el carácter y después lo devolverá a la entrada para que pueda ser leído por el código que tenga que ejecutarse a continuación.

Alguno de los métodos que proporciona la clase **PushbackReader** son:

<i>Método</i>	<i>Significado</i>
close	Cierra el flujo.
read	Lee un único carácter, o bien una matriz de caracteres.
unread	Devuelve a la entrada un único carácter, o bien todo o parte de una matriz de caracteres.
ready	Devuelve true si se puede leer del flujo porque hay caracteres disponibles; en otro caso devuelve false . Este método, heredado de la clase Reader , permite realizar operaciones análogas al método available de la clase InputStream .

Como ejemplo, vamos a modificar la última versión de la clase *Leer* que realizamos en el capítulo anterior, con el fin de añadirla algunas capacidades más, como mirar cuál es el siguiente carácter en la entrada, leer un solo carácter o limpiar el flujo de entrada.

Si echamos una ojeada al método *dato* de la clase *Leer* observaremos que define un flujo local. Pero los métodos que ahora vamos a añadir necesitan leer de ese mismo flujo; por lo tanto, tendremos que declararlo como un miembro de la clase; dicho miembro tendrá que ser declarado estático puesto que los métodos también son estáticos. Recuerde que esto lo hicimos así para poder utilizar las capacidades que proporciona la clase *Leer* sin necesidad de tener que utilizar un objeto de la misma.

Para poder implementar la capacidad de mirar cuál es el siguiente carácter en la entrada, el flujo debe de ser de la clase **PushbackReader**. El constructor de esta clase requiere un argumento que haga referencia a un objeto, origen de los datos a leer, de la clase **Reader** o de alguna de sus subclases y opcionalmente acepta un segundo argumento que especifica el tamaño del *buffer* para almacenar los caracteres devueltos por el método **unread** de **PushbackReader**, que por omisión es uno. Por lo tanto, la clase *Leer* puede ser ahora así:

```

import java.io.*;

public class Leer
{
    // Definir un flujo de caracteres de entrada: flujoE
    private static InputStreamReader isr =
        new InputStreamReader(System.in);
    private static PushbackReader flujoE = new PushbackReader(isr);

    public static void limpiar()
    {
        // limpiar flujoE
    }

    public static char mirar()
    {
        // retornar el primer carácter disponible sin extraerlo
    }

    public static char carácter()
    {
        // devolver el siguiente carácter de la entrada
    }

    public static String dato()
    {
        // devolver un String que almacene el dato tecleado
    }

    // ...
}

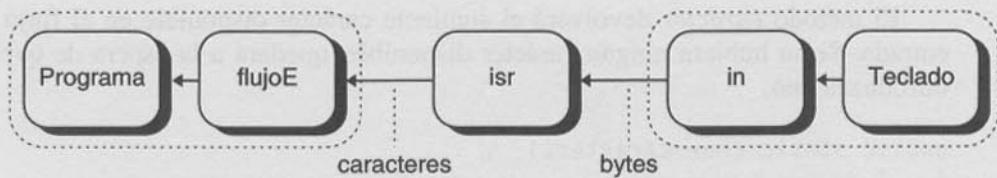
```

Ahora, la clase *Leer* define un miembro *flujoE* de la clase **PushbackReader** que se corresponde con el flujo abierto desde del origen de los caracteres. Como el origen real de los datos va a ser el teclado (dispositivo vinculado con **System.in** que proporciona *bytes*), es preciso conectar ambos flujos por otro que convierta los bytes procedentes del teclado a los caracteres que espera *flujoE*. De esto se encarga *isr*:

```
InputStreamReader isr = new InputStreamReader(System.in);
```

La clase **InputStreamReader** establece un puente para pasar flujos de bytes a flujos de caracteres.

Finalmente, una sentencia como *str = Leer.dato()* permitirá leer de *flujoE* caracteres proporcionados por *isr* resultantes de la conversión de los bytes que éste obtiene del origen **System.in**, según ilustra la figura siguiente:



Para limpiar el flujo de entrada definido por la clase *Leer* añadiremos al método *limpiar* el código que se muestra a continuación. Lo único que hace este método es leer caracteres, uno a uno, mientras haya caracteres disponibles, ya que cada carácter leído es automáticamente eliminado del flujo de entrada.

```

public static void limpiar()
{
    int car = 0;
    try
    {
        while (flujoE.ready()) flujoE.read(); // limpiar flujoE
    }
    catch(IOException e)
    {
        System.err.println("Error: " + e.getMessage());
    }
}
  
```

El método *mirar* permitirá conocer cuál es el siguiente carácter que se puede leer del flujo de entrada. Para ello, este método leerá el primer carácter disponible en el flujo y a continuación lo devolverá al mismo para que esté disponible para una siguiente lectura. El método retornará ese carácter para que quien lo invoque pueda analizar cuál será el siguiente carácter que se leerá del flujo; si no hubiera ningún carácter esperará a que el usuario realice una entrada.

```

public static char mirar()
{
    int car = 0;

    try
    {
        car = flujoE.read();
        flujoE.unread(car);
    }
    catch(IOException e)
    {
        System.err.println("Error: " + e.getMessage());
    }
    return (char)car; // retornar el primer carácter disponible
}
  
```

El método *carácter* devolverá el siguiente carácter disponible en el flujo de entrada. Si no hubiera ningún carácter disponible, quedará a la espera de que se introduzca uno.

```
public static char carácter()
{
    int car = 0;
    try
    {
        car = flujoE.read();
    }
    catch(IOException e)
    {
        System.err.println("Error: " + e.getMessage());
    }
    return (char)car; // devolver el dato tecleado
}
```

El método *dato* ha sido modificado para que realice la misma función que realizaba en su versión anterior; esto es, leer una línea de texto que devolverá en un objeto de la clase **String**, entendiendo por línea de texto la cadena formada por los caracteres que hay hasta encontrar uno de los siguientes: '\r', '\n' o ambos; estos caracteres serán leídos pero no almacenados. Esta nueva versión es como consecuencia de que la clase **PushbackReader** no tiene el método **readLine** como ocurría con la clase **BufferedReader**.

```
public static String dato()
{
    StringBuffer sdato = new StringBuffer();
    int car = 0;

    try
    {
        // Leer. La entrada finaliza al pulsar la tecla Entrar
        while ((car = flujoE.read()) != '\r' && car != -1)
            sdato.append((char)car);
        limpiar();
    }
    catch(IOException e)
    {
        System.err.println("Error: " + e.getMessage());
    }

    if (car == -1) return null;
    return sdato.toString(); // devolver el dato tecleado
}
```

Es importante tomar buena nota de cómo la salida de un flujo se puede conectar a la entrada de otro, lo que permite disponer de nuevas capacidades. ¿Qué flujos pueden utilizarse de esta forma? Todos aquellos cuyo constructor tenga un parámetro que haga referencia a otro flujo.

A continuación se muestra un ejemplo que utiliza la nueva versión de la clase *Leer* que acabamos de implementar. Dicho ejemplo se limita a leer un valor de tipo **double** si el primer carácter de la entrada efectuada por el usuario que ejecuta la aplicación es un dígito o el signo menos; en otro caso lee un **String**.

```
public class Test
{
    public static void main(String[] args)
    {
        char car = 0, cero = (char)'0', nueve = (char)'9',menos = (char)'-';
        String s = null;
        double d = 0.0;

        System.out.print("dato: ");
        if ((car = Leer.mirar()) >= cero && car <= nueve || car == menos)
            d = Leer.datoDouble();
        else
            s = Leer.dato();

        if (s != null)
            System.out.println(s);
        else
            System.out.println(d);
    }
}
```

La primera vez que se utilice la clase *Leer* cuando se ejecute la aplicación, seará definido el flujo de entrada referenciado por su miembro *flujoE*, que estará disponible hasta que finalice dicha aplicación.

Una vez descrita la jerarquía de clases que Java proporciona para realizar la E/S, es el momento de plantearnos la utilización de esta jerarquía de clases.

ABRIENDO FICHEROS PARA ACCESO SECUENCIAL

El tipo de acceso más simple a un fichero de datos es el secuencial. Un fichero abierto para *acceso secuencial* es un fichero que puede almacenar registros de cualquier longitud, incluso de un sólo byte. Cuando la información se escribe registro a registro, éstos son colocados uno a continuación de otro, y cuando se lee, se empieza por el primer registro y se continúa al siguiente hasta alcanzar el final.

Este tipo de acceso generalmente se utiliza con ficheros de texto en los que se escribe toda la información desde el principio hasta el final y se lee de la misma forma. En cambio, los ficheros de texto no son los más apropiados para almacenar grandes series de números, porque cada número es almacenado como una secuencia de bytes; esto significa que un número entero de nueve dígitos ocupa nueve bytes en lugar de los cuatro requeridos para un entero. De ahí que a continuación se expongan distintos tipos de flujos: de bytes y de caracteres para el tratamiento de texto, y de datos para el tratamiento de números.

Flujos de bytes

Los datos pueden ser escritos o leídos de un fichero byte a byte utilizando flujos de las clases **FileOutputStream** y **InputStream**.

FileOutputStream

Un flujo de la clase **FileOutputStream** permite escribir bytes en un fichero. Además de los métodos que esta clase hereda de **OutputStream**, la clase proporciona los constructores siguientes:

```
FileOutputStream(String nombre)
FileOutputStream(String nombre, boolean añadir)
FileOutputStream(File fichero)
```

El primer constructor abre un flujo de salida hacia el fichero especificado por *nombre*, mientras que el segundo hace lo mismo, pero con la posibilidad de añadir datos a un fichero existente (*añadir* = **true**); el tercero lo hace a partir de un objeto **File**. Un ejemplo aclarará los conceptos expuestos.

El siguiente ejemplo es una aplicación Java que lee una línea de texto desde el teclado y la guarda en un fichero denominado *texto.txt*.

La aplicación definida por la clase *CEscribirBytes* mostrada a continuación, realiza lo siguiente:

1. Define una matriz *buffer* de 81 bytes.
2. Lee una línea de texto desde el teclado y la almacena en *buffer*.
3. Define un flujo *fs* hacia un fichero denominado *texto.txt*. Tenga presente que si el fichero existe, se borrará en el momento de definir el flujo que permite su acceso, excepto si especifica como segundo parámetro **true**.

```
FileOutputStream fs = new FileOutputStream("texto.txt");
```

4. Escribe explícitamente la línea de texto en el flujo (implícitamente la escribe en el fichero). Esto se hace cuando el flujo recibe el mensaje **write**, lo que origina que se ejecute el método **write**, en este caso con tres parámetros: el primero es una referencia a la matriz que contiene los bytes que deseamos escribir, el segundo es la posición en la matriz del primer byte que se desea escribir y el tercero, el número de bytes a escribir.

```
fs.write(buffer, 0, nbytes);
```

El programa completo se muestra a continuación:

```
import java.io.*;
public class CEscribirBytes
{
    public static void main (String[] args)
    {
        FileOutputStream fs = null;
        byte[] buffer = new byte[81];
        int nbytes;

        try
        {
            System.out.println(
                "Escriba el texto que desea almacenar en el fichero:");
            nbytes = System.in.read(buffer);
            fs = new FileOutputStream("texto.txt");
            fs.write(buffer, 0, nbytes);
        }
        catch(IOException e)
        {
            System.out.println("Error: " + e.toString());
        }
    }
}
```

Cuando ejecute la aplicación escriba una línea de texto y pulse la tecla *Entrar*. A continuación, en la línea de órdenes del sistema, teclee *type texto.txt* en Windows, o bien *cat texto.txt* en UNIX, para mostrar el texto del fichero y comprobar que todo ha funcionado como esperaba.

Si lo que desea es añadir información al fichero, cree el flujo hacia el mismo como se indica a continuación:

```
fs = new FileOutputStream("texto.txt", true);
```

En este caso, si el fichero no existe se crea y si existe, los datos que se escriban en él se añadirán al final.

Es una buena costumbre cerrar un flujo cuando ya no se vaya a utilizar más. Aplicando esta idea en la aplicación anterior, el código quedaría así:

```

try
{
    System.out.println(
        "Escriba el texto que desea almacenar en el fichero:");
    nbytes = System.in.read(buffer);
    fs = new FileOutputStream("texto.txt");
    fs.write(buffer, 0, nbytes);
}
catch(IOException e)
{
    System.out.println("Error: " + e.toString());
}
finally
{
    try
    {
        // Cerrar el fichero
        if (fs != null) fs.close();
    }
    catch(IOException e)
    {
        System.out.println("Error: " + e.toString());
    }
}

```

En la biblioteca de Java puede observar que el método **close** de **FileOutputStream** declara que puede lanzar una excepción de la clase **IOException**, razón por la que nuestro código debe atraparla. Quizás haya pensado invocar al método **close** después de haber ejecutado el método **write** dentro del primer bloque **try**:

```

try
{
    System.out.println(
        "Escriba el texto que desea almacenar en el fichero:");
    nbytes = System.in.read(buffer);
    fs = new FileOutputStream("texto.txt");
    fs.write(buffer, 0, nbytes);
    if (fs != null) fs.close();
}
// ...

```

Aunque esta forma de proceder también es válida no es tan eficiente como la anterior, porque ¿qué sucedería si el método **write** lanzara una excepción? No se ejecutaría **close**, aunque finalmente el sistema se encargaría de cerrar el flujo.

FileInputStream

Un flujo de la clase **FileInputStream** permite leer bytes desde un fichero. Además de los métodos que esta clase hereda de **InputStream**, la clase proporciona los constructores siguientes:

```
FileInputStream(String nombre)
FileInputStream(File fichero)
```

El primer constructor abre un flujo de entrada desde el fichero especificado por *nombre*, mientras que el segundo lo hace a partir de un objeto **File**. Un ejemplo aclarará los conceptos expuestos.

El siguiente ejemplo es una aplicación Java que lee el texto guardado en el fichero *texto.txt* creado por la aplicación anterior y lo almacena en una matriz denominada *buffer*.

La aplicación definida por la clase *CLeerBytes* mostrada a continuación, realiza lo siguiente:

1. Define una matriz *buffer* de 81 bytes.
2. Define un flujo *fe* desde un fichero denominado *texto.txt*. Tenga presente que si el fichero no existe, se lanzará una excepción indicándolo.

```
FileInputStream fe = new FileInputStream("texto.txt");
```

3. Lee el texto desde el flujo y lo almacena en *buffer*. Esto se hace cuando el flujo recibe el mensaje **read**, lo que origina que se ejecute el método **read**, en este caso con tres parámetros: el primero es una referencia a la matriz que almacenará los bytes leídos, el segundo es la posición en la matriz del primer byte que se desea almacenar y el tercero, el número máximo de bytes que se leerán. El método devuelve el número de bytes leídos o -1 si no hay más datos porque se ha alcanzado el final del fichero.

```
nbytes = fe.read(buffer, 0, 81);
```

4. Crea un objeto **String** con los datos leídos.

El programa completo se muestra a continuación:

```
import java.io.*;
public class CLeerBytes
{
```

```

public static void main (String[] args)
{
    FileInputStream fe = null;
    byte[] buffer = new byte[81];
    int nbytes;

    try
    {
        fe = new FileInputStream("texto.txt");
        nbytes = fe.read(buffer, 0, 81);
        String str = new String(buffer, 0, nbytes);
        System.out.println(str);
    }
    catch(IOException e)
    {
        System.out.println("Error: " + e.toString());
    }
    finally
    {
        try
        {
            // Cerrar el fichero
            if (fe != null) fe.close();
        }
        catch(IOException e)
        {
            System.out.println("Error: " + e.toString());
        }
    }
}
}

```

Clase File

El ejemplo anterior utiliza un **String** para referirse al fichero, pero también podría haber utilizado un objeto de la clase **File**. Un objeto de esta clase representa el nombre de un fichero o de un directorio que puede existir en el sistema de ficheros de la máquina; por lo tanto, sus métodos permitirán interrogar al sistema sobre todas las características de ese fichero o directorio. Además de los métodos a los que nos referimos, la clase proporciona los constructores siguientes:

```

public File(String ruta_completa)
public File(String ruta, String nombre)
public File(File ruta, String nombre)

```

El primer constructor crea un objeto **File** a partir de un nombre de fichero más su ruta de acceso (relativa o absoluta). Por ejemplo, el siguiente código crea un

objeto **File** a partir de la ruta relativa *proyecto2\texto.txt*. Observe que el separador de directorios viene especificado por la secuencia de escape ‘\’. Este separador en un sistema UNIX es ‘/’.

```
File fichero = new File("proyecto\\texto.txt");

System.out.println("Nombre del fichero: " + fichero.getName());
System.out.println("Directorio padre: " + fichero.getParent());
System.out.println("Ruta relativa: " + fichero.getPath());
System.out.println("Ruta absoluta: " +
                    fichero.getAbsolutePath());
```

Los resultados que se visualizan cuando se ejecute el código anterior serían análogos a los siguientes:

```
Nombre del fichero: texto.txt
Directorio padre: proyecto
Ruta relativa: proyecto\texto.txt
Ruta absoluta: C:\java\proyecto\texto.txt
```

El segundo constructor crea un objeto **File** a partir de una ruta (absoluta o relativa) y un nombre de fichero separado. Por ejemplo, el objeto *fichero* del ejemplo anterior podría definirse también así:

```
File fichero = new File("proyecto", "texto.txt");
```

El tercer constructor crea un objeto **File** a partir de otro que represente una ruta (absoluta o relativa) y un nombre de fichero separado. Por ejemplo, el objeto *fichero* del ejemplo anterior podría definirse también así:

```
File dir = new File("proyecto");
File fichero = new File(dir, "texto.txt");
```

La tabla siguiente resume los métodos de la clase **File**:

Método	Significado
getName	Devuelve el nombre del fichero especificado por el objeto File que recibe este mensaje.
getParent	Devuelve el directorio padre.
getPath	Devuelve la ruta relativa del fichero.
getAbsolutePath	Devuelve la ruta absoluta del fichero.
exists	Devuelve true si el nombre especificado por el objeto File que recibe este mensaje existe.
canWriter	Devuelve true si se puede escribir en el fichero o directorio especificado por el objeto File .

Método	Significado
canRead	Devuelve true si se puede leer desde el fichero o directorio especificado por el objeto File .
isFile	Devuelve true si se trata de un fichero válido.
isDirectory	Devuelve true si se trata de un directorio válido.
isHidden	Devuelve true si se trata de un fichero o directorio oculto.
length	Devuelve el tamaño del fichero (cuando se trate de un directorio, el valor devuelto es cero).
list	Devuelve una matriz de objetos String que almacena los nombres de los ficheros y directorios que hay en el directorio especificado por el objeto File .
mkdir	Crea el directorio especificado por el objeto File .
mkdirs	Crea el directorio especificado por el objeto File incluyendo los directorios que no existan en la ruta especificada.
delete	Borra el fichero o directorio especificado por el objeto File . Cuando se trate de un directorio, éste debe de estar vacío.
deleteOnExit	Igual que delete , pero cuando la máquina virtual termina.
createTempFile	Crea el fichero vacío especificado por los argumentos pasados, en el directorio temporal del sistema.
renameTo	Renombra el fichero especificado por el objeto File que recibe este mensaje, con el nombre especificado por el objeto File pasado como argumento.
setReadOnly	Marcar el fichero o directorio especificado por el objeto File de sólo lectura.
toString	Devuelve la ruta especificada cuando se creó el objeto File .

Para más detalles sobre los métodos anteriores recurra a la ayuda proporcionada con el JDK. Utilizando las capacidades de la clase **File** podemos modificar la aplicación *CleerBytes* para que solicite un nombre de un fichero existente:

```
// ...
String nombreFichero = null;
File fichero = null;

try
{
    do
    {
        System.out.print("Nombre del fichero: ");
        nbytes = System.in.read(buffer);
        nombreFichero = new String(buffer, 0, nbytes-2); // menos CR+LF
        fichero = new File(nombreFichero);
    }
    while (!fichero.exists());
```

```

fe = new FileInputStream(fichero);
nbytes = fe.read(buffer, 0, 81);
// ...
// ...

```

Análogamente, utilizando las capacidades de la clase **File** podemos modificar la aplicación *EscribirBytes* para que verifique si existe el fichero en el que se va a escribir los datos leídos desde el teclado:

```

// ...
String nombreFichero = null;
File fichero = null;
try
{
    System.out.print("Nombre del fichero: ");
    nbytes = System.in.read(buffer);
    nombreFichero = new String(buffer, 0, nbytes-2); // menos CR+LF
    fichero = new File(nombreFichero);

    char resp = 's';
    if (fichero.exists())
    {
        System.out.print("El fichero existe ¿desea sobreescribirlo? (s/n) ");
        resp = (char)System.in.read();
        // Saltar los bytes no leídos del flujo in
        System.in.skip(System.in.available());
    }

    if (resp == 's')
    {
        System.out.println(
            "Escriba el texto que desea almacenar en el fichero:");
        nbytes = System.in.read(buffer);
        fs = new FileOutputStream(fichero);
        fs.write(buffer, 0, nbytes);
    }
}
// ...

```

Flujos de caracteres

Una vez que sabemos trabajar con flujos de bytes, hacerlo con flujos de caracteres es prácticamente lo mismo. Esto nos será útil cuando necesitamos trabajar con texto representado por un conjunto de caracteres ASCII o Unicode. Las clases que definen estos flujos son subclases de **Reader**, como **FileWriter** y **FileReader**.

FileWriter

Un flujo de la clase **FileWriter** permite escribir caracteres (**char**) en un fichero. Además de los métodos que esta clase hereda de **Writer**, la clase proporciona los constructores siguientes:

```
FileWriter(String nombre)
FileWriter(String nombre, boolean añadir)
FileWriter(File fichero)
```

El primer constructor abre un flujo de salida hacia el fichero especificado por *nombre*, mientras que el segundo hace lo mismo, pero con la posibilidad de añadir datos a un fichero existente (*añadir = true*); el tercero lo hace a partir de un objeto **File**.

El siguiente ejemplo es la versión de la aplicación Java *CEscribirBytes* realizada anteriormente, adaptada para escribir caracteres en lugar de bytes. Observe que las variaciones son mínimas:

```
import java.io.*;

public class CEscribirCars
{
    public static void main (String[] args)
    {
        FileWriter fs = null;
        byte[] buffer = new byte[81];
        int nbytes;
        String nombreFichero = null;
        File fichero = null;

        try
        {
            System.out.print("Nombre del fichero: ");
            nbytes = System.in.read(buffer);
            nombreFichero = new String(buffer, 0, nbytes-2); // menos CR+LF
            fichero = new File(nombreFichero);

            char resp = 's';
            if (fichero.exists())
            {
                System.out.print("El fichero existe ¿desea sobreescribirlo? (s/n) ");
                resp = (char)System.in.read();
                // Saltar los bytes no leídos del flujo in
                System.in.skip(System.in.available());
            }
        }
```

```
if (resp == 's')
{
    System.out.println(
        "Escriba el texto que desea almacenar en el fichero:");
    nbytes = System.in.read(buffer);
    String str = new String(buffer, 0, nbytes);
    fs = new FileWriter(fichero);
    fs.write(str, 0, str.length());
}
catch(IOException e)
{
    System.out.println("Error: " + e.toString());
}
finally
{
    try
    {
        // Cerrar el fichero
        if (fs != null) fs.close();
    }
    catch(IOException e)
    {
        System.out.println("Error: " + e.toString());
    }
}
```

FileReader

Un flujo de la clase **FileReader** permite leer caracteres desde un fichero. Además de los métodos que esta clase hereda de **Reader**, la clase proporciona los constructores siguientes:

```
FileReader(String nombre)  
FileReader(File fichero)
```

El primer constructor abre un flujo de entrada desde el fichero especificado por *nombre*, mientras que el segundo lo hace a partir de un objeto **File**.

El siguiente ejemplo es la versión de la aplicación Java *CLeerBytes* realizada anteriormente, adaptada para leer caracteres en lugar de bytes. Observe que las variaciones son mínimas:

```
import java.io.*;
```

```

public class CLeerCars
{
    public static void main (String[] args)
    {
        byte[] nomFich = new byte[81];
        String nombreFichero = null;
        File fichero = null;
        int nbytes, ncars;
        FileReader fe = null;
        char[] buffer = new char[81];

        try
        {
            do
            {
                System.out.print("Nombre del fichero: ");
                nbytes = System.in.read(nomFich);
                nombreFichero = new String(nomFich, 0, nbytes-2); // menos CR+LF
                fichero = new File(nombreFichero);
            }
            while (!fichero.exists());

            fe = new FileReader(fichero);
            ncars = fe.read(buffer, 0, 81);
            System.out.println(buffer);
        }
        catch(IOException e)
        {
            System.out.println("Error: " + e.toString());
        }
        finally
        {
            try
            {
                // Cerrar el fichero
                if (fe != null) fe.close();
            }
            catch(IOException e)
            {
                System.out.println("Error: " + e.toString());
            }
        }
    }
}

```

Flujos de datos

Seguramente, en alguna ocasión desearemos escribir en un fichero datos de tipos primitivos (**boolean**, **byte**, **double**, **float**, **long**, **int** y **short**) para posteriormente

recuperarlos como tal. Para estos casos, el paquete **java.io** proporciona las clases **DataInputStream** y **DataOutputStream**, las cuales permiten leer y escribir, respectivamente, datos de cualquier tipo primitivo. Entonces, ¿por qué no se han analizado previamente? Pues, simplemente porque no pueden utilizarse con los dispositivos ASCII de E/S estándar. Un flujo **DataInputStream** sólo puede leer datos almacenados en un fichero a través de un flujo **DataOutputStream**.

Observe que los flujos de estas clases actúan como filtros; esto es, los datos obtenidos del origen o enviados al destino son transformados mediante alguna operación; en este caso, sufren una conversión a un formato portable (UTF-8: Unicode ligeramente modificado) cuando son almacenados y viceversa cuando son recuperados. El procedimiento para utilizar un filtro es básicamente así:

- Se crea un flujo asociado con un origen o destino de los datos.
- Se asocia un filtro con el flujo anterior.
- Finalmente, el programa leerá o escribirá datos a través de ese filtro.

DataOutputStream

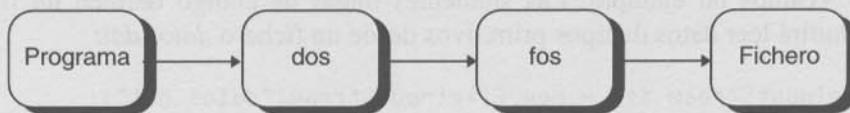
Un flujo de la clase **DataOutputStream**, derivada indirectamente de **OutputStream**, permite a una aplicación escribir en un flujo de salida subordinado, datos de cualquier tipo primitivo.

Todos los métodos proporcionados por esta clase están definidos en la interfaz **DataOutput** implementada por la misma.

Veamos un ejemplo. Las siguientes líneas de código definen un filtro que permitirá escribir datos de tipos primitivos en un fichero *datos.dat*:

```
FileOutputStream fos = new FileOutputStream("datos.dat");
DataOutputStream dos = new DataOutputStream(fos);
```

Un programa que quiera almacenar datos en el fichero *datos.dat*, escribirá tales datos en el filtro *dos*, que a su vez está conectado al flujo *fos* abierto hacia ese fichero. La figura siguiente muestra de forma gráfica lo expuesto:



El siguiente fragmento de código muestra cómo utilizar el filtro anterior para almacenar los datos *nombre*, *dirección* y *teléfono* en un fichero especificado por *nombreFichero*:

```

FileOutputStream fos = new FileOutputStream(nombreFichero);
DataOutputStream dos = new DataOutputStream(fos);
// Almacenar el nombre la dirección y el teléfono en el fichero
dos.writeUTF("un nombre");
dos.writeUTF("una dirección");
dos.writeLong(942334455);

dos.close(); fos.close();

```

Los métodos más utilizados de esta clase se resumen en la tabla siguiente:

Método	Descripción
writeBoolean	Escribe un valor de tipo boolean .
writeByte	Escribe un valor de tipo byte .
writeBytes	Escribe un String como una secuencia de bytes.
writeChar	Escribe un valor de tipo char .
writeChars	Escribe un String como una secuencia de caracteres.
writeShort	Escribe un valor de tipo short .
writeInt	Escribe un valor de tipo int .
writeLong	Escribe un valor de tipo long .
writeFloat	Escribe un valor de tipo float .
writeDouble	Escribe un valor de tipo double .
writeUTF	Escribe una cadena de caracteres en formato UTF-8; los dos primeros bytes especifican el número de bytes de datos escritos a continuación.

DataInputStream

Un flujo de la clase **DataInputStream**, derivada indirectamente de **InputStream**, permite a una aplicación leer de un flujo de entrada subordinado, datos de cualquier tipo primitivo escritos por un flujo de la clase **DataOutputStream**.

Todos los métodos proporcionados por esta clase están definidos en la interfaz **DataInput** implementada por la misma.

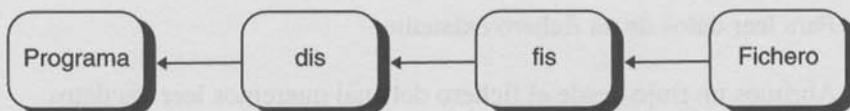
Veamos un ejemplo. Las siguientes líneas de código definen un filtro que permitirá leer datos de tipos primitivos desde un fichero *datos.dat*:

```

FileInputStream fis = new FileInputStream("datos.dat");
DataInputStream dis = new DataInputStream(fis);

```

Un programa que quiera almacenar datos en el fichero *datos.dat*, escribirá tales datos en el filtro *dis*, que a su vez está conectado al flujo *fis* abierto desde ese fichero. La figura siguiente muestra de forma gráfica lo expuesto:



El siguiente fragmento de código muestra cómo utilizar el filtro anterior para leer los datos *nombre*, *dirección* y *teléfono* desde un fichero especificado por *nombreFichero*:

```

FileInputStream fis = new FileInputStream(nombreFichero);
DataInputStream dis = new DataInputStream(fis);
// Leer el nombre la dirección y el teléfono del fichero
nombre = dis.readUTF();
dirección = dis.readUTF();
teléfono = dis.readLong();

dis.close(); fis.close();
  
```

Los métodos más utilizados de esta clase se resumen en la tabla siguiente:

Método	Descripción
readBoolean	Devuelve un valor de tipo boolean .
readByte	Devuelve un valor de tipo byte .
readShort	Devuelve un valor de tipo short .
readChar	Devuelve un valor de tipo char .
readInt	Devuelve un valor de tipo int .
readLong	Devuelve un valor de tipo long .
readFloat	Devuelve un valor de tipo float .
readDouble	Devuelve un valor de tipo double .
readUTF	Devuelve una cadena de caracteres en formato UTF-8; los dos primeros bytes especifican el número de bytes de datos que serán leídos a continuación.

Un ejemplo de acceso secuencial

Después de la teoría expuesta hasta ahora acerca del trabajo con ficheros, habrá observado que la metodología de trabajo se repite. Es decir, para escribir datos en un fichero:

- Definimos un flujo hacia el fichero en el que deseamos escribir datos.
- Leemos los datos del dispositivo de entrada o de otro fichero y los escribimos en nuestro fichero. Este proceso se hace normalmente registro a registro. Para ello, utilizaremos los métodos proporcionados por la interfaz del flujo.
- Cerramos el flujo.

Para leer datos de un fichero existente:

- Abrimos un flujo desde el fichero del cual queremos leer los datos.
- Leemos los datos del fichero y los almacenamos en variables de nuestro programa con el fin de trabajar con ellos. Este proceso se hace normalmente registro a registro. Para ello, utilizaremos los métodos proporcionados por la interfaz del flujo.
- Cerramos el flujo.

Esto pone de manifiesto que un fichero no es más que un medio permanente de almacenamiento de datos, dejando esos datos disponibles para cualquier programa que necesite manipularlos. Lógicamente, los datos serán recuperados del fichero con el mismo formato con el que fueron escritos, de lo contrario los resultados serán inesperados. Es decir, si en el ejercicio siguiente los datos son guardados en el orden: una cadena, otra cadena y un **long**, tendrán que ser recuperados en este orden y con este mismo formato. Sería un error recuperar primero un **long** después una cadena y finalmente la otra cadena, o recuperar primero una cadena, después un **float** y finalmente la otra cadena; etc.

El siguiente ejemplo lee de la entrada estándar grupos de datos (registros) definidos de la forma que se indica a continuación y los almacena en un fichero.

```
String nombre, dirección;
long teléfono;
```

Para realizar este ejemplo, escribiremos una clase aplicación *CrearListaTfnos* con dos métodos: *crearFichero* y *main*.

El método *crearFichero* recibe como parámetro un objeto **File** que define el nombre del fichero que se desea crear y realiza las tareas siguientes:

- Crea un flujo hacia el fichero especificado por el objeto **File** que permite escribir datos de tipos primitivos utilizando un buffer.
- Lee grupos de datos *nombre*, *dirección* y *teléfono* de la entrada estándar y los escribe en el fichero.
- Si durante su ejecución alguno de los métodos invocados lanza una excepción, la vuelve a lanzar para que sea atrapada por el método que le invocó.

El método **main** realiza las tareas siguientes:

- Crea un objeto **File** a partir del nombre del fichero leído desde la entrada estándar.
- Verifica si el fichero existe.

- Si no existe, o bien si existe y se desea sobreescribir, invoca al método *crearFichero* pasando como argumento el objeto **File** creado.

```

import java.io.*;

// Se utiliza también la clase Leer modificada en este capítulo

public class CrearListaTfnos
{
    public static void crearFichero(File fichero)
        throws IOException
    {
        PrintStream flujoS = System.out; // salida estándar
        DataOutputStream dos = null; // salida de datos hacia el fichero
        char resp;

        try
        {
            // Crear un flujo hacia el fichero que permita escribir
            // datos de tipos primitivos y que utilice un buffer.
            dos = new DataOutputStream(new BufferedOutputStream(
                new FileOutputStream(fichero)));
        }

        // Declarar los datos a escribir en el fichero
        String nombre, dirección;
        long teléfono;

        // Leer datos de la entrada estándar y escribirlos
        // en el fichero
        do
        {
            flujoS.print("nombre:    "); nombre = Leer.dato();
            flujoS.print("dirección: "); dirección = Leer.dato();
            flujoS.print("teléfono:  "); teléfono = Leer.datoLong();

            // Almacenar un nombre, una dirección y un teléfono en
            // el fichero
            dos.writeUTF(nombre);
            dos.writeUTF(dirección);
            dos.writeLong(teléfono);

            flujoS.print("¿desea escribir otro registro? (s/n) ");
            resp = Leer.carácter();
            Leer.limpiar();
        }
        while (resp == 's');
    }
    finally
    {
        // Cerrar el flujo
    }
}

```

```
    if (dos != null) dos.close();
}

public static void main(String[] args)
{
    PrintStream flujoS = System.out; // salida est ndar
    String nombreFichero = null;      // nombre del fichero
    File fichero = null; // objeto que identifica el fichero

    try
    {
        // Crear un objeto File que identifique al fichero
        flujoS.print("Nombre del fichero: ");
        nombreFichero = Leer.dato();
        fichero = new File(nombreFichero);

        // Verificar si el fichero existe
        char resp = 's';
        if (fichero.exists())
        {
            flujoS.print("El fichero existe ?desea sobreescribirlo? (s/n) ");
            resp = Leer.caracter();
            Leer.limpiar();
        }
        if (resp == 's')
        {
            crearFichero(fichero);
        }
    }
    catch(IOException e)
    {
        flujoS.println("Error: " + e.getMessage());
    }
}
```

Para leer el fichero creado por la aplicación anterior, vamos a escribir otra basada en la clase *MostrarListaTfnos*. Esta clase define dos métodos: *mostrarFichero* y **main**.

El método `mostrarFichero` recibe como parámetro un objeto `String` que almacena el nombre del fichero que se desea leer y realiza las tareas siguientes:

- Crea un objeto **File** para identificar al fichero.
 - Si el fichero especificado existe, crea un flujo desde el mismo que permite leer datos de tipos primitivos utilizando un *buffer*.

- Lee un grupo de datos *nombre*, *dirección* y *teléfono* desde el fichero y los muestra. Cuando se alcance el final del fichero Java lanzará una excepción del tipo **EOFException**, instante en el que finalizará la ejecución de este método.
- Si durante su ejecución alguno de los métodos invocados lanza una excepción **IOException**, la pasa para que sea atrapada por el método que le invocó.

El método **main** recibe como parámetro el nombre del fichero que se desea crear y realiza las tareas siguientes:

- Verifica si se pasó un argumento cuando se ejecutó la aplicación con el nombre del fichero cuyo contenido se desea visualizar.
- Si no se pasó un argumento, la aplicación mostrará un mensaje indicando la sintaxis que se debe de emplear para ejecutar la misma y finalizará. En otro caso, invoca al método *mostrarFichero* pasando como argumento *args[0]*.

```
import java.io.*;

public class MostrarListaTfnos
{
    public static void mostrarFichero(String nombreFichero)
        throws IOException
    {
        PrintStream flujoS = System.out; // salida estándar
        DataInputStream dis = null; // entrada de datos desde el fichero
        File fichero = null; // objeto que identifica el fichero

        try
        {
            // Crear un objeto File que identifique al fichero
            fichero = new File(nombreFichero);

            // Verificar si el fichero existe
            if (fichero.exists())
            {
                // Si existe, abrir un flujo desde el mismo
                dis = new DataInputStream(new BufferedInputStream(
                    new FileInputStream(fichero)));

                // Declarar los datos a leer desde el fichero
                String nombre, dirección;
                long teléfono;
                do
                {
                    // Leer un nombre, una dirección y un teléfono desde el
                    // fichero. Cuando se alcance el final del fichero Java
                    // lanzará una excepción del tipo EOFException.
                    nombre = dis.readUTF();
                    dirección = dis.readUTF();
```

```

        teléfono = dis.readLong();

        // Mostrar los datos nombre, dirección y teléfono
        flujoS.println(nombre);
        flujoS.println(dirección);
        flujoS.println(teléfono);
        flujoS.println();
    }
    while (true);
}
else
    flujoS.println("El fichero no existe");
}
catch(EOFException e)
{
    flujoS.println("Fin del listado");
}
finally
{
    // Cerrar el flujo
    if (dis != null) dis.close();
}
}

public static void main(String[] args)
{
    if (args.length != 1)
        System.err.println("Sintaxis: java MostrarListaTfnos " +
                           "<fichero fuente>");
    else
    {
        try
        {
            mostrarFichero(args[0]);
        }
        catch(IOException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

SERIACIÓN DE OBJETOS

En el apartado anterior hemos aprendido cómo escribir y leer grupos de datos a y desde un fichero. Pero en un desarrollo orientado a objetos debemos pensar en objetos; por lo tanto, ese grupo de datos al que nos hemos referido no lo trataremos.

mos aisladamente; más bien se corresponderá con los atributos de un objeto, por ejemplo de la clase *CPersona*, lo que nos conducirá a escribir y leer objetos a y desde un fichero.

Normalmente la operación de enviar una serie de objetos a un fichero en disco para hacerlos persistentes recibe el nombre de *seriación*, y la operación de leer o recuperar su estado del fichero para reconstruirlos en memoria recibe el nombre de *deseriación*. Para realizar estas operaciones de una forma automática, el paquete **java.io** proporciona las clases **ObjectOutputStream** y **ObjectInputStream**. Ambas clases dan lugar a flujos que procesan sus datos; en este caso, se trata de convertir el estado de un objeto (los atributos excepto las variables estáticas), incluyendo la clase del objeto y el prototipo de la misma, en una secuencia de bytes y viceversa. Por esta razón los flujos **ObjectOutputStream** y **ObjectInputStream** deben ser construidos sobre otros flujos que canalicen esos bytes a y desde el fichero. El esquema gráfico que responde a este proceso es el siguiente:



Para poder seriar los objetos de una clase, ésta debe de implementar la interfaz **Serializable**. Se trata de una interfaz vacía; esto es, sin ningún método; su propósito es simplemente identificar clases cuyos objetos se pueden seriar.

El siguiente ejemplo, define la clase *CPersona* como una clase cuyos objetos se pueden seriар:

```

import java.io.*;

public class CPersona implements Serializable
{
    // Cuerpo de la clase
}

```

Como la interfaz **Serializable** está vacía no hay que escribir ningún método extra en la clase.

Escribir objetos en un fichero

Un flujo de la clase **ObjectOutputStream** permite enviar datos de tipos primitivos y objetos hacia un flujo **OutputStream** o derivado; concretamente, cuando se trate de almacenarlos en un fichero, utilizaremos un flujo **FileOutputStream**. Posteriormente, esos objetos podrán ser reconstruidos a través de un flujo **ObjectInputStream**.

Para escribir un objeto en un flujo **ObjectOutputStream** utilizaremos el método **writeObject**. Los objetos pueden incluir *Strings* y matrices, y el almacenamiento de los mismos puede combinarse con datos de tipos primitivos, ya que esta clase implementa la interfaz **DataOutput**. Este método lanzará la excepción **NotSerializableException** si se intenta escribir un objeto de una clase que no implementa la interfaz **Serializable**.

Por ejemplo, el siguiente código construye un **ObjectOutputStream** sobre un **FileOutputStream**, y lo utiliza para almacenar un **String** y un objeto **CPersona** en un fichero denominado *datos*:

```
FileOutputStream fos = new FileOutputStream("datos");
ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeUTF("Fichero datos");
oos.writeObject(new CPersona(nombre, dirección, teléfono));
oos.close();
```

Como ejercicio, vamos a modificar la aplicación *CrearListaTfnos* anterior para que permita almacenar objetos **CPersona** en un fichero (la clase **CPersona** fue implementada en el capítulo 9 al hablar de matrices de objetos):

```
import java.io.*;
// Se utiliza también la clase Leer, modificada en este capítulo

public class CrearListaTfnos
{
    public static void crearFichero(File fichero)
        throws IOException
    {
        PrintStream flujoS = System.out; // salida estándar
        ObjectOutputStream oos = null; // salida de datos hacia el fichero
        char resp;
```

```

try
{
    // Crear un flujo hacia el fichero que permita escribir
    // objetos y datos de tipos primitivos.
    oos = new ObjectOutputStream(new FileOutputStream(fichero));
    // Declarar los datos a escribir en el fichero
    String nombre, dirección;
    long teléfono;

    // Leer datos de la entrada estándar y escribirlos
    // en el fichero
    do
    {
        flujoS.print("nombre:    "); nombre = Leer.dato();
        flujoS.print("dirección: "); dirección = Leer.dato();
        flujoS.print("teléfono:  "); teléfono = Leer.datoLong();

        // Crear un objeto CPersona y almacenarlo en el fichero
        oos.writeObject(new CPersona(nombre, dirección, teléfono));
        // ...
    }
    while (resp == 's');
    // ...
}

```

Leer objetos desde un fichero

Un flujo de la clase **ObjectInputStream** permite recuperar datos de tipos primitivos y objetos desde un flujo **InputStream** o derivado; concretamente, cuando se trate de datos de tipos primitivos y objetos almacenados en un fichero, utilizaremos un flujo **FileInputStream**. La clase **ObjectInputStream** implementa la interfaz **DataInput** para permitir leer también datos de tipos primitivos.

Para leer un objeto desde un flujo **ObjectInputStream** utilizaremos el método **readObject**. Si se almacenaron objetos y datos de tipos primitivos, deben ser recuperados en el mismo orden.

Por ejemplo, el siguiente código construye un **ObjectInputStream** sobre un **FileInputStream**, y lo utiliza para recuperar un **String** y un objeto **CPersona** de un fichero denominado *datos*:

```

FileInputStream fos = new FileInputStream("datos");
ObjectInputStream ois = new ObjectInputStream(fos);

String str = (String)ois.readUTF();
CPersona persona = (CPersona)ois.readObject();
ois.close();

```

Como ejercicio, vamos a modificar la aplicación *MostrarListaTfnos* anterior para que permita recuperar objetos *CPersona* desde un fichero (la clase *CPersona* fue implementada en el capítulo 9 al hablar de matrices de objetos):

```
import java.io.*;

public class MostrarListaTfnos
{
    public static void mostrarFichero(String nombreFichero)
        throws IOException
    {
        PrintStream flujoS = System.out; // salida estándar
        ObjectInputStream ois = null; // entrada de datos desde el fichero
        File fichero = null; // objeto que identifica el fichero

        try
        {
            // Crear un objeto File que identifique al fichero
            fichero = new File(nombreFichero);

            // Verificar si el fichero existe.
            if (fichero.exists())
            {
                // Si existe, abrir un flujo desde el mismo
                ois = new ObjectInputStream(new FileInputStream(fichero));
                // Declarar los datos a leer desde el fichero
                CPersona persona;
                String nombre, dirección;
                long teléfono;
                do
                {
                    // Leer un objeto CPersona desde el fichero. Cuando se
                    // alcance el final del fichero Java lanzará una
                    // excepción del tipo EOFException.
                    persona = (CPersona)ois.readObject();
                    // ...
                } while (true);
            }
            else
                flujoS.println("El fichero no existe");
        }
        catch(EOFException e)
        {
            flujoS.println("Fin del listado");
        }
        catch(ClassNotFoundException e)
        {
            flujoS.println("Error: " + e.getMessage());
        }
    }
}
```

```

    finally
    {
        // Cerrar el flujo
        if (ois != null) ois.close();
    }
}

public static void main(String[] args)
{
    // ...
}
}

```

Seriar objetos que referencian a objetos

Cuando en un fichero se escribe un objeto que hace referencia a otros objetos, entonces todos los objetos accesibles desde el primero deben ser escritos en el mismo proceso para mantener así la relación existente entre todos ellos. Este proceso es llevado a cabo automáticamente por el método **writeObject**, que escribe el objeto especificado, recorriendo sus referencias a otros objetos recursivamente, escribiendo así todos ellos.

Análogamente, si el objeto recuperado del flujo por el método **readObject** hace referencia a otros objetos, **readObject** recorrerá sus referencias a otros objetos recursivamente, para recuperar todos ellos manteniendo la relación que existía entre ellos cuando fueron escritos.

Veamos un ejemplo donde se aplique lo expuesto. Si recuerda, en el capítulo 9 implementamos una clase *CListaTfnos* para mantener una lista de teléfonos; y después en el 11, la mejoramos añadiendo el tratamiento de algunas excepciones. Un objeto de esta clase representaba una lista de teléfonos; la lista está implementada como una matriz de referencias a objetos *CPersona*, y cada objeto *CPersona* tiene como atributos el nombre, la dirección y el teléfono de un miembro de la lista. Después escribimos una clase aplicación *Test*, que utilizando las clases *CListaTfnos* y *CPersona* permitía buscar, añadir y eliminar teléfonos en una lista. Para que dicha aplicación fuera útil sólo le faltaba un detalle muy importante, que la lista creada fuera persistente. Esto implica guardar la lista en un fichero cuando la aplicación finalice, y recuperarla del fichero cuando la aplicación se inicie. Realicemos pues, una tercera versión que incluya de forma automática las operaciones de guardar y recuperar el objeto *CListaTfnos* en un fichero denominado “listatfnos.dat”.

Según lo explicado, para poder seriar un objeto de la clase *CListaTfnos*, ésta debe implementar la interfaz **Serializable**:

```
import java.io.*;
public class CListaTfnos implements Serializable
{
    // ...
}
```

Pero seriar un objeto *CListaTfnos* implica seriар los objetos *CPersona* referenciados por el primero. Por lo tanto, esta segunda clase también tiene que implementar la interfaz **Serializable**:

```
import java.io.*;
public class CPersona implements Serializable
{
    // ...
}
```

Después de realizar estas modificaciones, sólo queda cambiar la clase aplicación *Test* para que guarde la lista, sólo si ha sido modificada, en un fichero denominado "listatfnos.dat" cuando la aplicación finalice, y la recupere cuando la aplicación se inicie. A continuación se muestra el código completo de la clase *Test*, en el que se resaltan los añadidos más importantes que se han realizado:

```
import java.io.*;
// Aplicación lista de teléfonos. Cuando la aplicación finaliza
// la lista es salvada en un fichero "listatfnos.dat" y cuando
// se inicia se recupera de ese fichero.
//
public class Test
{
    public static int menú()
    {
        System.out.print("\n\n");
        System.out.println("1. Buscar");
        System.out.println("2. Buscar siguiente");
        System.out.println("3. Añadir");
        System.out.println("4. Eliminar");
        System.out.println("5. Salir");
        System.out.println();
        System.out.print("    Opción: ");
        int op;
        do
            op = Leer.datoInt();
        while (op < 1 || op > 5);
        return op;
    }

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

```

// Definir un flujo de caracteres de entrada: flujoE
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader flujoE = new BufferedReader(isr);
// Definir una referencia al flujo estándar de salida: flujoS
PrintStream flujoS = System.out;

CListaTfnos listatfnos;
int opción = 0, pos = -1;
String cadenabuscar = null;
String nombre, dirección;
long teléfono;
boolean eliminado = false;
boolean listaModificada = false;

try
{
    // Crear un objeto lista de teléfonos vacío (con 0 elementos)
    // o con el contenido del fichero listatfnos.dat si existe.
    File fichero = new File("listatfnos.dat");
    if (!fichero.exists())
    {
        listatfnos = new CListaTfnos();
        flujoS.println("Se ha creado una lista de teléfonos nueva");
    }
    else
    {
        ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream("listatfnos.dat"));
        listatfnos = (CListaTfnos)ois.readObject();
        ois.close();
        flujoS.println("Se cargó la lista de teléfonos con éxito");
    }
}

do
{
    opción = menú();

    switch (opción)
    {
        case 1: // buscar
            flujoS.print("conjunto de caracteres a buscar ");
            cadenabuscar = flujoE.readLine();
            pos = listatfnos.buscar(cadenabuscar, 0);
            if (pos == -1)
                if (listatfnos.longitud() != 0)
                    flujoS.println("búsqueda fallida");
                else
                    flujoS.println("lista vacía");
            else
    }
}

```

```

        flujoS.println(listatfnos.valorEn(pos).obtenerNombre());
        flujoS.println(listatfnos.valorEn(pos).obtenerDirección());
        flujoS.println(listatfnos.valorEn(pos).obtenerTeléfono());
    }
    break;
case 2: // buscar siguiente
    pos = listatfnos.buscar(cadenabuscar, pos + 1);
    if (pos == -1)
        if (listatfnos.longitud() != 0)
            flujoS.println("búsqueda fallida");
        else
            flujoS.println("lista vacía");
    else
    {
        flujoS.println(listatfnos.valorEn(pos).obtenerNombre());
        flujoS.println(listatfnos.valorEn(pos).obtenerDirección());
        flujoS.println(listatfnos.valorEn(pos).obtenerTeléfono());
    }
    break;
case 3: // añadir
    flujoS.print("nombre: "); nombre = flujoE.readLine();
    flujoS.print("dirección: "); dirección = flujoE.readLine();
    flujoS.print("teléfono: "); teléfono = Leer.datoLong();
    listatfnos.añadir(new CPersona(nombre, dirección, teléfono));
    listaModificada = true;
    break;
case 4: // eliminar
    flujoS.print("teléfono: "); teléfono = Leer.datoLong();
    eliminado = listatfnos.eliminar(teléfono);
    if (eliminado)
    {
        flujoS.println("registro eliminado");
        listaModificada = true;
    }
    else
        if (listatfnos.longitud() != 0)
            flujoS.println("teléfono no encontrado");
        else
            flujoS.println("lista vacía");
    break;
case 5: // salir
    // guardar lista
    if (listaModificada)
    {
        ObjectOutputStream ous = new ObjectOutputStream(
                new FileOutputStream("listatfnos.dat"));
        ous.writeObject(listatfnos);
        ous.close();
    }
}

```

```
        listatfnos = null;
    }
}
while(opción != 5);
}
catch (IOException e)
{
    System.out.println("Error: " + e.getMessage());
}
catch (ClassNotFoundException e)
{
    System.out.println("Error: " + e.getMessage());
}
}
```

ABRIENDO FICHEROS PARA ACCESO ALEATORIO

Hasta este punto, hemos trabajado con ficheros de acuerdo con el siguiente esquema: abrir el fichero, leer o escribir hasta el final del mismo, y cerrar el fichero. Pero no hemos leído o escrito a partir de una determinada posición dentro del fichero. Esto es particularmente importante cuando necesitamos modificar algunos de los valores contenidos en el fichero o cuando necesitemos extraer una parte concreta dentro del fichero.

El paquete **java.io** contiene la clase **RandomAccessFile** la que proporciona las capacidades que permiten este tipo de acceso directo. Además, un flujo de esta clase permite realizar tanto operaciones de lectura como de escritura sobre el fichero vinculado con el mismo. Esta clase se deriva directamente de **Object**, e implementa las interfaces **DataInput** y **DataOutput**.

Un fichero accedido aleatoriamente es comparable a una matriz. En una matriz para acceder a uno de sus elementos utilizamos un índice. En un fichero accedido aleatoriamente el índice es sustituido por un puntero de lectura o escritura (L/E). Dicho puntero es situado automáticamente al principio del fichero cuando éste se abre para leer y/o escribir. Por lo tanto, una operación de lectura o de escritura comienza en la posición donde esté el puntero dentro del fichero; finalmente, su posición coincidirá justo a continuación del último byte leído o escrito.

La clase RandomAccessFile

Un flujo de esta clase permite acceder directamente a cualquier posición dentro del fichero vinculado con él.

La clase **RandomAccessFile** proporciona dos constructores:

```
RandomAccessFile(String nombre-fichero, String modo)
RandomAccessFile(File objeto-File, String modo)
```

El primer constructor abre un flujo vinculado con el fichero especificado por *nombre-fichero*, mientras que el segundo hace lo mismo, pero a partir de un objeto **File**. El argumento *modo* puede ser:

Modo	Significado
r	<i>read</i> . Sólo se permiten realizar operaciones de lectura.
rw	<i>read/write</i> . Se pueden realizar operaciones de lectura y de escritura sobre el fichero.

Por ejemplo, el siguiente fragmento de código construye un objeto **File** para verificar si el nombre especificado para el fichero existe como tal. Si existe y no corresponde a un fichero se lanza una excepción; si existe y se trata de un fichero, se crea un flujo para escribir y leer a y desde ese fichero; y si no existe, también se crea el flujo y el fichero.

```
File fichero = new File("listatfnos.dat");
if (fichero.exists() && !fichero.isFile())
    throw new IOException(fichero.getName() + " no es un fichero");
RandomAccessFile listaTeléfonos = new RandomAccessFile(fichero, "rw");
```

Asimismo, la clase **RandomAccessFile** provee, además de los métodos de las interfaces **DataInput** y **DataOutput** (vea el apartado “flujos de datos” expuesto anteriormente), los métodos **getFilePointer**, **length** y **seek** que se definen de la forma siguiente:

```
public long getFilePointer() throws IOException
```

Este método devuelve la posición actual en bytes del puntero de L/E en el fichero. Piense en el puntero de L/E análogamente a como lo hace cuando piensa en el índice de una matriz. Este puntero marca siempre la posición donde se iniciará la siguiente operación de lectura o de escritura en el fichero.

```
public long length() throws IOException
```

Este otro método devuelve la longitud del fichero en bytes.

```
public void seek(long pos) throws IOException
```

Y este otro método, mueve el puntero de L/E a una nueva localización desplazada *pos* bytes del principio del fichero. No se permiten desplazamientos ne-

gativos. El desplazamiento requerido puede ir más allá del final del fichero; esta acción no cambia la longitud del fichero; la longitud del fichero sólo cambiará si a continuación, realizamos una operación de escritura.

Según lo expuesto, las dos líneas de código siguientes sitúan el puntero de L/E, la primera *desp* bytes antes del final del fichero y la segunda *desp* bytes después de la posición actual.

```
listaTeléfonos.seek(listaTeléfonos.length() - desp);
listaTeléfonos.seek(listaTeléfonos.getFilePointer() + desp);
```

Con esta clase no tenemos posibilidad de serializar objetos. Los datos deben guardarse uno a uno utilizando el método adecuado de la clase según su tipo. Por ejemplo, las siguientes líneas de código escriben en el fichero "datos" a partir de la posición *desp*, los atributos *nombre*, *dirección* y *teléfono* relativos a un objeto *CPersona*:

```
CPersona objeto;
// ...
RandomAccessFile fes = new RandomAccessFile("datos", "rw");
fes.seek(desp);
fes.writeUTF(objeto.obtenerNombre());
fes.writeUTF(objeto.obtenerDirección());
fes.writeLong(objeto.obtenerTeléfono());
```

Si para nuestros propósitos, pensamos en los atributos *nombre*, *dirección* y *teléfono* como si de un registro se tratara, ¿cuál es el tamaño en bytes de ese registro? Si escribimos más registros ¿todos tienen el mismo tamaño? Evidentemente no; el tamaño de cada registro dependerá del número de caracteres almacenados en los **String** *nombre* y *dirección* (*teléfono* es un dato de tamaño fijo, 8 bytes, puesto que se trata de un **long**) ¿A cuento de qué viene esta exposición?

Al principio de este apartado dijimos que el acceso aleatorio a ficheros es particularmente importante cuando necesitemos modificar algunos de los valores contenidos en el fichero, o bien cuando necesitemos extraer una parte concreta dentro del fichero. Esto puede resultar bastante complicado si las unidades de grabación que hemos denominado registros no son todas iguales, ya que intervienen los factores de: posición donde comienza un registro y longitud del registro. Tenga presente que cuando necesite reemplazar el registro *n* de un fichero por otro, no debe sobrepasarse el número de bytes que actualmente tiene. Todo esto es viable llevando la cuenta en una matriz de la posición de inicio de cada uno de los registros y de cada uno de los campos si fuera preciso (esta información se almacenaría en un fichero índice para su utilización posterior), pero resulta mucho más fácil si todos los registros tienen la misma longitud.

Como ejemplo, vamos a escribir otra versión de la aplicación “lista de teléfonos” desarrollada en el capítulo 9 y modificada en el 11 y en éste. Esta nueva versión básicamente sustituirá la matriz de objetos *CPersona* encapsulada en *CListaTfnos* por un fichero con registros, contenido cada uno de ellos los atributos *nombre*, *dirección* y *teléfono* de un objeto *CPersona*.

La clase *CPersona*

La clase *CPersona* sólo se ve modificada por el hecho de haber añadido un método denominado *tamaño* que devuelve la longitud en bytes correspondiente a los atributos de un objeto *CPersona*.

```
/////////////////////////////
// Definición de la clase CPersona
//
public class CPersona
{
    // Atributos
    private String nombre;
    private String dirección;
    private long teléfono;

    // Métodos
    public CPersona() {}

    public CPersona(String nom, String dir, long tel)
    {
        nombre = nom;
        dirección = dir;
        teléfono = tel;
    }

    public void asignarNombre(String nom)
    {
        nombre = nom;
    }

    public String obtenerNombre()
    {
        return nombre;
    }

    public void asignarDirección(String dir)
    {
        dirección = dir;
    }
}
```

```

public String obtenerDirección()
{
    return dirección;
}

public void asignarTeléfono(long tel)
{
    teléfono = tel;
}

public long obtenerTeléfono()
{
    return teléfono;
}

public int tamaño()
{
    // Longitud en bytes de los atributos (un Long = 8 bytes)
    return nombre.length()*2 + dirección.length()*2 + 8;
}
}

```

La clase *CListaTfnos*

La interfaz de la clase *CListaTfnos* será prácticamente la misma. De esta forma un usuario no diferenciaría si está trabajando con una matriz o con un fichero, excepto en que ahora, al utilizar un fichero, los datos persisten de una ejecución a otra de la aplicación.

Constructor *CListaTfnos*

Cuando desde algún método se cree un objeto *CListaTfnos* ¿qué esperamos que ocurra? Lógicamente que se cargue la lista de teléfonos especificada, o bien que se cree una nueva cuando el fichero especificado no exista. Por ejemplo:

```

File fichero = new File("listatfnos.dat");
CListaTfnos listatfnos = new CListaTfnos(fichero);

```

Según lo expuesto, la lista de teléfonos especificada se cargará desde un fichero almacenado en el disco y si ese fichero no existe, se creará uno nuevo. Para ello, el constructor de la clase *CListaTfnos* abrirá un flujo para acceso aleatorio desde el fichero especificado, almacenará una referencia al mismo en un atributo *fes* de la clase, y en otro, *nregs*, almacenará el número de registros existentes en el fichero. Un atributo más, *tamañoRegs*, especificará el tamaño que hayamos previsto para cada registro. En nuestro caso, la información almacenada en un registro se corresponde con el *nombre*, *dirección* y *teléfono* de un objeto *CPersona*.

Ateniéndonos a lo explicado, veamos a continuación el esqueleto de la clase *CListaTfnos* y el constructor de la misma:

```
///////////////////////////////
// Definición de la clase CListaTfnos.
//
import java.io.*;
public class CListaTfnos
{
    private RandomAccessFile fes; // flujo
    private int nregs;           // número de registros
    private int tamañoReg = 140; // tamaño del registro en bytes

    public CListaTfnos(File fichero) throws IOException
    {
        if (fichero.exists() && !fichero.isFile())
            throw new IOException(fichero.getName() + " no es un fichero");
        fes = new RandomAccessFile(fichero, "rw");
        // Como es casi seguro que el último registro no ocupe el
        // tamaño fijado, utilizamos ceil para redondear por encima.
        nregs = (int)Math.ceil((double)fes.length() / (double)tamañoReg);
    }

    public void cerrar() throws IOException { fes.close(); }
    public int longitud() { return nregs; } // número de registros

    public boolean ponerValorEn( int i, CPersona objeto )
        throws IOException
    {
        // ...
    }

    public CPersona valorEn( int i ) throws IOException
    {
        // ..
    }

    public void añadir(CPersona obj) throws IOException
    {
        // ...
    }

    public boolean eliminar(long tel) throws IOException
    {
        // ...
    }

    public int buscar(String str, int pos) throws IOException
    {
        // ...
    }
}
```

Observe que el constructor de la clase verifica si el argumento pasado corresponde a un nombre existente en el directorio actual de trabajo y, en caso de que exista, si realmente se trata de un nombre de fichero; si no es así, lanzará una excepción del tipo **IOException**; en otro caso, abre un flujo desde el fichero para leer y escribir que permite el acceso aleatoriamente al mismo y calcula el número de registros del fichero.

Escribir un registro en el fichero

El método *ponerValorEn* se ha diseñado para que permita escribir los atributos de un objeto *CPersona* dentro del fichero a partir de una posición determinada. Tiene dos parámetros: el primero indica el número de registro que se desea escribir, que puede coincidir con un registro existente, en cuyo caso se sobreescibirá este último, o bien con el número del siguiente registro que se puede añadir al fichero; y el segundo, hace referencia al objeto *CPersona* cuyos atributos deseamos escribir. El método devolverá un valor **true** si se ejecuta satisfactoriamente y **false** en otro caso.

```
public boolean ponerValorEn( int i, CPersona objeto )
    throws IOException
{
    if (i >= 0 && i <= nregs)
    {
        if (objeto.tamaño() + 4 > tamañoReg)
            System.err.println("tamaño del registro excedido");
        else
        {
            fes.seek(i * tamañoReg); // situar el puntero de L/E
            fes.writeUTF(objeto.obtenerNombre());
            fes.writeUTF(objeto.obtenerDirección());
            fes.writeLong(objeto.obtenerTeléfono());
            return true;
        }
    }
    else
        System.err.println("número de registro fuera de límites");
    return false;
}
```

Se observa que lo primero que hace el método es verificar si el número de registro es válido (cuando *i* sea igual a *nregs* es porque se quiere añadir un registro al final del fichero). El primer registro es el cero. Después comprueba que el tamaño de los atributos del objeto *CPersona* más 4, no superen el tamaño establecido para el registro (más 4 porque cada vez que **writeUTF** escribe un **String**, añade 2 bytes iniciales para dejar constancia del número de bytes que se escriben; esto permitirá posteriormente al método **readUTF** saber cuántos bytes tiene que

leer). Si el tamaño está dentro de los límites permitidos, sitúa el puntero de L/E en la posición de inicio correspondiente a ese registro dentro del fichero y escribe los atributos del objeto uno a continuación de otro (vea la definición de `seek`).

Añadir un registro al final del fichero

El método `añadir` tiene como misión añadir un nuevo registro al final del fichero. Para ello, invoca al método `ponerValorEn` pasando como argumentos la posición que ocupará el nuevo registro, que coincide con el valor de `nregs`, y el objeto cuyos atributos se desean escribir.

```
public void añadir(CPersona obj) throws IOException
{
    if (ponerValorEn( nregs, obj )) nregs++;
}
```

Leer un registro del fichero

Para leer un registro del fichero que almacena la lista de teléfonos, la clase `CListaTfnos` proporciona el método `valorEn`. Este método tiene un parámetro para identificar el número de registro que se desea leer y devuelve el objeto `CPersona` creado a partir de los datos *nombre*, *dirección* y *teléfono* leídos desde el fichero.

```
public CPersona valorEn( int i ) throws IOException
{
    if (i >= 0 && i < nregs)
    {
        fes.seek(i * tamañoReg); // situar el puntero de L/E

        String nombre, dirección;
        long teléfono;
        nombre = fes.readUTF();
        dirección = fes.readUTF();
        teléfono = fes.readLong();

        return new CPersona(nombre, dirección, teléfono);
    }
    else
    {
        System.out.println("número de registro fuera de límites");
        return null;
    }
}
```

Se observa que lo primero que hace el método es verificar si el número de registro es válido (el primer registro es el cero). Si el número de registro está dentro de los límites permitidos, sitúa el puntero de L/E en la posición de inicio corres-

pondiente a ese registro dentro del fichero y lee los datos *nombre*, *dirección* y *teléfono* (esto se hace enviando al flujo *fes* vinculado con el fichero, el mensaje **readUTF**, una vez por cada dato). Finalmente, devuelve un objeto *CPersona* construido a partir de los datos leídos (el valor devuelto será **null** si el número de registro está fuera de límites).

Eliminar un registro del fichero

Puesto que el fichero manipulado se corresponde con una lista de teléfonos, parece lógico identificar el registro que se deseé eliminar por el número de teléfono, ya que éste es único. Para este propósito escribiremos un método *eliminar* con un parámetro que almacene el número de teléfono a eliminar y que devuelva un valor **true** si la operación se realiza con éxito, o **false** en caso contrario.

```
public boolean eliminar(long tel) throws IOException
{
    CPersona obj;
    // Buscar el teléfono y marcar el registro para
    // posteriormente eliminarlo
    for ( int reg_i = 0; reg_i < nregs; reg_i++ )
    {
        obj = valorEn(reg_i);
        if (obj.obtenerTeléfono() == tel)
        {
            obj.asignarTeléfono(0);
            ponerValorEn( reg_i, obj );
            return true;
        }
    }
    return false;
}
```

El proceso seguido por el método *eliminar* es leer registros del fichero, empezando por el registro cero, y comprobar por cada uno de ellos si el teléfono coincide con el valor pasado como argumento (este proceso recibe el nombre de búsqueda secuencial). Si existe un registro con el número de teléfono buscado, no se borra físicamente del fichero, sino que se marca el registro poniendo un cero como número de teléfono. Esta forma de proceder deja libertad al usuario de la clase *CListaTfnos* para eliminar de una sola vez todos los registros marcados al finalizar su aplicación, lo que redunda en velocidad de ejecución, para restaurar un registro marcado para eliminar, para crear un histórico, etc.

Buscar un registro en el fichero

Una operación muy común en el trabajo con registros es localizar uno determinado. ¿Cómo buscar un teléfono en una lista de teléfonos? Lo más común es buscar

por el nombre del propietario de ese teléfono, aunque también podría realizarse la búsqueda por la dirección. El método *buscar* que se expone a continuación permite realizar la búsqueda por cualquier subcadena perteneciente al nombre. Para ello utiliza dos parámetros: la subcadena a buscar y a partir de qué registro del fichero se desea buscar. Si la búsqueda termina con éxito, el método devuelve el número del registro correspondiente; en otro caso devuelve el valor -1.

```
public int buscar(String str, int pos) throws IOException
{
    // Buscar un registro por una subcadena del nombre
    // a partir de un registro determinado
    CPersona obj;
    String nom;
    if (str == null) return -1;
    if (pos < 0) pos = 0;
    for (int reg_i = pos; reg_i < nregs; reg_i++)
    {
        obj = valorEn(reg_i);
        nom = obj.obtenerNombre();
        // ¿str está contenida en nom?
        if (nom.indexOf(str) > -1)
            return reg_i;
    }
    return -1;
}
```

Se observa que el método *buscar*, al igual que el método *eliminar*, realiza una búsqueda secuencial desde el registro *pos*, comprobando si el nombre de alguno de ellos contiene la subcadena *str*. Lógicamente, al realizar una búsqueda secuencial, el resultado será el número del primer registro que contenga en su nombre la subcadena pasada como argumento; pero también es evidente que es posible continuar la búsqueda a partir del siguiente registro, invocando de nuevo al método *buscar*, pasando como argumentos la misma subcadena y el número de registro siguiente al devuelto en el proceso de búsqueda anterior.

Un ejemplo de acceso aleatorio a un fichero

A continuación vamos a escribir una aplicación basada en una clase *Test* para trabajar con una lista de teléfonos construida a partir de un objeto *CListaTfnos*. Esta aplicación será prácticamente la misma que vimos en las versiones de la aplicación “lista de teléfonos” desarrollada en el capítulo 9 y modificada en el 11 y al principio de éste; por eso no abundaremos en detalles en aquellas partes que ya hayan sido explicadas. Esto demuestra que cuando una clase, como *CListaTfnos*, se modifica y no se alteran los prototipos de los métodos que componen su interfaz, las aplicaciones que la utilizan no se ven afectadas. En nuestro caso, esta clase tiene

un nuevo constructor con un parámetro de tipo **File** (el otro constructor lo eliminamos para no complicar la clase, pero lo podíamos haber conservado).

El esqueleto de esta aplicación se muestra a continuación. En él se puede observar que se han definido como atributos de la clase *Test* los flujos para acceder a la entrada y salida estándar, así como una referencia *listatfnos* al objeto que encapsulará la lista de teléfonos con la que deseamos trabajar.

```
import java.io.*;
///////////////////////////////
// Aplicación para trabajar con un fichero accedido aleatoriamente
//
public class Test
{
    // Definir una referencia al flujo estándar de salida: flujoS
    static PrintStream flujoS = System.out;

    static CListaTfnos listatfnos;

    public static boolean modificar(int nreg) throws IOException
    {
        // ...
    }

    public static void actualizar(File fActual) throws IOException
    {
        // ...
    }

    public static int menú()
    {
        // ...
    }

    public static void main(String[] args)
    {
        int opción = 0, pos = -1;
        String cadenabuscar = null;
        String nombre, dirección;
        long teléfono;

        boolean eliminado = false;
        boolean modificado = false;
        boolean listaModificada = false;

        try
        {
            // Crear un objeto lista de teléfonos vacío (con 0 elementos)
            // o con el contenido del fichero listatfnos.dat si existe.
```

```

File fichero = new File("listatfnos.dat");
listatfnos = new CListaTfnos(fichero);

do
{
    opción = menú();

    switch (opción)
    {
        case 1: // buscar
            // ...
        case 2: // buscar siguiente
            // ...
        case 3: // modificar
            // ...
        case 4: // añadir
            // ...
        case 5: // eliminar
            // ...
        case 6: // salir
            // ...
    }
}
while(opción != 6);
}

catch (IOException e)
{
    flujoS.println("Error: " + e.getMessage());
}
}

```

La ejecución de la aplicación se iniciará por el método **main** que, en primer lugar, crea el objeto *CListaTfnos* cuya interfaz nos dará acceso aleatorio al fichero especificado. Después, ejecutará un bucle que invocará al método *menú* encargado de solicitar la elección de una de las opciones presentadas por él:

```
public static int menú()
{
    flujoS.print("\n\n");
    flujoS.println("1. Buscar");
    flujoS.println("2. Buscar siguiente");
    flujoS.println("3. Modificar");
    flujoS.println("4. Añadir");
    flujoS.println("5. Eliminar");
    flujoS.println("6. Salir");
    flujoS.println();
    flujoS.print("    Opción: ");
}
```

```

int op;
do
    op = Leer.datoInt();
while (op < 1 || op > 6);
return op;
}

```

Elegida una opción del menú presentado, una sentencia **switch** permitirá ejecutar el código que dará solución a la operación seleccionada. Las opciones *Buscar*, *Buscar siguiente*, *Añadir* y *Eliminar* no han variado respecto a la versión de las mismas presentada en el apartado “Seriar objetos que referencian a objetos” expuesto anteriormente en este mismo capítulo, a excepción de que cuando se muestra la información de un registro, ahora también se muestra el número del mismo, y de que al salir de la aplicación, los cambios debidos a *Añadir* o *Eliminar* ya han realizados sobre el fichero (ahora se trabaja directamente sobre el fichero). El código completo lo puede ver en el CD-ROM que acompaña al libro.

Modificar un registro

Una operación importante en el trabajo con ficheros que se puede realizar de forma rápida y fácil cuando se permite el acceso aleatorio al mismo es modificar alguna parte concreta de la información almacenada en él. En nuestro caso, el objetivo es modificar un registro. Para ello vamos a añadir a la clase aplicación, un método estático denominado *modificar* con un parámetro que identifique el número de registro del fichero que se desea modificar. Si durante la ejecución no sabemos con exactitud el número del registro que se desea modificar, podemos utilizar las opciones *Buscar* y *Buscar siguiente* para obtenerlo.

Para realizar tal modificación, el proceso seguido por el método es:

- Leer el registro correspondiente al número pasado como argumento y crear un objeto *CPersona* a partir de los datos leídos. Esto permitirá manipular el registro utilizando la interfaz del objeto.
- Presentar un menú que permita modificar el *nombre*, la *dirección* o el *teléfono*, así como salir del proceso guardando los cambios efectuados, o bien salir sin guardar los cambios. Los nuevos datos serán solicitados desde el teclado.
- Una vez realizadas las modificaciones, si se eligió salir guardando los cambios efectuados, el método enviará al objeto *CListaTfnos* el mensaje *ponerValorEn* pasando como argumento el número de registro que se está modificando y el objeto *CPersona* que aporta los nuevos atributos; el resultado es que se sobreescribe en el fichero el registro especificado.

```
public static boolean modificar(int nreg) throws IOException
{
    String nombre, dirección;
    long teléfono;
    int op;
    // Leer el registro
    CPersona obj = listatfnos.valorEn(nreg);
    if (obj == null) return false;

    // Modificar el registro
    do
    {
        flujoS.print("\n\n");
        flujoS.println("Modificar el dato:");
        flujoS.println("1. Nombre");
        flujoS.println("2. Dirección");
        flujoS.println("3. Teléfono");
        flujoS.println("4. Salir y salvar los cambios");
        flujoS.println("5. Salir sin salvar los cambios");
        flujoS.println();
        flujoS.print("    Opción: ");
        op = Leer.datoInt();

        switch( op )
        {
            case 1: // modificar nombre
                flujoS.print("nombre:    ");
                nombre = Leer.dato();
                obj.asignarNombre(nombre);
                break;
            case 2: // modificar dirección
                flujoS.print("dirección: ");
                dirección = Leer.dato();
                obj.asignarDirección(dirección);
                break;
            case 3: // modificar teléfono
                flujoS.print("teléfono: ");
                teléfono = Leer.datoLong();
                obj.asignarTeléfono(teléfono);
                break;
            case 4: // guardar los cambios
                break;
            case 5: // salir sin guardar los cambios
                break;
        }
    } while( op != 4 && op != 5);

    if (op == 4)
    {
```

```

        listatfnos.ponerValorEn(nreg, obj);
        return true;
    }
    else
        return false;
}

```

Actualizar el fichero

Los datos del fichero con el que estamos trabajando pueden verse alterados por tres procesos diferentes: *modificar*, *añadir* o *eliminar* un registro. En el caso de *modificar* o *añadir* un registro los cambios son realizados directamente sobre el fichero. Pero en el caso de *eliminar* un registro, éste simplemente es marcado con un número de teléfono 0 para su posterior eliminación, si se cree conveniente. En nuestro caso, vamos a escribir en la clase aplicación un método *actualizar* que se invoque cuando el usuario de la aplicación seleccione la opción *Salir*, con el objeto de actualizar el fichero, eliminando físicamente los registros marcados.

```

case 6: // salir
    // guardar lista
    if (eliminado) actualizar(fichero);
    listatfnos = null;
}

```

El proceso seguido para realizar lo expuesto es sencillo. Básicamente se creará un fichero temporal (fichero que existe durante un corto espacio de tiempo, mientras lo necesitemos) para guardar todos los registros del fichero actual cuyo número de teléfono sea distinto de cero. Después de realizar esta operación, cerraremos ambos ficheros y utilizaremos la interfaz de la clase **File** para borrar el fichero actual y renombrar el fichero temporal con el nombre que tenía el fichero actual.

```

public static void actualizar(File fActual) throws IOException
{
    // Crear un fichero temporal
    File ficheroTemp = new File("listatfnos.tmp");
    CListaTfnos ftemp = new CListaTfnos(ficheroTemp);

    int nregs = listatfnos.longitud();
    // Copiar en el fichero temporal todos los registros del
    // fichero actual que en su campo teléfono no tengan un 0
    CPersona obj;
    for (int reg_i = 0; reg_i < nregs; reg_i++)
    {
        obj = listatfnos.valorEn(reg_i);
        if (obj.obtenerTeléfono() != 0)
            ftemp.añadir(obj);
    }
}

```

```

listatfnos.cerrar();
ftemp.cerrar();
fActual.delete();
if (!ficheroTemp.renameTo(fActual))
    throw new IOException("no se renombró el fichero");
}

```

UTILIZACIÓN DE DISPOSITIVOS ESTÁNDAR

La salida de un programa puede también ser enviada a un dispositivo de salida que no sea el disco o la pantalla; por ejemplo, a una impresora conectada al puerto paralelo. Como Java no tiene definido un flujo estándar para el puerto paralelo, la solución es definir uno y vincularlo a dicho dispositivo.

Una forma de realizar lo expuesto es crear un flujo hacia el dispositivo *LPT1*, *LPT2*, o *PRN* y escribir en ese flujo (los nombres indicados son los establecidos por Windows para nombrar a la impresora; en UNIX la primera impresora tiene asociado el nombre */dev/lp0*, la segunda */dev/lp1*, etc.). Las siguientes líneas de código muestran cómo realizar esto:

```

// Crear un flujo hacia la impresora
FileWriter flujoS = new FileWriter("LPT1");
flujoS.write("Esta línea se escribe en la impresora\r\n");
flujoS.write("\r\n"); // saltar una línea
Long n = new Long(123456789);
flujoS.write("Valor: " + n.toString() + "\r\n");
flujoS.write("\f"); // saltar a la página siguiente
flujoS.close(); // cerrar el flujo

```

El flujo creado es de la clase **FileWriter**, pero se podría haber creado de otra clase que permita definir flujos de salida, como **FileOutputStream**, **DataOutputStream**, **RandomAccessFile**, etc. Se puede observar que para obtener datos impresos legibles se envían cadenas de caracteres a la impresora, ya que se trata de un dispositivo ASCII. En general podemos enviar datos de tipo **char** o **byte**.

Como ejemplo, vamos a añadir al menú de la aplicación anterior, una opción *imprimir* que permita obtener la lista de teléfonos por la impresora.

```

case 6: // imprimir
    imprimirListaTfnos();
    break;
case 7: // salir
    // ...

```

El código anterior indica que cuando el usuario seleccione la opción 6 del menú de la aplicación, se invocará al método estático *imprimirListaTfnos* de la clase aplicación *Test*. Este método, creará un flujo hacia la impresora, obtendrá el número total de registros del fichero “lista de teléfonos” y establecerá un bucle para imprimir cada uno de ellos. El código completo se muestra a continuación:

```
public static void imprimirListaTfnos() throws IOException
{
    // Crear un flujo hacia la impresora
    FileWriter flujoS = new FileWriter("LPT1");

    String crlf = "\r\n"; // cambiar a la siguiente línea
    String ff = "\f"; // saltar a la siguiente página
    Integer i; // referencia a un objeto Integer
    Long l; // referencia a un objeto Long
    int nregs = listatfnos.longitud(); // número de registros

    for (int n = 0; n < nregs; n++)
    {
        // Saltar página inicialmente y después cada 60 líneas
        if (n % 60 == 0) flujoS.write(ff);
        // Imprimir el registro n de la lista de teléfonos
        i = new Integer(n); // número de registro
        flujoS.write("Registro: " + i.toString() + crlf);
        flujoS.write(listatfnos.valorEn(n).obtenerNombre() + crlf);
        flujoS.write(listatfnos.valorEn(n).obtenerDirección() + crlf);
        l = new Long(listatfnos.valorEn(n).obtenerTeléfono());
        flujoS.write(l.toString() + crlf);
        flujoS.write(crlf); // saltar una línea
    }
    flujoS.write(ff); // saltar a la siguiente página
    flujoS.close(); // cerrar el flujo hacia la impresora
}
```

EJERCICIOS RESUELTOS

1. Escribir una clase aplicación denominada *CopiarFichero* que permita copiar el contenido de un fichero en otro. La aplicación será invocada de la forma siguiente:

```
java CopiarFichero <fichero fuente> <fichero destino>
```

Este ejemplo utilizará la clase **File** para asegurarse de que el fichero fuente existe y no está protegido contra lectura. También utilizará esta clase para asegurarse de que el fichero destino existe y no está protegido contra escritura, o bien se trata de un directorio no protegido contra escritura, destino del fichero.

Para leer los bytes del fichero fuente y escribirlos en el destino, este ejemplo utilizará las clases **FileInputStream** y **FileOutputStream**, respectivamente.

La funcionalidad de la clase *CopiarFichero* estará soportada fundamentalmente por el método *copiar* que tiene el prototipo siguiente:

```
public static void copiar(String fuente, String destino)
```

Este método, básicamente chequea la existencia y permisos de los ficheros fuente y destino y copia el fichero origen en el destino; si el fichero destino existe pregunta si se desea sobreescribir. En el caso de que ocurra algún error, este método lanzará una excepción del tipo *ECopiarFichero* indicando lo ocurrido. Finalmente, utilizará un bloque **finally** para cerrar los flujos abiertos.

A continuación se muestra la aplicación completa, suficientemente comentada como para no tener que abundar en más explicaciones:

```
import java.io.*;

public class CopiarFichero
{
    public static void copiar(String fuente, String destino)
        throws IOException
    {
        // Si el fichero fuente y el destino son el mismo fichero ...
        if (fuente.compareTo(destino) == 0)
            throw new ECopiarFichero("No puede sobreescibirse un " +
                "fichero sobre sí mismo");

        // Definiciones de variables, referencias y objetos
        File fichFuente = new File(fuente);
        File fichDestino = new File(destino);
        FileInputStream fFuente = null;
        FileOutputStream fDestino = null;
        byte[] buffer;
        int nbytes;

        try
        {
            // Asegurarse de que "fuente" es un fichero, existe
            // y se puede leer.
            if (!fichFuente.exists() || !fichFuente.isFile())
                throw new ECopiarFichero("No existe el fichero " + fuente);
            if (!fichFuente.canRead())
                throw new ECopiarFichero("El fichero " + fuente +
                    " no se puede leer");
            // Si "destino" existe, asegurarse de que es un fichero que
            // se puede escribir y preguntar si se quiere sobreescibir.
```

```

if (fichDestino.exists()) // ¿existe el destino?
{
    if (fichDestino.isFile()) // ¿es un fichero?
    {
        if (!fichDestino.canWrite())
            throw new ECopiarFichero("No se puede escribir en " +
                                         "el fichero " + destino);
        // Indicar que el fichero existe y preguntar si se desea
        // sobreescribir.
        System.out.print("El fichero " + destino + " existe. " +
                         "¿Desea sobreescribirlo? (s/n): ");
        // Leer la respuesta
        char resp = (char)System.in.read();
        System.in.skip(System.in.available());
        if (resp == 'n' || resp == 'N')
            throw new ECopiarFichero("Copia cancelada");
    }
    else
        throw new ECopiarFichero(destino + " no es un fichero");
}
else // si "destino" no existe verificar que el directorio
      // padre existe y no está protegido contra escritura
{
    File dirPadre = directorioPadre(fichDestino);

    if (!dirPadre.exists())
        throw new ECopiarFichero("El directorio " + destino +
                                         " no existe");
    if (!dirPadre.canWrite())
        throw new ECopiarFichero("No se puede escribir en el " +
                                         "directorio " + destino);
}

// Para realizar la copia, abrir un flujo de entrada desde
// el fichero fuente y otro de salida hacia el destino.
fFuente = new FileInputStream(fichFuente);
fDestino = new FileOutputStream(fichDestino);
buffer = new byte[1024];

// Copiar el fichero fuente en el destino
while (true)
{
    nbytes = fFuente.read(buffer);
    if (nbytes == -1) break; // se llegó al final del fichero
    fDestino.write(buffer, 0, nbytes);
}
// Cerrar cualquier flujo que esté abierto
finally
{

```

```
try
{
    if (fFuente != null) fFuente.close();
    if (fDestino != null) fDestino.close();
}
catch(IOException e)
{
    System.out.println("Error: " + e.toString());
}
}

// File.getParent devuelve null si el fichero se especifica sin
// un directorio. El método siguiente trata este caso.
private static File directorioPadre(File f)
{
    String nombreDir = f.getParent();
    if (nombreDir == null)
        // El método getProperty con el parámetro "user.dir" devuelve
        // el directorio actual de trabajo.
        return new File(System.getProperty("user.dir"));
    else
        // Devolver el directorio padre del fichero
        return new File(nombreDir);
}

public static void main(String[] args)
{
    // main debe recibir dos parámetros: el fichero fuente y
    // el destino.
    if (args.length != 2)
        System.err.println("Sintaxis: java CopiarFichero " +
                           "<fichero fuente> <fichero destino>");
    else
    {
        try
        {
            copiar(args[0], args[1]); // realizar la copia
        }
        catch(IOException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```
// Si se produce un error durante la copia, se lanzará
// el siguiente tipo de excepción:
class ECopiarFichero extends IOException
{
    public ECopiarFichero(String mensaje)
    {
        super(mensaje);
    }
}
```

2. Queremos escribir una aplicación denominada *Grep* que permita buscar palabras en uno o más ficheros de texto. Como resultado se visualizará, por cada uno de los ficheros, su nombre, el número de línea y el contenido de la misma para cada una de las líneas del fichero que contenga la palabra buscada.

La clase aplicación, *Grep*, deberá proporcionar al menos los siguientes métodos:

- a) *BuscarCadena* para buscar una cadena de caracteres dentro de otra. El prototipo de este método será:

```
static boolean BuscarCadena(String cadena1, String cadena2)
```

Este método devolverá **true** si *cadena2* se encuentra dentro de *cadena1*; en otro caso, devolverá **false**.

- b) *BuscarEnFich* para buscar una cadena de caracteres en un fichero de texto e imprimir el número y el contenido de la línea que contiene a la cadena. El prototipo de este método será:

```
static void BuscarEnFich(String nombrefich, String cadena)
```

- c) **main** para que utilizando los métodos anteriores permita buscar una palabra en uno o más ficheros.

La forma de invocar a la aplicación será así:

```
java Grep palabra fichero_1 fichero_2 ... fichero_n
```

A continuación se muestra la aplicación completa, suficientemente comentada.

Observe que los datos obtenidos del fichero fuente son filtrados dos veces para poder llegar a utilizar el método **readLine** de **BufferedReader**. Recuerde que este método permite leer líneas de texto. Concretamente lo que se ha hecho ha sido:

- Crear un flujo **FileInputStream** asociado con el fichero de texto.
- Conectar un filtro **InputStreamReader** con el flujo anterior.
- Y finalmente, conectar el filtro **BufferedReader** con el filtro anterior.

```
import java.io.*;  
  
public class Grep  
{  
    public static boolean BuscarCadena(String cadenal, String cadena2)  
    {  
        // ¿cadena2 está contenida en cadenal?  
        if (cadenal.indexOf(cadena2) > -1)  
            return true; // sí  
        else  
            return false; // no  
    }  
  
    public static void BuscarEnFich(String nombrefich, String cadena)  
    {  
        // Definiciones de variables  
        File fichFuente = new File(nombrefich);  
        BufferedReader flujoE = null;  
  
        try  
        {  
            // Asegurarse de que el fichero, existe y se puede leer  
            if (!fichFuente.exists() || !fichFuente.isFile())  
            {  
                System.err.println("No existe el fichero " + nombrefich);  
                return;  
            }  
            if (!fichFuente.canRead())  
            {  
                System.err.println("El fichero " + nombrefich +  
                                   " no se puede leer");  
                return;  
            }  
  
            // Abrir un flujo de entrada desde el fichero fuente  
            FileInputStream fis = new FileInputStream(fichFuente);  
            InputStreamReader isr = new InputStreamReader(fis);  
            flujoE = new BufferedReader(isr);  
  
            // Buscar cadena en el fichero fuente  
            String linea;  
            int nroLinea = 0;  
  
            while ((linea = flujoE.readLine()) != null)  
            {  
                if (linea.contains(cadena))  
                    System.out.println("Linea " + nroLinea + " : " + linea);  
            }  
        }  
        catch (Exception e)  
        {  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```

```

        // Si se alcanzó el final del fichero,
        // readLine devuelve null
        nroLinea++; // contador de líneas
        if (BuscarCadena(linea, cadena))
            System.out.println(nombrefich + " " + nroLinea + " " +
                linea);
    }
}
catch(IOException e)
{
    System.out.println("Error: " + e.getMessage());
}
finally
{
    // Cerrar el flujo
    try
    {
        if (flujoE != null) flujoE.close();
    }
    catch(IOException e)
    {
        System.out.println("Error: " + e.toString());
    }
}
public static void main(String[] args)
{
    // main debe recibir dos o más parámetros: la cadena a buscar
    // y los ficheros fuente. Por ejemplo:
    // java Grep catch Grep.java Leer.java

    if (args.length < 2)
        System.err.println("Sintaxis: java Grep " + "<cadena> " +
                           "<fichero 1> <fichero 2> ...");
    else
    {
        for (int i = 1; i < args.length; i++)
            // Buscar args[0] en args[i]
            BuscarEnFich(args[i], args[0]);
    }
}
}

```

3. Realizar un programa que permita crear un fichero nuevo, abrir uno existente, añadir, modificar o eliminar registros, y visualizar el contenido del fichero. El nombre del fichero será introducido a través del teclado. Cada registro del fichero

estará formado por los datos *referencia* y *precio*. Así mismo, para que el usuario pueda elegir cualquiera de las operaciones enunciadas, el programa visualizará en pantalla un menú similar al siguiente:

Nombre del fichero: artículos

1. Fichero nuevo
2. Abrir fichero
3. Añadir registro
4. Modificar registro
5. Eliminar registro
6. Visualizar registros
7. Salir

Opción:

No se permitirá crear un *Fichero nuevo* cuando exista, ni *Abrir un fichero* que no exista. Cuando se intente *Abrir un fichero* que no exista, se ofrecerá la posibilidad de mostrar un listado del directorio actual. Finalmente, la opción *Visualizar registros* permitirá mostrar aquellos registros cuya referencia sea una especificada, o bien contenga una subcadena especificada.

Se deberá realizar al menos un método para cada una de las opciones, excepto para *Salir*.

A partir de un análisis del enunciado se deduce que, además del objeto aplicación (objeto de una clase que denominaremos *Test*), potencialmente existen dos clases de objetos más: una que represente al fichero y otra que represente a los registros del fichero.

Escribiremos entonces una clase *CRegistro* para manipular cada uno de los registros de un fichero y otra *CBaseDeDatos* con una interfaz pública que permita realizar las operaciones habituales de trabajo sobre un fichero.

Según el enunciado, la funcionalidad de la clase *CRegistro* estará soportada por los atributos *referencia* y *precio* y por los métodos siguientes:

- Un constructor sin parámetros y otro con parámetros para poder crear objetos con unos atributos determinados.
- Los métodos *obtenerReferencia* y *obtenerPrecio* para obtener los valores de los campos de un registro (atributos del objeto *CRegistro*).
- Los métodos *asignarReferencia* y *asignarPrecio* para asignar nuevos valores a los campos de un registro (atributos del objeto *CRegistro*).
- Y el método *tamaño* que devolverá el tamaño en bytes de los atributos.

La declaración de esta clase se muestra a continuación:

```
///////////////////////////// /////////////////////////////////
// Definición de la clase CRegistro
//
public class CRegistro
{
    // Atributos
    private String referencia;
    private double precio;

    // Métodos
    public CRegistro() {}
    public CRegistro(String ref, double pre)
    {
        referencia = ref;
        precio = pre;
    }

    public void asignarReferencia(String ref)
    {
        referencia = ref;
    }

    public String obtenerReferencia()
    {
        return referencia;
    }

    public void asignarPrecio(double pre)
    {
        precio = pre;
    }

    public double obtenerPrecio()
    {
        return precio;
    }

    public int tamaño()
    {
        // Longitud en bytes de los atributos (un Double = 8 bytes)
        return referencia.length()*2 + 8;
    }
}
```

Siguiendo con el ejemplo, la funcionalidad de la clase *CBaseDatos* deberá permitir, abrir el fichero, cerrarlo, calcular su longitud, insertar, obtener, buscar y eliminar un registro, así como actualizar el fichero cuando sea preciso. Para ello,

dotaremos a esta clase de cuatro atributos, un objeto **File** que encapsule el nombre del fichero actual de trabajo, un flujo vinculado con el fichero, el número de registros del fichero y la longitud estimada para cada registro; y de los siguientes métodos:

- Un constructor que admita como argumento un objeto **File** que proporcione el nombre de la base de datos.
- Los métodos *cerrar* y *longitud* para cerrar el fichero y calcular su longitud, respectivamente.
- Los métodos *ponerValorEn*, para sobreescribir un registro en una posición cualquiera dentro del fichero, y *añadir* para insertarlo al final.
- El método *valorEn* para obtener un registro del fichero.
- El método *buscar* para localizar un determinado registro en el fichero.
- El método *eliminar* para marcar un registro del fichero como eliminado.
- Y el método *actualizar* para eliminar físicamente del fichero los registros marcados por el método *eliminar*.

Según lo expuesto, la declaración de la clase *CBaseDeDatos* puede ser como se muestra a continuación. Los comentarios introducidos son suficientes para entender el código sin necesidad de tener que abundar más explicaciones.

```
////////////////////////////////////////////////////////////////
// Definición de la clase CBaseDeDatos.
//
import java.io.*;
public class CBaseDeDatos
{
    // Atributos
    private File ficheroActual;    // objeto File (nombre del fichero)
    private RandomAccessFile fes; // flujo hacia/desde el fichero
    private int nregs;           // número de registros
    private int tamañoReg = 50;   // tamaño del registro en bytes

    // Métodos
    public CBaseDeDatos(File fichero) throws IOException
    {
        // ¿Existe el fichero?
        if (fichero.exists() && !fichero.isFile())
            throw new IOException(fichero.getName() + " no es un fichero");
        // Asignar valores a los atributos
        ficheroActual = fichero;
        fes = new RandomAccessFile(fichero, "rw");
    }
}
```

```
// El último registro no ocupa el tamaño especificado.  
// Por esta causa utilizamos ceil, para redondear por encima.  
nregs = (int)Math.ceil((double)fes.length() / (double)tamañoReg);  
}  
  
public void cerrar() throws IOException { fes.close(); }  
  
public int longitud() { return nregs; } // número de registros  
  
public boolean ponerValorEn( int i, CRegistro objeto )  
throws IOException  
{  
    if (i >= 0 && i <= nregs)  
    {  
        // Los 2 primeros bytes que escribe writeUTF indican la  
        // longitud de la String que escribe a continuación. Esta  
        // información es utilizada por readUTF.  
        if (objeto.tamaño() + 2 > tamañoReg)  
            System.err.println("tamaño del registro excedido");  
        else  
        {  
            // Situar el puntero de L/E en el registro i.  
            fes.seek(i * tamañoReg);  
            // Sobreescribir el registro con la nueva información  
            fes.writeUTF(objeto.obtenerReferencia());  
            fes.writeDouble(objeto.obtenerPrecio());  
            return true;  
        }  
    }  
    else  
        System.err.println("número de registro fuera de límites");  
    return false;  
}  
  
public void añadir(CRegistro obj) throws IOException  
{  
    // Añadir un registro al final del fichero e incrementar  
    // el número de registros  
    if (ponerValorEn( nregs, obj )) nregs++;  
}  
  
public CRegistro valorEn( int i ) throws IOException  
{  
    if (i >= 0 && i < nregs)  
    {  
        // Situar el puntero de L/E en el registro i.  
        fes.seek(i * tamañoReg);  
  
        String referencia;  
        double precio;
```

```
// Leer la información correspondiente al registro i.
referencia = fes.readUTF();
precio = fes.readDouble();

// Devolver el objeto CRegistro correspondiente.
return new CRegistro(referencia, precio);
}
else
{
    System.out.println("número de registro fuera de límites");
    return null;
}
}

public int buscar(String str, int nreg) throws IOException
{
    // Buscar un registro por una subcadena de la referencia
    // a partir de un registro determinado. Si se encuentra,
    // se devuelve el número de registro, o -1 en otro caso.
    CRegistro obj;
    String ref;
    if (str == null) return -1;
    if (nreg < 0) nreg = 0;
    for ( int reg_i = nreg; reg_i < nregs; reg_i++ )
    {
        // Obtener el registro reg_i
        obj = valorEn(reg_i);
        // Obtener su referencia
        ref = obj.obtenerReferencia();
        // ¿str está contenida en referencia?
        if (ref.indexOf(str) > -1)
            return reg_i; // devolver el número de registro
    }
    return -1; // la búsqueda falló
}

public boolean eliminar(String ref) throws IOException
{
    CRegistro obj;
    // Buscar la referencia y marcar el registro correspondiente
    // para poder eliminarlo en otro proceso.
    for ( int reg_i = 0; reg_i < nregs; reg_i++ )
    {
        // Obtener el registro reg_i
        obj = valorEn(reg_i);
        // ¿Tiene la referencia ref?
        if (ref.compareTo(obj.obtenerReferencia()) == 0)
        {
            // Marcar el registro con la referencia "borrar"
            obj.asignarReferencia("borrar");
        }
    }
}
```

```

    // Grabarlo
    ponerValorEn( reg_i, obj );
    return true;
}
}
return false;
}

public void actualizar() throws IOException
{
    // Crear un fichero temporal.
    File ficheroTemp = new File("articulos.tmp");
    CBaseDeDatos ftemp = new CBaseDeDatos(ficheroTemp);

    // Copiar en el fichero temporal todos los registros del
    // fichero actual que no estén marcados para "borrar"
    CRegistro obj;
    for ( int reg_i = 0; reg_i < nregs; reg_i++ )
    {
        obj = valorEn(reg_i);
        if (obj.obtenerReferencia().compareTo("borrar") != 0)
            ftemp.añadir(obj);
    }
    // Borrar el fichero actual y renombrar el temporal con el
    // nombre del actual. Para hacer estas operaciones los ficheros
    // no pueden estar en uso.
    this.cerrar();           // cerrar el fichero actual
    ftemp.cerrar();          // cerrar el fichero temporal
    ficheroActual.delete(); // borrar el fichero actual
    if (!ficheroTemp.renameTo(ficheroActual)) // renombrar
        throw new IOException("no se actualizó el fichero");
}
}

```

Volviendo al enunciado del programa, éste tiene que permitir a través de un menú, crear un fichero nuevo, abrir un fichero existente, añadir, modificar o eliminar un registro del fichero y visualizar un conjunto determinado de registros. El método *menú* presentará todas estas opciones en pantalla y devolverá como resultado un entero (1, 2, 3, 4, 5, 6 ó 7) correspondiente a la opción elegida por el usuario. Este menú junto con el esqueleto de la clase aplicación se muestra a continuación:

```

import java.io.*;
///////////////////////////////
// Aplicación para trabajar con un fichero accedido aleatoriamente
// Utiliza la clase Leer para leer de la entrada estándar cadenas
// y datos de tipos primitivos.
public class Test
{

```

```
// Definir una referencia al flujo estándar de salida: flujoS
static PrintStream flujoS = System.out;
static CBaseDeDatos artículos;
static boolean ficheroAbierto = false;

public static void nuevoFich() throws IOException
{
    // ...
}

public static void abrirFich() throws IOException
{
    // ...
}

public static void añadirReg() throws IOException
{
    // ...
}

public static void modificarReg() throws IOException
{
    // ...
}

public static boolean eliminarReg() throws IOException
{
    // ...
}

public static void visualizarRegs() throws IOException
{
    // ...
}

public static int menú()
{
    flujoS.print("\n\n");
    flujoS.println("1. Nuevo fichero");
    flujoS.println("2. Abrir fichero");
    flujoS.println("3. Añadir registro");
    flujoS.println("4. Modificar registro");
    flujoS.println("5. Eliminar registro");
    flujoS.println("6. Visualizar registros");
    flujoS.println("7. Salir");
    flujoS.println();
    flujoS.print("    Opción: ");
    int op;
    do
    {
```

```

op = Leer.datoInt();
if (op < 1 || op > 7)
    flujoS.print("Opción no válida. Elija otra: ");
}
while (op < 1 || op > 7);

if (op > 2 && op < 7 && !ficheroAbierto)
{
    flujoS.println("No hay un fichero abierto.");
    return 0;
}
return op;
}

public static void main(String[] args)
{
    int opción = 0;
    boolean eliminado = false; // true cuando se marque un registro
                                // para "borrar"
    try
    {
        do
        {
            opción = menú();
            switch (opción)
            {
                case 1: // nuevo fichero
                    nuevoFich();
                    break;
                case 2: // abrir fichero
                    abrirFich();
                    break;
                case 3: // añadir registro al final del fichero
                    añadirReg();
                    break;
                case 4: // modificar registro
                    modificarReg();
                    break;
                case 5: // eliminar registro
                    eliminado = eliminarReg();
                    break;
                case 6: // visualizar registros
                    visualizarRegs();
                    break;
                case 7: // salir
                    if (eliminado) artículos.actualizar();
                    artículos = null;
            }
        }
        while(opción != 7);
    }
}

```

```
        }
    catch (IOException e)
    {
        flujoS.println("Error: " + e.getMessage());
    }
}
```

Se puede observar que la clase aplicación *Test* define tres atributos: un flujo hacia la salida estándar, una referencia a la base de datos (fichero) con la que se va a trabajar y una variable *ficheroAbierto* de tipo **boolean** para saber en todo momento si hay o no un fichero abierto (su valor será **true** si el fichero está abierto y **false** en caso contrario). Esta variable será utilizada para no crear o abrir un fichero cuando ya haya uno abierto, y para no intentar añadir, modificar, eliminar o visualizar registros cuando no haya un fichero abierto.

Cada una de las opciones del menú, excepto la opción *Salir*, se resuelve ejecutando un método de los expuestos a continuación.

Finalmente, cuando se seleccione la opción *Salir*, se actualizará el fichero sólo si se marcó algún registro para borrar. El resto de las operaciones (añadir y modificar) realizan los cambios directamente sobre el fichero.

Nuevo fichero

El método *nuevoFich* tiene como misión crear un fichero vacío cuyo nombre especificaremos a través del teclado, sólo si dicho fichero no existe; si existe, se solicitará un nuevo nombre de fichero. Finalmente, a partir del fichero especificado creará un objeto *artículos* de la clase *CBaseDeDatos* cuya interfaz nos permitirá operar sobre ese fichero.

```
public static void nuevoFich() throws IOException
{
    if (ficheroAbierto)
    {
        flujoS.println("Ya hay un fichero abierto.");
        return;
    }

    flujoS.print("Nombre del fichero: ");
    File objFichero = new File(Leer.dato()); // nombre fichero
    while (objFichero.exists())
    {
        flujoS.println("Este fichero existe. Escriba otro.");
        objFichero = new File(Leer.dato());
    }
    articulos = new CBaseDeDatos(objFichero);
}
```

```
ficheroAbierto = true;
}
```

Abrir fichero

El método *abrirFich* tiene como misión abrir un fichero existente cuyo nombre especificaremos a través del teclado. Si el nombre especificado para el fichero no se localiza en el directorio actual de trabajo, se dará la posibilidad de visualizar el contenido de este directorio y de introducir un nuevo nombre. Finalmente, a partir del fichero especificado creará un objeto *artículos* de la clase *CBaseDeDatos* cuya interfaz nos permitirá operar sobre ese fichero.

```
public static void abrirFich() throws IOException
{
    if (ficheroAbierto)
    {
        flujoS.println("Ya hay un fichero abierto.");
        return;
    }

    flujoS.print("Nombre del fichero: ");
    File objFichero = new File(Leer.dato()); // nombre fichero

    File obj = null;
    char resp;
    while (!objFichero.exists())
    {
        flujoS.println("Este fichero no existe.");
        flujoS.print("¿Desea ver la lista de ficheros? s/n: ");
        resp = Leer.carácter();
        Leer.limpiar();
        if (resp == 'n') return;
        // Obtener un listado del directorio actual de trabajo
        obj = new File(System.getProperty("user.dir"));
        String[] nombresDir = obj.list();
        for (int i = 0; i < nombresDir.length; i++)
            flujoS.print(nombresDir[i] + ", ");
        flujoS.println("\n");
        objFichero = new File(Leer.dato());
    }
    artículos = new CBaseDeDatos(objFichero);
    ficheroAbierto = true;
}
```

Añadir un registro al fichero

El método *añadirReg* tiene como misión añadir un registro al final del fichero. Para ello, solicitará los datos a través del teclado y enviará al objeto *artículos* el

mensaje *añadir* (se ejecuta el método *añadir* de su clase) pasando como argumento el objeto *CRegistro* obtenido a partir de los datos leídos.

```
public static void añadirReg() throws IOException
{
    String referencia;
    double precio;

    flujoS.print("Referencia:    ");
    referencia = Leer.dato();
    flujoS.print("Precio:        ");
    precio = Leer.datoDouble();
    artículos.añadir(new CRegistro(referencia, precio));
}
```

Modificar un registro del fichero

El método *modificarReg* tiene como finalidad permitir modificar cualquier registro del fichero actual con el que estamos trabajando. Para ello, solicitará el número de registro a modificar, lo leerá, visualizará los campos correspondientes, y presentará un menú que permita modificar cualquiera de esos campos:

Modificar el dato:

1. Referencia
2. Precio
3. Salir y salvar los cambios
4. Salir sin salvar los cambios

Opción:

Finalmente, sólo si se eligió la opción 3, enviará al objeto *artículos* el mensaje *ponerValorEn* (se ejecuta el método *ponerValorEn* de su clase) pasando como argumento el objeto *CRegistro* obtenido a partir de los nuevos datos leídos.

```
public static void modificarReg() throws IOException
{
    String referencia;
    double precio;
    int op, nreg;

    // Solicitar el número de registro a modificar
    flujoS.print("Número de registro entre 0 y " +
                (artículos.longitud() - 1) + ": ");
    nreg = Leer.datToInt();

    // Leer el registro
    CRegistro obj = artículos.valorEn(nreg);
    if (obj == null) return;
```

```

// Visualizarlo
flujoS.println(obj.obtenerReferencia());
flujoS.println(obj.obtenerPrecio());

// Modificar el registro
do
{
    flujoS.print("\n\n");
    flujoS.println("Modificar el dato:");
    flujoS.println("1. Referencia");
    flujoS.println("2. Precio");
    flujoS.println("3. Salir y salvar los cambios");
    flujoS.println("4. Salir sin salvar los cambios");
    flujoS.println();
    flujoS.print("    Opción: ");
    op = Leer.datoInt();

    switch( op )
    {
        case 1: // modificar referencia
            flujoS.print("Referencia:    ");
            referencia = Leer.dato();
            obj.asignarReferencia(referencia);
            break;
        case 2: // modificar precio
            flujoS.print("Precio:    ");
            precio = Leer.datoDouble();
            obj.asignarPrecio(precio);
            break;
        case 3: // guardar los cambios
            break;
        case 4: // salir sin guardar los cambios
            break;
    }
}
while( op != 3 && op != 4);

if (op == 3) artículos.ponerValorEn(nreg, obj);
}

```

Eliminar un registro del fichero

El método *eliminarReg* permite marcar un registro del fichero como borrado. Para marcar un registro se enviará al objeto *artículos* el mensaje *eliminar* (se ejecuta el método *eliminar* de su clase) pasando como argumento su *referencia*, la cual se solicitará a través del teclado. Este método devolverá el mismo valor returned por el método *eliminar*: **true** si la operación se realiza satisfactoriamente y **false** en caso contrario.

```

public static boolean eliminarReg() throws IOException
{
    String referencia;
    flujoS.print("Referencia: "); referencia = Leer.dato();
    boolean eliminado = artículos.eliminar(referencia);
    if (eliminado)
        flujoS.println("registro eliminado");
    else
        if (artículos.longitud() != 0)
            flujoS.println("referencia no encontrada");
        else
            flujoS.println("lista vacía");
    return eliminado;
}

```

Visualizar registros del fichero

El método *visualizarRegs* se diseñará para que visualice el conjunto de registros cuyo campo *referencia* coincida o contenga la cadena/subcadena solicitada a través del teclado. Para ello, enviará al objeto *artículos* el mensaje *buscar* (se ejecuta el método *buscar* de su clase) pasando como argumentos la cadena/subcadena a buscar y el número de registro donde debe empezar la búsqueda (initialmente el cero), proceso que se repetirá utilizando el siguiente número de registro al último encontrado, mientras la búsqueda no falle.

```

public static void visualizarRegs() throws IOException
{
    int nreg = -1, nregs = artículos.longitud();
    if (nregs == 0)
        flujoS.println("lista vacía");

    flujoS.print("conjunto de caracteres a buscar ");
    String str = Leer.dato();
    CRegistro obj = null;

    do
    {
        nreg = artículos.buscar(str, nreg+1);
        if (nreg > -1)
        {
            obj = artículos.valorEn(nreg);
            flujoS.println("Registro: " + nreg);
            flujoS.println(obj.obtenerReferencia());
            flujoS.println(obj.obtenerPrecio());
            flujoS.println();
        }
    }
    while (nreg != -1);
}

```

```

    if (obj == null)
        flujoS.println("no se encontró ningún registro");
}

```

EJERCICIOS PROPUESTOS

1. Escribir una aplicación que permita escribir por la impresora un fichero de texto. La aplicación se ejecutará de la forma siguiente: *java imprimir fichero*, donde *imprimir* es el nombre de la aplicación y *fichero* el nombre del fichero de texto que se desea imprimir.
2. Realizar un programa que permita trabajar sobre un fichero que almacena los resultados obtenidos después de medir las temperaturas en un punto geográfico durante un intervalo de tiempo. El fichero constará de una cabecera definida según la siguiente estructura de datos:

```

public class Cabecera
{
    private class Posicion
    {
        // Atributos
        int grados, minutos;
        float segundos;
        // Métodos
        // ...
    }
    // Posición geográfica del punto
    private Posicion latitud;
    private Posicion longitud;
    private int total_muestras;
    // Métodos
    // ...
}

```

A continuación de la cabecera estarán especificadas todas las temperaturas. Cada una de ellas es un valor **float**.

El programa deberá permitir realizar las operaciones siguientes:

1. Fichero nuevo
2. Abrir fichero
3. Añadir temperatura
4. Modificar temperatura
5. Visualizar la temperatura media
6. Salir

Opción:

3. Suponga que disponemos de un fichero en disco llamado *alumnos*, donde cada registro se corresponde con los atributos de una clase como la siguiente:

```
public class CRegistro
{
    // Atributos
    private int númeroMatrícula;
    private String nombre;
    private String calificación;
    // Métodos
    // ...
}
```

La calificación viene dada por dos caracteres: SS (suspenso), AP (aprobado), NT (notable) y SB (sobresaliente). Realizar un programa que permita visualizar el % de los alumnos suspendidos, aprobados, notables y sobresalientes.

4. Suponga que disponemos en el disco dos ficheros denominados *alumnos* y *modificaciones*. La estructura de cada uno de los registros para ambos ficheros se corresponde con los atributos de una clase como la siguiente:

```
public class CRegistro
{
    // Atributos
    private String nombre;
    private float nota;
    // Métodos
    // ...
}
```

Suponga también que ambos ficheros están clasificados ascendentemente por el campo *nombre*.

En el fichero *modificaciones* se han grabado las modificaciones que posteriormente realizaremos sobre el fichero *alumnos*. En *modificaciones* hay como máximo un registro por alumno y se corresponden con:

- Registros que también están en el fichero *alumnos* pero que han variado en su campo *nota*.
- Registros nuevos; esto es, registros que no están en el fichero *alumnos*.
- Registros que también están en el fichero *alumnos* y que deseamos *eliminar*. Estos registros se distinguen porque su campo *nota* vale -1.

Se pide realizar un programa que permita obtener a partir de los ficheros *alumnos* y *modificaciones* un tercer fichero siguiendo los criterios de actualización anteriormente descritos. El fichero resultante terminará llamándose *alumnos*.

CAPÍTULO 13

© F.J.Ceballos/RA-MA

ESTRUCTURAS DINÁMICAS

La principal característica de las estructuras dinámicas es la facultad que tienen para variar su tamaño y hay muchos problemas que requieren de este tipo de estructuras. Esta propiedad las distingue claramente de las estructuras estáticas fundamentales como las matrices. Cuando se crea una matriz su número de elementos se fija en ese instante y después no puede agrandarse o disminuirse elemento a elemento, conservando el espacio actualmente asignado; en cambio, cuando se crea una estructura dinámica eso sí es posible.

Por tanto, no es posible asignar una cantidad fija de memoria para una estructura dinámica, y como consecuencia un compilador no puede asociar direcciones explícitas con las componentes de tales estructuras. La técnica que se utiliza más frecuentemente para resolver este problema consiste en realizar una asignación dinámica para las componentes individuales, al tiempo que son creadas durante la ejecución del programa, en vez de hacer la asignación de una sola vez para un número de componentes determinado.

Cuando se trabaja con estructuras dinámicas, el compilador asigna una cantidad fija de memoria para mantener la dirección del componente asignado dinámicamente, en vez de hacer una asignación para el componente en sí. Esto implica que debe haber una clara distinción entre datos y referencias a datos, y que consecuentemente se deben emplear tipos de datos cuyos valores sean referencias a otros datos.

Cuando se asigna memoria dinámicamente para un objeto de un tipo cualquiera, se devuelve una referencia a la zona de memoria asignada. Para realizar esta operación disponemos en Java del operador **new** (vea en el capítulo 4, el apartado "Crear un objeto de una clase").

Este capítulo introduce técnicas en programación orientada a objetos para construir estructuras abstractas de datos. Una vez que haya trabajado los ejemplos de este capítulo, será capaz de explotar en sus aplicaciones la potencia de las listas enlazadas, pilas, colas y árboles binarios.

LISTAS LINEALES

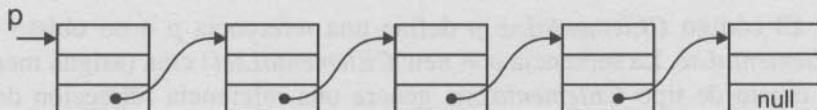
Hasta ahora hemos trabajado con matrices que como sabemos son colecciones de elementos todos del mismo tipo, ubicados en memoria uno a continuación de otro; el número de elementos es fijado en el instante de crear la matriz. Si más adelante, durante la ejecución del programa, necesitáramos modificar su tamaño para que contenga más o menos elementos, la única alternativa posible sería asignar un nuevo espacio de memoria del tamaño requerido y además, copiar en él los datos que necesitemos conservar de la matriz original. La nueva matriz pasará a ser la matriz actual y la matriz origen se destruirá, si ya no fuera necesaria.

Es evidente que cada vez que necesitemos añadir o eliminar un elemento a una colección de elementos, la solución planteada en el párrafo anterior no es la más idónea; seguro que estamos pensando en algún mecanismo que nos permita añadir un único elemento a la colección, o bien eliminarlo. Este mecanismo es viable si en lugar de trabajar con matrices lo hacemos con listas lineales. Una lista lineal es una colección, originalmente vacía, de elementos u objetos de cualquier tipo no necesariamente consecutivos en memoria, que durante la ejecución del programa puede crecer o decrecer elemento a elemento según las necesidades previstas en el mismo.

Según la definición dada surge una pregunta: si los elementos no están consecutivos en memoria ¿cómo pasamos desde un elemento al siguiente cuando recorramos la lista? La respuesta es que cada elemento debe almacenar información de dónde está el siguiente elemento o el anterior, o bien ambos. En función de la información que cada elemento de la lista almacene respecto a la localización de sus antecesores y/o predecesores, las listas pueden clasificarse en: listas simplemente enlazadas, listas circulares, listas doblemente enlazadas y listas circulares doblemente enlazadas.

Listas lineales simplemente enlazadas

Una *lista lineal simplemente enlazada* es una colección de objetos (elementos de la lista), cada uno de los cuales contiene datos o una referencia a los datos, y una referencia al siguiente objeto en la colección (elemento de la lista). Gráficamente puede representarse de la forma siguiente:



Lista lineal

Para construir una lista lineal, primero tendremos que definir el tipo de los elementos que van a formar parte de la misma. Por ejemplo, cada elemento de la lista puede definirse como una estructura de datos con dos miembros: una referencia al elemento siguiente y una referencia al área de datos. El área de datos puede ser de un tipo predefinido o de un tipo definido por el usuario. Según esto, el tipo de cada elemento de una lista puede venir definido de la forma siguiente:

```
class CElementoLse
{
    // Atributos
    // Defina aquí los datos o las referencias a los datos
    // ...
    CElementoLse siguiente; // referencia al siguiente elemento

    // Métodos
    CElementoLse() {} // constructor sin parámetros
    // ...
}
```

Se puede observar que la clase *CElementoLse* definirá una serie de atributos correspondientes a los datos que deseemos manipular, además de un atributo especial, denominado *siguiente*, para permitir que cada elemento pueda referenciar a su sucesor formando así una lista enlazada.

Una vez creada la clase de objetos *CElementoLse* la asignación de memoria para un elemento se haría así:

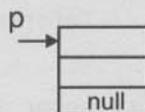
```
public class Test
{
    public static void main(String[] args)
    {
        CElementoLse p; // referencia a un elemento
        // Asignar memoria para un elemento
        p = new CElementoLse();
        // Este elemento no tiene un sucesor
        p.siguiente = null;
        // Operaciones cualesquiera
        // Permitir que se libere la memoria ocupada por el elemento p
        p = null;
    }
}
```

El código *CElementoLse* *p* define una referencia *p* a un objeto de la clase *CElementoLse*. La sentencia *p = new CElementoLse()* crea (asigna memoria para) un objeto de tipo *CElementoLse*, genera una referencia (dirección de memoria) que direcciona este nuevo objeto y asigna esta referencia a la variable *p*. La sentencia *p.sigiente = null* asigna al miembro *siguiente* del objeto referenciado por *p* el valor **null**, indicando así que después de este elemento no hay otro; esto es, que este elemento es el último de la lista.

El valor **null**, referencia nula, permite crear estructuras de datos finitas. Así mismo, suponiendo que *p* hace referencia al principio de la lista, diremos que dicha lista está vacía si *p* vale **null**. Por ejemplo, después de ejecutar las sentencias:

```
p = null;           // lista vacía
p = new CElementoLse(); // elemento p
p.sigiente = null; // no hay siguiente elemento
```

tenemos una lista de un elemento:

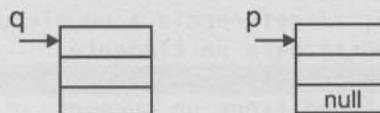


Para añadir un nuevo elemento a la lista, procederemos así:

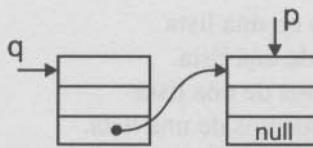
```
q = new CElementoLse(); // crear un nuevo elemento
q.sigiente = p; // almacenar la localización del elemento siguiente
p = q; // p hace referencia al principio de la lista
```

donde *q* es una referencia a un objeto de tipo *CElementoLse*. Ahora tenemos una lista de dos elementos. Observe que los elementos nuevos se añaden al principio de la lista.

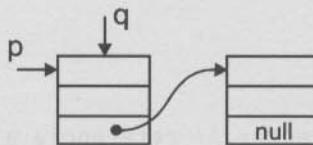
Para verlo con claridad analicemos las tres sentencias anteriores. Partimos de que tenemos una lista referenciada por *p* con un solo elemento. La sentencia *q = new CElementoLse()* crea un nuevo elemento:



La sentencia *q.sigiente = p* hace que el sucesor del elemento creado sea el anteriormente creado. Observe que ahora *q.sigiente* y *p* tienen el mismo valor; esto es, la misma dirección, por lo tanto, referencian el mismo elemento:



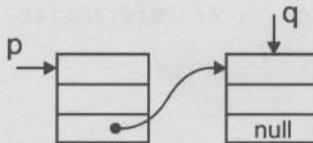
Por último, la sentencia $p = q$ hace que la lista quede de nuevo referenciada por p ; es decir, para nosotros p es siempre el primer elemento de la lista.



Ahora p y q referencian al mismo elemento, al primero. Si ahora se ejecutara una sentencia como la siguiente ¿qué sucedería?

```
q = q.sigiente;
```

¿Quién es $q.sigiente$? Es el atributo *siguiente* del objeto referenciado por q que contiene la dirección de memoria donde se localiza el siguiente elemento al referenciado por p . Si este valor se lo asignamos a q , entonces q referenciará al mismo elemento que referenciaba $q.sigiente$. El resultado es que q referencia ahora al siguiente elemento como se puede ver en la figura mostrada a continuación:



Esto nos da una idea de cómo avanzar elemento a elemento sobre una lista. Si ejecutamos de nuevo la misma sentencia:

```
q = q.sigiente;
```

¿Qué sucede? Sucede que como $q.sigiente$ vale **null**, a q se le ha asignado el valor **null**. Conclusión, cuando en una lista utilizamos una referencia para ir de un elemento al siguiente, en el ejemplo anterior q , diremos que hemos llegado al final de la lista cuando q toma el valor **null**.

Operaciones básicas

Las operaciones que podemos realizar con listas incluyen fundamentalmente las siguientes:

1. Insertar un elemento en una lista.
2. Borrar un elemento de una lista.
3. Recorrer los elementos de una lista.
4. Borrar todos los elementos de una lista.
5. Buscar un elemento en una lista.

Partiendo de las definiciones:

```
class CElementoLse
{
    // Atributos
    int dato;
    CElementoLse siguiente; // referencia al siguiente elemento

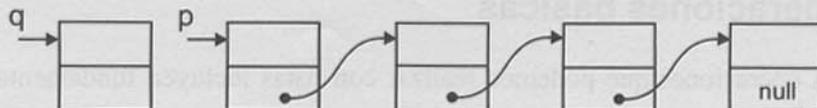
    // Métodos
    CElementoLse() {} // constructor sin parámetros
    CElementoLse( int d ) // constructor con parámetros
    {
        dato = d;
    }
}
public class Test
{
    public static void main( String[] args )
    {
        CElementoLse p, q, r; // referencias
        // ...
    }
}
```

vamos a exponer en los siguientes apartados cómo realizar cada una de las operaciones básicas. Observe que por sencillez vamos a trabajar con una lista de enteros.

Inserción de un elemento al comienzo de la lista

Supongamos una lista lineal referenciada por *p*. Para insertar un elemento al principio de la lista, primero se crea el elemento y después se reasignan las referencias, tal como se indica a continuación:

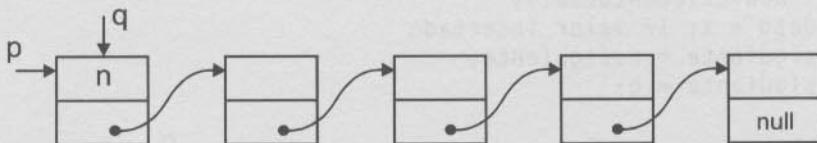
```
q = new CElementoLse();
```



```

q.dato = n;      // asignación de valores
q.siguiente = p; // reasignación de referencias
p = q;
  
```

El orden en el que se realizan estas operaciones es esencial. El resultado es:



Esta operación básica nos sugiere cómo crear una lista. Para ello, y partiendo de una lista vacía, no tenemos más que repetir la operación de insertar un elemento al comienzo de una lista, tantas veces como elementos deseemos que tenga dicha lista. Veámoslo a continuación:

```

///////////////////////////////
// Crear una lista lineal simplemente enlazada
//
public class Test
{
    public static void main(String[] args)
    {
        CElementoLse p, q; // referencias
        int n, eof = Integer.MIN_VALUE;

        // Crear una lista de enteros
        System.out.println("Introducir datos. Finalizar con Ctrl+Z.");

        p = null; // lista vacía

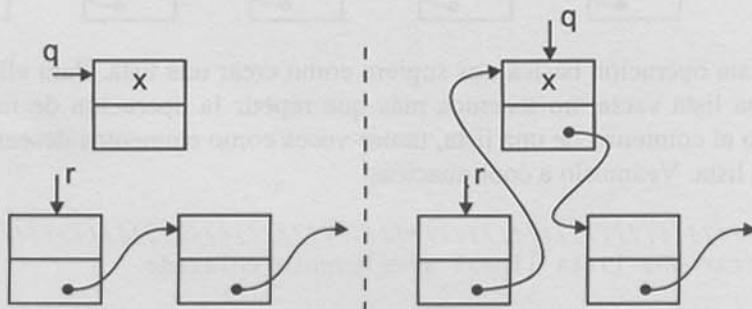
        System.out.print("dato: ");
        while ((n = Leer.datoInt()) != eof)
        {
            q = new CElementoLse();
            q.dato = n;
            q.siguiente = p;
            p = q;
            System.out.print("dato: ");
        }
    }
}
  
```

Notar que el orden de los elementos en la lista, es inverso al orden en el que han llegado. Así mismo, como es ya habitual, utilizamos la clase *Leer* diseñada en el capítulo 5 y revisada en el 11 y 12, para leer datos desde el teclado.

Inserción de un elemento en general

La inserción de un elemento en la lista, a continuación de otro elemento cualquiera referenciado por r , es de la forma siguiente:

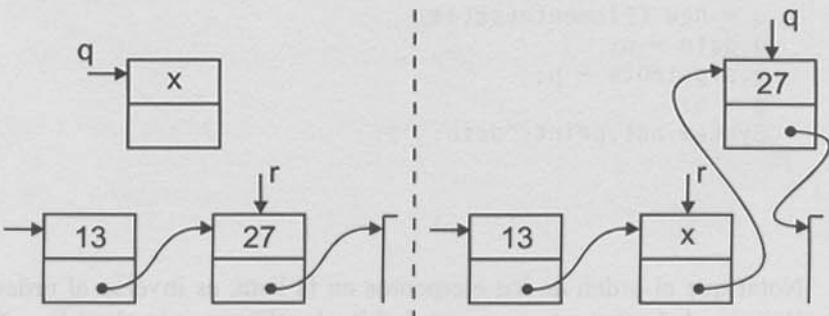
```
q = new CElementoLse();
q.dato = x; // valor insertado
q.siguiente = r.siguiente;
r.siguiente = q;
```



Inserción en la lista detrás del elemento referenciado por r

La inserción de un elemento en la lista antes de otro elemento referenciado por r , se hace insertando un nuevo elemento detrás del elemento referenciado por r , intercambiando previamente los valores del nuevo elemento y del elemento referenciado por r .

```
q = new CElementoLse();
q.dato = r.dato;      // copiar miembro a miembro un objeto en otro
q.siguiente = r.siguiente;
r.dato = x;           // valor insertado
r.siguiente = q;
```

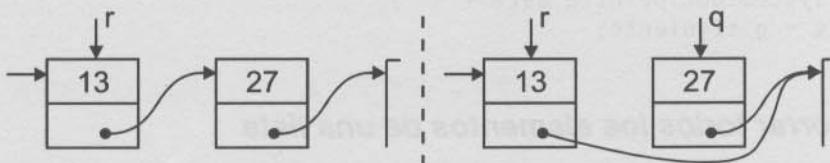


Inserción en la lista antes del elemento referenciado por r

Borrar un elemento de la lista

Para borrar el sucesor de un elemento referenciado por r , las operaciones a realizar son las siguientes:

```
q = r.siguiente;           // q referencia el elemento a borrar
r.siguiente = q.siguiente; // enlazar los elementos anterior
                          // y posterior al borrado
q = null; // objeto referenciado por q a la basura (borrar)
```

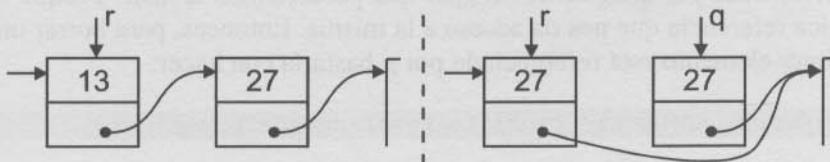


Borrar el sucesor del elemento referenciado por r

Un objeto es enviado a la basura para ser recogido por el recolector de basura de Java, sólo cuando se eliminan todas las referencias que permiten acceder al mismo.

Observe que para acceder a los miembros de un elemento, éste tiene que estar referenciado por una variable. Por esta razón, lo primero que hemos hecho ha sido referenciar el elemento a borrar por q .

Para borrar un elemento referenciado por r , las operaciones a realizar son las siguientes:



Borrar el elemento referenciado por r

```
q = r.siguiente;
r.dato = q.dato; // copiar miembro a miembro un objeto en otro
r.siguiente = q.siguiente;
q = null;         // objeto referenciado por q a la basura (borrar)
```

Como ejercicio, escribir la secuencia de operaciones que permitan borrar el último elemento de una lista.

Recorrer una lista

Supongamos que hay que realizar una operación con todos los elementos de una lista, cuyo primer elemento está referenciado por *p*. Por ejemplo, escribir el valor de cada elemento de la lista. La secuencia de operaciones sería la siguiente:

```
q = p; // salvar la referencia al primer elemento de la lista
while (q != null)
{
    System.out.print(q.dato + " ");
    q = q.siguiente;
}
```

Borrar todos los elementos de una lista

Borrar todos los elementos de una lista equivale a enviar a la basura a cada uno de los elementos de la misma. Supongamos que queremos borrar una lista, cuyo primer elemento está referenciado por *p*. La secuencia de operaciones es la siguiente:

```
q = p;           // q referencia el primer elemento de la lista
while (q != null)
{
    p = p.siguiente; // p referencia al siguiente elemento
    q = null;        // objeto referenciado por q a la basura
    q = p;           // q hace referencia al mismo elemento que p
}
```

Observe que antes de borrar el elemento referenciado por *q*, hacemos que *p* referencia al siguiente elemento, porque si no perderíamos el resto de la lista; la referenciada por *q.siguiente*. Y ¿por qué perderíamos la lista? Porque se pierde la única referencia que nos da acceso a la misma. Entonces, para borrar un lista cuyo primer elemento está referenciado por *p* bastaría con hacer:

```
p = null; // borrar la lista referenciada por p
```

Evidentemente, el proceso anterior no es necesario. Para eliminar una lista basta con poner a **null** la variable que hace referencia al primer elemento de la misma, porque esto implica que todos los elementos de ella queden desreferenciados y sean enviados a la basura para ser recogidos por el recolector de basura.

Buscar en una lista un elemento con un valor x

Supongamos que queremos buscar un determinado elemento en una lista cuyo primer elemento está referenciado por *p*. La búsqueda es secuencial y termina cuando se encuentra el elemento, o bien cuando se llega al final de la lista.

```

q = p;           // q referencia el primer elemento de la lista
System.out.print("dato a buscar: "); x = Leer.datoInt();
while (q != null && q.dato != x)
    q = q.siguiente; // q referencia al siguiente elemento

```

Observe el orden de las expresiones que forman la condición del bucle **while**. Sabemos que en una operación **&&** (AND), cuando una de las expresiones es falsa la condición ya es falsa, por lo que el resto de las expresiones no necesitan ser evaluadas. De ahí que cuando *q* valga **null** si la expresión *q.dato* fuera evaluada, Java lanzaría una excepción **NullPointerException**.

UNA CLASE PARA LISTAS LINEALES

Basándonos en las operaciones básicas sobre listas lineales descritas anteriormente, vamos a escribir a continuación una clase que permita crear una lista lineal simplemente enlazada en la que cada elemento conste de dos miembros: un valor real de tipo **double** y una referencia a un elemento del mismo tipo.

La clase la denominaremos *CListaLinealSE* (*Clase Lista Lineal Simplemente Enlazada*). Dicha clase incluirá un atributo *p* para almacenar de forma permanente una referencia al primer elemento de la lista, y una clase interna *CElemento* que definirá la estructura de un elemento de la lista, que según hemos indicado anteriormente será así:

```

private class CElemento
{
    // Atributos
    private double dato;
    private CElemento siguiente; // siguiente elemento

    // Métodos
    private CElemento() {} // constructor
}

```

Finalmente, para simplificar, la interfaz pública de la clase *CListaLinealSE* proporcionará solamente tres métodos: un constructor sin parámetros, *añadirAlPrincipio* y *mostrarTodos*.

El constructor dará lugar a una lista vacía. El método *añadirAlPrincipio* permitirá añadir un nuevo elemento al principio de la lista, en nuestro caso un valor de tipo **double** recibido como parámetro por el método, y *mostrarTodos* permitirá visualizar por pantalla todos los elementos de la lista, en nuestro caso la lista de valores de tipo **double** que almacena.

Según lo expuesto, *CListaLinealSE* será así:

```
///////////
// Lista lineal simplemente enlazada
//
public class CListaLinealSE
{
    // p: referencia al primer elemento de la lista
    private CElemento p = null;

    // Elemento de una lista lineal simplemente enlazada
    private class CElemento
    {
        // Atributos
        private double dato;
        private CElemento siguiente; // siguiente elemento
        // Métodos
        private CElemento() {} // constructor
    }

    public CListaLinealSE() {} // constructor

    // Añadir un elemento al principio de la lista
    public void añadirAlPrincipio(double n)
    {
        CElemento q = new CElemento();
        q.dato = n;      // asignación de valores
        q.siguiente = p; // reasignación de referencias
        p = q;
    }

    public void mostrarTodos()
    {
        // Recorrer la lista
        CElemento q = p; // referencia al primer elemento
        while (q != null)
        {
            System.out.print(q.dato + " ");
            q = q.siguiente;
        }
    }
}
/////////
```

Apoyándonos en esta clase, vamos a escribir una aplicación basada en una clase *Test* que cree una lista lineal simplemente enlazada que almacene una serie de valores de tipo **double** introducidos desde el teclado. Finalmente, para verificar que todo ha sucedido como esperábamos, mostraremos la lista de valores.

Para llevar a cabo lo expuesto, el método **main** de esta aplicación realizará tres cosas:

1. Definirá un objeto *lse* de la clase *CListaLinealSE*.
2. Solicitará datos de tipo **double** del teclado y los añadirá a la lista, para lo cual enviará al objeto *lse* el mensaje *añadirAlPrincipio* por cada dato que añada.
3. Mostrará la lista de datos, para lo cual enviará al objeto *lse* el mensaje *mostrarTodos*.

```
///////////////////////////////
// Crear una lista lineal simplemente enlazada
//
public class Test
{
    public static void main(String[] args)
    {
        // Crear una lista lineal vacía
        CListaLinealSE lse = new CListaLinealSE();

        // Leer datos reales y añadirlos a la lista
        double n;
        boolean eof = true;

        System.out.println("Introducir datos. Finalizar con Ctrl+Z.");
        System.out.print("dato: ");
        while (Double.isNaN(n = Leer.datoDouble()) != eof)
        {
            lse.añadirAlPrincipio(n);
            System.out.print("dato: ");
        }

        // Mostrar la lista de datos
        System.out.println();
        lse.mostrarTodos();
    }
}
```

Si en un instante determinado necesitara borrar todos los elementos de la lista, bastaría con escribir *lse = null*.

Con el fin de acercarnos más a la realidad de cómo debe ser la clase *CListaLinealSE*, vamos a sustituir el método *mostrarTodos* por otro método *obtener* que devuelva el valor almacenado en el elemento *i* de la lista. De esta forma, será la aplicación que utilice la clase *CListaLinealSE* la que decida qué hacer con el valor retornado (imprimirllo, acumularlo, etc.).

El método *obtener* recibirá como parámetro la posición del elemento que se desea obtener (la primera posición es la cero) y devolverá como resultado el dato almacenado por este elemento, o bien el valor **NaN** si la lista está vacía o la posición especificada está fuera de límites.

```

public double obtener(int i)
{
    if (p == null)
    {
        System.err.println("lista vacía");
        return Double.NaN;
    }

    CElemento q = p; // referencia al primer elemento
    if (i >= 0)
    {
        // Posicionarse en el elemento i
        for (int n = 0; q != null && n < i; n++)
            q = q.siguiente;

        // Retornar el dato
        if (q != null) return q.dato;
    }

    // Índice fuera de límites
    return Double.NaN;
}

```

Ahora, para que el método **main** de la aplicación anterior muestre los datos utilizando este método, tenemos que reescribir la parte del mismo que realizaba este proceso, como se indica a continuación:

```

public static void main(String[] args)
{
    // ...

    // Mostrar la lista de datos
    System.out.println();
    int i = 0;
    double d = lse.obtener(i);
    while (!Double.isNaN(d))
    {
        System.out.print(d + " ");
        i++;
        d = lse.obtener(i);
    }
}

```

Lo que hace el segmento de código mostrado es obtener y visualizar los valores de los elementos 0, 1, 2, ... de la lista *lse* hasta que el método *obtener* devuelva el valor **NaN**, señal de que se ha llegado al final de la lista.

Clase genérica para listas lineales

La clase *CListaLinealSE* implementada anteriormente ha sido diseñada para manipular listas de un tipo específico de elementos: datos de tipo **double**. No cabe duda que esta clase tendría un mayor interés para el usuario si estuviera diseñada para permitir listas de objetos de cualquier tipo. Ésta es la dirección en la que vamos a trabajar a continuación. En otros lenguajes como C++, esto se hace utilizando plantillas. En Java se hace utilizando la clase **Object**.

Sabemos que **Object** es la superclase de todas las clases; esto es, cuando implementamos una clase y no se especifica explícitamente su superclase, dicha clase está derivada de **Object**. Esto significa que las dos definiciones de clase siguientes son equivalentes:

```
public class CListaLinealSE { ... }

public class CListaLinealSE extends Object { ... }
```

También sabemos que Java permite convertir implícitamente una referencia a un objeto de una subclase en una referencia a su superclase directa o indirecta. Por ejemplo, la siguiente línea convierte una referencia a un objeto de la clase **Double** a una referencia a su superclase **Object**. Este ejemplo puede extenderse a cualquier clase de la biblioteca de Java o definida por el usuario.

```
Object datos = new Double(n);
```

Según esto, para que la clase *CListaLinealSE* permita listas de objetos de cualquier tipo, basta con que su clase interna *CElemento* (clase de cada uno de los elementos de la lista) tenga un atributo que sea una referencia de tipo **Object**. Un atributo así definido puede referenciar cualquier objeto de cualquier clase.

Esta modificación implica dos cambios más: el parámetro del método *añadir-AlPrincipio* tiene que ser ahora de tipo **Object**, y el método *obtener* tiene que devolver ahora una referencia de tipo **Object**.

```
///////////////////////////////
// Lista lineal simplemente enlazada
//
public class CListaLinealSE
{
    // p: referencia al primer elemento de la lista
    private CElemento p = null;

    // Elemento de una lista lineal simplemente enlazada
```

```

private class CElemento
{
    // Atributos
    private Object datos;
    private CElemento siguiente; // siguiente elemento
    // Métodos
    private CElemento() {}      // constructor
}

public CListaLinealSE() {}      // constructor

// Añadir un elemento al principio de la lista
public void añadirAlPrincipio(Object obj)
{
    CElemento q = new CElemento();
    q.datos = obj; // asignación de valores
    q.siguiente = p; // reasignación de referencias
    p = q;
}

public Object obtener(int i)
{
    if (p == null)
    {
        System.err.println("lista vacía");
        return null;
    }

    CElemento q = p; // referencia al primer elemento
    if (i >= 0)
    {
        // Posicionarse en el elemento i
        for (int n = 0; q != null && n < i; n++)
            q = q.siguiente;
        // Retornar los datos
        if (q != null) return q.datos;
    }
    // Índice fuera de límites
    return null;
}
}
/////////////////////////////////////////////////////////////////

```

Veamos ahora en qué se modifica la aplicación *Test* que creaba una lista de valores de tipo **double** introducidos desde el teclado. Igual que antes, el método **main** de *Test* realizará tres cosas:

1. Definirá un objeto *lse* de la clase *CListaLinealSE*.

```
CListaLinealSE lse = new CListaLinealSE();
```

2. Solicitará datos de tipo **double** del teclado y los añadirá a la lista, para lo cual enviará al objeto *lse* el mensaje *añadirAlPrincipio* por cada dato que añada. Pero como *añadirAlPrincipio* tiene un parámetro de tipo **Object**, el argumento pasado tiene que ser un objeto; en este caso un objeto que encapsule un valor de tipo **double**. Estos objetos son construidos a partir de la clase **Double** del paquete **java.lang**.

```
lse.añadirAlPrincipio(new Double(n));
```

3. Mostrará la lista de valores, para lo cual enviará al objeto *lse* el mensaje *obtener* por cada uno de los elementos de la lista. El método *obtener* devuelve una referencia de tipo **Object** que, en nuestro caso, apunta a un objeto de su subclase **Double**. Pero, si accedemos a un objeto de una subclase por medio de una referencia a su superclase, ese objeto sólo puede ser manipulado por los métodos de su superclase. Tendremos entonces que convertir la referencia de tipo **Object** a tipo **Double**. Según estudiamos en el capítulo 10, se trata de una conversión explícita de una referencia del tipo de una superclase a una referencia a una de sus subclases, lo cual sólo puede hacerse si el objeto referenciado es del tipo de la subclase, condición que en nuestro ejemplo se cumple, ya que el objeto es de la clase **Double**.

```
Double d = (Double)lse.obtener(i);
```

A continuación se muestra la aplicación *Test* completa:

```
///////////////////////////////
// Crear una lista lineal simplemente enlazada
//
public class Test
{
    public static void main(String[] args)
    {
        // Crear una lista lineal vacía
        CListaLinealSE lse = new CListaLinealSE();

        // Leer datos reales y añadirlos a la lista
        double n;
        boolean eof = true;
        System.out.println("Introducir datos. Finalizar con Ctrl+Z.");
        System.out.print("dato: ");
        while (Double.isNaN(n = Leer.datoDouble()) != eof)
        {
            lse.añadirAlPrincipio(new Double(n));
            System.out.print("dato: ");
        }

        // Mostrar la lista de datos
        System.out.println();
```

```
int i = 0;
Double d = (Double)lse.obtener(i);
while (d != null)
{
    System.out.print(d.doubleValue() + " ");
    i++;
    d = (Double)lse.obtener(i);
}
```

Para finalizar, vamos a completar la clase *CListaLinealSE* con otros métodos de interés que especificamos en la tabla siguiente:

Método	Significado
tamaño	Devuelve el número de elementos de la lista. No tiene parámetros.
añadir	Añade un elemento en la posición <i>i</i> . Tiene dos parámetros: posición <i>i</i> y una referencia al objeto a añadir. Devuelve true si la operación se ejecuta satisfactoriamente y false en caso contrario.
añadirAlPrincipio	Añade un elemento al principio. Tiene un parámetro que es una referencia al objeto a añadir. Devuelve true o false , igual que <i>añadir</i> .
añadirAlFinal	Añade un elemento al final. Tiene un parámetro que es una referencia al objeto a añadir. Devuelve true o false , igual que <i>añadir</i> .
borrar	Borra el elemento de la posición <i>i</i> . Tiene un parámetro que indica la posición <i>i</i> del objeto a borrar. Devuelve una referencia al objeto borrado o null si la lista está vacía o el índice está fuera de límites.
borrarPrimero	Borra el primer elemento. No tiene parámetros. Devuelve una referencia al objeto borrado o null si la lista está vacía.
borrarÚltimo	Borra el último elemento. No tiene parámetros. Devuelve una referencia al objeto borrado o null si la lista está vacía.
obtener	Devuelve el elemento de la posición <i>i</i> , o bien null si la lista está vacía o el índice está fuera de límites. Tiene un parámetro que se corresponde con la posición <i>i</i> del objeto que se desea obtener.
obtenerPrimero	Devuelve el primer elemento, o bien null si la lista está vacía.
obtenerÚltimo	Devuelve el último elemento, o bien null si la lista está vacía.

A continuación se muestra el código completo de la clase *CListaLinealSE*. Observar que además de los métodos especificados en la tabla anterior, se ha añadido a la clase *CElemento* un constructor con parámetros.

```
///////////////////////////////
// Lista lineal simplemente enlazada
//
public class CListaLinealSE
{
    // p: referencia al primer elemento de la lista.
    // Es el elemento de cabecera.
    private CElemento p = null;

    // Elemento de una lista lineal simplemente enlazada
    private class CElemento
    {
        // Atributos
        private Object datos;
        private CElemento siguiente; // siguiente elemento
        // Métodos
        private CElemento() {} // constructor
        private CElemento(Object d, CElemento s) // constructor
        {
            datos = d;
            siguiente = s;
        }
    }

    public CListaLinealSE() {} // constructor

    public int tamaño()
    {
        // Devuelve el número de elementos de la lista
        CElemento q = p; // referencia al primer elemento
        int n = 0; // número de elementos
        while (q != null)
        {
            n++;
            q = q.siguiente;
        }
        return n;
    }

    public boolean añadir(int i, Object obj)
    {
        // Añadir un elemento en la posición i
        int numeroDeElementos = tamaño();
        if (i > numeroDeElementos || i < 0)
        {
            System.err.println("índice fuera de límites");
        }
    }
}
```

```
    return false;
}

// Crear el elemento a añadir
CElemento q = new CElemento(obj, null);

// Si la lista referenciada por p está vacía, añadirlo sin más
if (númeroDeElementos == 0)
{
    // Añadir el primer elemento
    p = q;
    return true;
}

// Si la lista no está vacía, encontrar el punto de inserción
CElemento elemAnterior = p;
CElemento elemActual = p;
// Posicionarse en el elemento i
for (int n = 0; n < i; n++)
{
    elemAnterior = elemActual;
    elemActual = elemActual.siguiente;
}
// Dos casos:
// 1) Insertar al principio de la lista
// 2) Insertar después del anterior (incluye insertar al final)
if (elemAnterior == elemActual) // insertar al principio
{
    q.siguiente = p;
    p = q; // cabecera
}
else // insertar después del anterior
{
    q.siguiente = elemActual;
    elemAnterior.siguiente = q;
}
return true;
}

public boolean añadirAlPrincipio(Object obj)
{
    // Añadir un elemento al principio
    return añadir(0, obj);
}

public boolean añadirAlFinal(Object obj)
{
    // Añadir un elemento al final
    return añadir(tamaño(), obj);
}
```

```
public Object borrar(int i)
{
    // Borrar el elemento de la posición i
    int númeroDeElementos = tamaño();
    if (i >= númeroDeElementos || i < 0)
        return null;

    // Entrar en la lista y encontrar el índice del elemento
    CElemento elemAnterior = p;
    CElemento elemActual = p;
    // Posicionarse en el elemento i
    for (int n = 0; n < i; n++)
    {
        elemAnterior = elemActual;
        elemActual = elemActual.siguiente;
    }
    // Dos casos:
    // 1) Borrar el primer elemento de la lista
    // 2) Borrar el siguiente a elemAnterior (elemActual)
    if (elemActual == p) // 1)
        p = p.siguiente; // cabecera
    else // 2)
        elemAnterior.siguiente = elemActual.siguiente;

    return elemActual.datos; // retornar el elemento borrado.

    // El elemento referenciado por elemActual será enviado a la
    // basura (borrado) al quedar desreferenciado, por ser
    // elemActual una variable local.
}

public Object borrarPrimero()
{
    // Borrar el primer elemento
    return borrar(0);
}

public Object borrarÚltimo()
{
    // Borrar el último elemento
    return borrar(tamaño() - 1);
}

public Object obtener(int i)
{
    // Obtener el elemento de la posición i
    int númeroDeElementos = tamaño();
    if (i >= númeroDeElementos || i < 0)
        return null;
```

```
CElemento q = p; // referencia al primer elemento
// Posicionarse en el elemento i
for (int n = 0; n < i; n++)
    q = q.siguiente;
// Retornar los datos
return q.datos;
}

public Object obtenerPrimero()
{
    // Retornar el primer elemento
    return obtener(0);
}

public Object obtenerUltimo()
{
    // Retornar el último elemento
    return obtener(tamaño() - 1);
}
```

Como ejercicio, supongamos que deseamos crear una lista lineal simplemente enlazada con la intención de almacenar los nombres de los alumnos de un determinado curso y sus notas de la asignatura de Programación. Según este enunciado ¿a qué tipo de objeto hará referencia cada elemento de la lista? Pues, a objetos cuya estructura interna sea capaz de almacenar un nombre (dato de tipo **String**) y una nota (dato de tipo **double**). Además, estos objetos podrán recibir una serie de mensajes con la intención de extraer o modificar su contenido. La clase representativa de los objetos descritos la vamos a denominar *CDatos* y puede escribirse de la forma siguiente:

```
public class CDatos
{
    // Atributos
    private String nombre;
    private double nota;

    // Métodos
    public CDatos() {}          // constructor sin parámetros
    public CDatos(String nom, double n) // constructor con parámetros
    {
        nombre = nom;
        nota = n;
    }

    public void asignarNombre(String nom)
    {

```

```

        nombre = nom;
    }

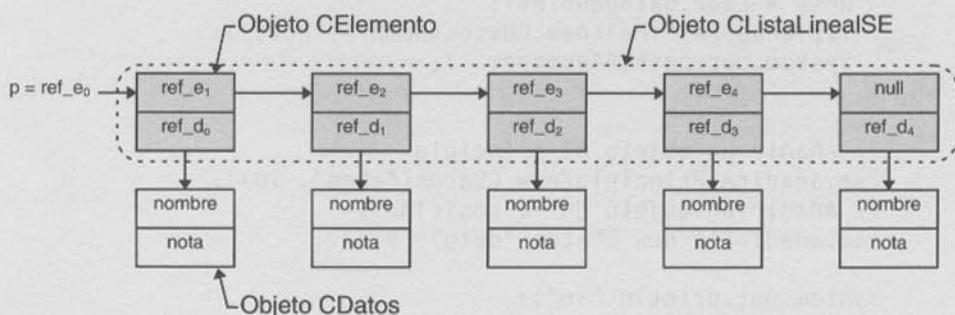
    public String obtenerNombre()
    {
        return nombre;
    }

    public void asignarNota(double n)
    {
        nota = n;
    }

    public double obtenerNota()
    {
        return nota;
    }
}

```

Sólo nos queda realizar una aplicación que utilizando las clases *CListaLinealSE* y *CDatos* cree una lista lineal y ponga en práctica las distintas operaciones que sobre ella pueden realizarse. La figura siguiente muestra de forma gráfica la estructura de datos que queremos construir. Observe que, en realidad, la lista lo que mantiene son referencias a los datos (objetos *CDatos*) y no los datos en sí, aunque, por sencillez, también resulta aceptable pensar que éstos forman parte de la lista lineal. La variable *p* es una referencia (*ref_e₀*) al elemento de índice 0; este elemento mantiene una referencia (*ref_e₁*) al elemento de la lista de índice 1 y una referencia (*ref_d₀*) al objeto de datos correspondiente, y así sucesivamente.



El código de la aplicación *Test* que se muestra a continuación enseña cómo crear y manipular una estructura de datos como la de la figura anterior:

```
///////////
// Crear una lista lineal simplemente enlazada
//
```

```
public class Test
{
    public static void mostrarLista(CListaLinealSE lse)
    {
        // Mostrar todos los elementos de la lista
        int i = 0, tam = lse.size();
        CDatos obj;
        while (i < tam)
        {
            obj = (CDatos)lse.obtener(i);
            System.out.println(i + ".- " + obj.obtenerNombre() + " " +
                               obj.obtenerNota());
            i++;
        }
    }

    public static void main(String[] args)
    {
        // Crear una lista lineal vacía
        CListaLinealSE lse = new CListaLinealSE();

        // Leer datos y añadirlos a la lista
        String nombre;
        double nota;
        int i = 0;
        System.out.println("Introducir datos. Finalizar con Ctrl+Z.");
        System.out.print("nombre: ");
        while ((nombre = Leer.dato()) != null)
        {
            System.out.print("nota:   ");
            nota = Leer.datoDouble();
            lse.añadirAlFinal(new CDatos(nombre, nota));
            System.out.print("nombre: ");
        }

        // Añadir un objeto al principio
        lse.añadirAlPrincipio(new CDatos("abcd", 10));
        // Añadir un objeto en la posición 1
        lse.añadir(1, new CDatos("defg", 9.5));

        System.out.println("\n");
        // Mostrar el primero
        CDatos obj = (CDatos)lse.obtenerPrimero();
        System.out.println("Primero: " + obj.obtenerNombre() + " " +
                           obj.obtenerNota());

        // Mostrar el último
        obj = (CDatos)lse.obtenerÚltimo();
```

```
System.out.println("Último: " + obj.obtenerNombre() + " " +  
    obj.obtenerNota());  
// Mostrar todos  
System.out.println("Lista:");  
mostrarLista(lse);  
  
// Borrar el elemento de índice 2  
obj = (CDatos)lse.borrar(2);  
if (obj == null)  
    System.out.println("Error: elemento no borrado");  
  
// Modificar el elemento de índice 1  
obj = (CDatos)lse.obtener(1);  
obj.asignarNota(9);  
  
// Mostrar todos  
System.out.println("Lista:");  
mostrarLista(lse);
```

Clase LinkedList

La clase **LinkedList** pertenece a la biblioteca de Java; está incluida en el paquete **java.util**. Tiene unas características similares a nuestra clase *CListaLinealSE*. La tabla siguiente muestra algunos de los métodos proporcionados por esta clase:

Método	Significado
<code>LinkedList</code>	Es el constructor de la clase. Tiene uno sin parámetros.
<code>size</code>	Devuelve un valor de tipo int correspondiente al número de elementos de la lista. No tiene parámetros.
<code>add</code>	Añade un elemento en la posición <i>i</i> . Tiene dos parámetros: posición <i>i</i> y una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<code>addFirst</code>	Añade un elemento al principio. Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<code>addLast</code>	Añade un elemento al final. Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<code>remove</code>	Borra el elemento de la posición <i>i</i> . Tiene un parámetro de tipo int que se corresponde con la posición <i>i</i> del objeto a borrar. Devuelve una referencia de tipo Object al objeto borrado o lanza una excepción si la lista está vacía o el índice está fuera de límites.

Método	Significado
<i>removeFirst</i>	Borra el primer elemento. No tiene parámetros. Devuelve una referencia de tipo Object al objeto borrado o lanza una excepción si la lista está vacía.
<i>removeLast</i>	Borra el último elemento. No tiene parámetros. Devuelve una referencia de tipo Object al objeto borrado o lanza una excepción si la lista está vacía.
<i>get</i>	Devuelve una referencia de tipo Object correspondiente al elemento de la posición <i>i</i> , o bien lanza una excepción si la lista está vacía o el índice está fuera de límites. Tiene un parámetro de tipo int correspondiente a la posición <i>i</i> del objeto que se desea obtener.
<i>getFirst</i>	Devuelve una referencia de tipo Object correspondiente al primer elemento o lanza una excepción si la lista está vacía. No tiene parámetros.
<i>getLast</i>	Devuelve una referencia de tipo Object correspondiente al último elemento o lanza una excepción si la lista está vacía. No tiene parámetros.

A continuación se presenta otra versión de la aplicación *Test* utilizando ahora la clase **LinkedList** en lugar de *CListaLinealSE*.

```

import java.util.*;
// Crear una lista lineal simplemente enlazada
public class Test
{
    public static void mostrarLista(LinkedList lse)
    {
        int i = 0, tam = lse.size();
        CDatos obj;
        while (i < tam)
        {
            obj = (CDatos)lse.get(i);
            System.out.println(i + ". - " + obj.obtenerNombre() + " " +
                               obj.obtenerNota());
            i++;
        }
    }

    public static void main(String[] args)
    {
        // Crear una lista lineal vacía
        LinkedList lse = new LinkedList();
    }
}

```

```

// Leer datos y añadirlos a la lista
String nombre;
double nota;
int i = 0;
System.out.println("Introducir datos. Finalizar con Ctrl+Z.");
System.out.print("nombre: ");
while ((nombre = Leer.dato()) != null)
{
    System.out.print("nota:   ");
    nota = Leer.datoDouble();
    lse.addLast(new CDatos(nombre, nota));
    System.out.print("nombre: ");
}

// Añadir un objeto al principio
lse.addFirst(new CDatos("abcd", 10));
// Añadir un objeto en la posición 1
lse.add(1, new CDatos("defg", 9.5));

System.out.println("\n");
// Mostrar el primero
CDatos obj = (CDatos)lse.getFirst();
System.out.println("Primero: " + obj.obtenerNombre() + " " +
                   obj.obtenerNota());

// Mostrar el último
obj = (CDatos)lse.getLast();
System.out.println("Último: " + obj.obtenerNombre() + " " +
                   obj.obtenerNota());
// Mostrar todos
System.out.println("Lista:");
mostrarLista(lse);

// Borrar el elemento de índice 2
obj = (CDatos)lse.remove(2);

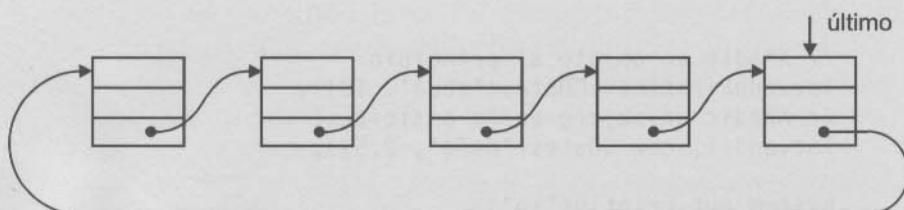
// Modificar el elemento de índice 1
obj = (CDatos)lse.get(1);
obj.asignarNota(9);

// Mostrar todos
System.out.println("Lista:");
mostrarLista(lse);
}
}

```

LISTAS CIRCULARES

Una *lista circular* es una lista lineal en la que el último elemento apunta al primero. Entonces es posible acceder a cualquier elemento de la lista desde cualquier punto dado. Las operaciones sobre una lista circular resultan más sencillas, ya que se evitan casos especiales. Por ejemplo, el método *añadir* de la clase *CListaLinealSE* expuesta anteriormente contempla dos casos: insertar al principio de la lista e insertar a continuación de un elemento. Con una lista circular, estos dos casos se reducen a uno. La siguiente figura muestra cómo se ve una lista circular simplemente enlazada.



Cuando recorremos una lista circular, diremos que hemos llegado al final de la misma cuando nos encontramos de nuevo en el punto de partida, suponiendo, desde luego, que el punto de partida se guarda de alguna manera en la lista; por ejemplo, con una referencia fija al mismo. Esta referencia puede ser al primer elemento de la lista; también puede ser al último elemento, en cuyo caso también es conocida la dirección del primer elemento. Otra posible solución sería poner un elemento especial identificable en cada lista circular como lugar de partida. Este elemento especial recibe el nombre de elemento de *cabecera* de la lista. Esto presenta, además, la ventaja de que la lista circular no estará nunca vacía.

Como ejemplo, vamos a construir una lista circular con una referencia fija al último elemento. Una lista circular con una referencia al último elemento es equivalente a una lista lineal recta con dos referencias, una al principio y otra al final.

Para construir una lista circular, primero tendremos que definir la clase de objetos que van a formar parte de la misma. Por ejemplo, cada elemento de la lista puede definirse como una estructura de datos con dos miembros: una referencia al elemento siguiente y otra al área de datos. El área de datos puede ser de un tipo predefinido o de un tipo definido por el usuario. Según esto, el tipo de cada elemento de la lista puede venir definido de la forma siguiente:

```
private class CElemento
{
    // Atributos
    private Object datos;
    private CElemento siguiente; // siguiente elemento
```

```

// Métodos
private CElemento() {} // constructor
private CElemento(Object d, CElemento s) // constructor
{
    datos = d;
    siguiente = s;
}

```

Vemos que por tratarse de una lista lineal simplemente enlazada, aunque sea circular, la estructura de sus elementos no varían con respecto a lo estudiado anteriormente.

Podemos automatizar el proceso de implementar una lista circular diseñando una clase *CListaCircularSE* (*Clase Lista Circular Simplemente Enlazada*) que proporcione los atributos y métodos necesarios para crear cada elemento de la lista, así como para permitir el acceso a los mismos. Esta clase nos permitirá posteriormente derivar otras clases que sean más específicas; por ejemplo, una clase para manipular *pilas* o una clase para manipular *colas*. Estas estructuras de datos las estudiaremos un poco más adelante.

Clase CListaCircularSE

La clase *CListaCircularSE* que vamos a implementar incluirá un atributo *último* que valdrá **null** cuando la lista esté vacía y cuando no, referenciará siempre a su último elemento; una clase interna, *CElemento*, que definirá la estructura de los elementos; y los métodos indicados en la tabla siguiente:

Método	Significado
<i>tamaño</i>	Devuelve el número de elementos de la lista. No tiene parámetros.
<i>añadirAlPrincipio</i>	Añade un elemento al principio (el primer elemento es el referenciado por <i>último.sigiente</i>). Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<i>añadirAlFinal</i>	Añade un elemento al final (el último elemento siempre estará referenciado por <i>último</i>). Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<i>borrar</i>	Borra el elemento primero (el primer elemento es el referenciado por <i>último.sigiente</i>). No tiene parámetros. Devuelve una referencia al objeto borrado o null si la lista está vacía.

Método	Significado
obtener	Devuelve el elemento de la posición <i>i</i> , o bien null si la lista está vacía o el índice está fuera de límites. Tiene un parámetro correspondiente a la posición <i>i</i> del objeto que se desea obtener.

A continuación se presenta el código correspondiente a la definición de la clase *CListaCircularSE*:

```
///////////////////////////////
// Lista lineal circular simplemente enlazada
//
public class CListaCircularSE
{
    // último: referencia el último elemento.
    // último.siguiente referencia al primer elemento de la lista.
    private CElemento último = null;

    // Elemento de una lista lineal circular simplemente enlazada
    private class CElemento
    {
        // Atributos
        private Object datos;          // referencia a los datos
        private CElemento siguiente;   // siguiente elemento
        // Métodos
        private CElemento() {} // constructor
        private CElemento(Object d, CELEMENTO s) // constructor
        {
            datos = d;
            siguiente = s;
        }
    }

    public CListaCircularSE() {} // constructor

    public int tamaño()
    {
        // Devuelve el número de elementos de la lista
        if (último == null) return 0;
        CELEMENTO q = último.siguiente; // primer elemento
        int n = 1;                      // número de elementos
        while (q != último)
        {
            n++;
            q = q.siguiente;
        }
        return n;
    }
}
```

```
public void añadirAlPrincipio(Object obj)
{
    // Añade un elemento al principio de la lista.
    // Crear el nuevo elemento
    CElemento q = new CElemento(obj, null);

    if( último != null ) // existe una lista
    {
        q.siguiente = último.siguiente;
        último.siguiente = q;
    }
    else // inserción del primer elemento
    {
        último = q;
        último.siguiente = q;
    }
}

public void añadirAlFinal(Object obj)
{
    // Añade un elemento al final de la lista.
    // Por lo tanto, último referenciará este nuevo elemento.
    // Crear el nuevo elemento.
    CElemento q = new CElemento(obj, null);

    if( último != null ) // existe una lista
    {
        q.siguiente = último.siguiente;
        último = último.siguiente = q;
    }
    else // inserción del primer elemento
    {
        último = q;
        último.siguiente = q;
    }
}

public Object borrar()
{
    // Devuelve una referencia a los datos del primer elemento de
    // la lista y borra este elemento.
    if( último == null )
    {
        System.err.println( "Lista vacía\n" );
        return null;
    }

    CElemento q = último.siguiente;
    Object obj = q.datos;
```

```

        if( q == último )
            último = null;
        else
            último.siguiente = q.siguiente;

        return obj;
        // El elemento referenciado por q es enviado a la basura, al
        // quedar desreferenciado cuando finaliza este método por ser
        // q una variable local.
    }

public Object obtener(int i)
{
    // Obtener el elemento de la posición i
    int númeroDeElementos = tamaño();
    if (i >= númeroDeElementos || i < 0)
        return null;

    CElemento q = último.siguiente; // primer elemento
    // Posicionarse en el elemento i
    for (int n = 0; n < i; n++)
        q = q.siguiente;

    // Retornar los datos
    return q.datos;
}
}
/////////////////////////////////////////////////////////////////

```

Una vez que hemos escrito la clase *CListaCircularSE* vamos a realizar una aplicación que utilizándola cree una lista circular y ponga a prueba las distintas operaciones que sobre ella pueden realizarse. Los elementos de esta lista serán objetos de la clase *CDatos* utilizada en ejemplos anteriores. El código de esta aplicación puede ser el siguiente:

```

/////////////////////////////////////////////////////////////////
// Crear una lista lineal circular simplemente enlazada
//
public class Test
{
    public static void mostrarLista(CListaCircularSE lcse)
    {
        // Mostrar todos los elementos de la lista
        int i = 0, tam = lcse.tamaño();
        CDatos obj;

        while (i < tam)
        {
            obj = (CDatos)lcse.obtener(i);

```

```

        System.out.println(i + " - " + obj.obtenerNombre() + " " +
                           obj.obtenerNota());
        i++;
    }
    if (tam == 0) System.out.println("lista vacía");
}

public static void main(String[] args)
{
    // Crear una lista circular vacía
    CListaCircularSE lcse = new CListaCircularSE();

    // Leer datos y añadirlos a la lista
    String nombre;
    double nota;
    int i = 0;
    System.out.println("Introducir datos. Finalizar con Ctrl+Z.");
    System.out.print("nombre: ");
    while ((nombre = Leer.dato()) != null)
    {
        System.out.print("nota: ");
        nota = Leer.datoDouble();
        lcse.añadirAlFinal(new CDatos(nombre, nota));
        System.out.print("nombre: ");
    }

    // Añadir un objeto al principio
    lcse.añadirAlPrincipio(new CDatos("abcd", 10));

    System.out.println("\n");
    // Mostrar la lista
    System.out.println("Lista:");
    mostrarLista(lcse);

    // Borrar el elemento primero
    CDatos obj = (CDatos)lcse.borrar();

    // Mostrar la lista
    System.out.println("Lista:");
    mostrarLista(lcse);
}
}

```

PILAS

Una *pila* es una lista lineal en la que todas las inserciones y supresiones se hacen en un extremo de la lista. Un ejemplo de esta estructura es una pila de platos. En ella, el añadir o quitar platos se hace siempre por la parte superior de la pila. Este

tipo de listas recibe también el nombre de listas *LIFO* (*last in first out* - último en entrar, primero en salir).

Las operaciones de meter y sacar en una pila son conocidas en los lenguajes ensambladores como *push* y *pop*, respectivamente. La operación de sacar un elemento de la pila suprime dicho elemento de la misma.

Para trabajar con pilas podemos diseñar una clase *CPila* (*Clase Pila*) derivada de la clase base *CListaCircularSE*, que soporte los siguientes métodos:

Método	Significado
<i>meterEnPila</i>	Mete un elemento en la cima de la pila (todas las inserciones se hacen por el principio de la lista). Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<i>sacarDePila</i>	Saca el primer elemento de la cima de la pila, eliminándolo de la misma (todas las supresiones se hacen por el principio de la lista). No tiene parámetros. Devuelve una referencia al objeto sacado de la pila o null si la pila está vacía.

Según lo expuesto, la definición de esta clase puede ser así:

```
///////////////////////////////
// Pila: lista en la que todas las inserciones y supresiones se
// hacen en un extremo de la misma.
//
public class CPila extends CListaCircularSE
{
    public CPila() {}

    public void meterEnPila(Object obj)
    {
        añadirAlPrincipio(obj);
    }

    public Object sacarDePila()
    {
        return borrar();
    }
}
/////////////////////////////
```

El constructor de la clase *CPila* llama primero al constructor de la clase base que crea una lista con cero elementos. El que la lista sea circular es transparente al usuario de la clase.

Para meter el elemento referenciado por el parámetro *obj* en la pila, el método *meterEnPila* invoca al método *añadirAlPrincipio* de la clase base *CListaCircularSE*; y para sacar el elemento de la cima de la pila y eliminarlo de la misma, el método *sacarDePila* invoca al método *borrar* de la clase base.

Observe que la derivación de la clase *CPila* de *CListaCircularSE* no oculta al usuario la interfaz pública de ésta, lo que permitiría utilizarla. Si quisieramos ocultarla podríamos haber declarado todos sus miembros de forma predeterminada, además de definir la clase dentro de un paquete concreto (por ejemplo, *estructuras*). Éste es un trabajo que queda fuera de los propósitos de este capítulo, pero con lo explicado en el capítulo de “Subclases e interfaces” no tendría ninguna dificultad en hacerlo. Lo que se ha pretendido aquí es crear un interfaz específica para el trabajo con pilas, y eso es lo que se ha hecho. Veremos una pequeña aplicación después de exponer el apartado relativo a *colas*.

COLAS

Una *cola* es una lista lineal en la que todas las inserciones se hacen por un extremo de la lista (por el final) y todas las supresiones se hacen por el otro extremo (por el principio). Por ejemplo, una fila en un banco. Este tipo de listas recibe también el nombre de listas *FIFO* (*first in first out* - primero en entrar, primero en salir). Este orden es la única forma de insertar y recuperar un elemento de la cola. Una cola no permite acceso aleatorio a un elemento específico. Tenga en cuenta que la operación de sacar elimina el elemento de la cola.

Para trabajar con colas podemos diseñar una clase *CCola* (*Clase Cola*) derivada de la clase base *CListaCircularSE*, que soporte los siguientes métodos:

Método	Significado
<i>meterEnCola</i>	Mete un elemento al final de la cola (todas las inserciones se hacen por el final de la lista). Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<i>sacarDeCola</i>	Saca el primer elemento de la cola, eliminándolo de la misma (todas las supresiones se hacen por el principio de la lista). No tiene parámetros. Devuelve una referencia al objeto sacado de la cola o null si la cola está vacía.

Según lo expuesto, la definición de esta clase puede ser así:

```
///////////////////////////////
// Cola: lista en la que todas las inserciones se hacen por un
// extremo de la lista (por el final) y todas las supresiones se
```

```

// hacen por el otro extremo (por el principio).
//
public class CCola extends CListaCircularSE
{
    public CCola() {}

    public void meterEnCola(Object obj)
    {
        añadirAlFinal(obj);
    }

    public Object sacarDeCola()
    {
        return borrar();
    }
}
/////////////////////////////////////////////////////////////////

```

El constructor de la clase *CCola* llama primero al constructor de la clase base que crea una lista con cero elementos. El que la lista sea circular es transparente al usuario de la clase.

Para meter el elemento referenciado por el parámetro *obj* en la cola, el método *meterEnCola* invoca al método *añadirAlFinal* de la clase base *CListaCircularSE*; y para sacar el elemento de la cola y eliminarlo de la misma, el método *sacarDeCola* invoca al método *borrar* de la clase base.

Observe que la derivación de la clase *CCola* de *CListaCircularSE* no oculta al usuario la interfaz pública de ésta, lo que permitiría utilizarla. Lo que se ha pretendido aquí es crear un interfaz específica para el trabajo con colas, y eso es lo que se ha hecho. Vea la explicación que hemos dado en este sentido al exponer las pilas.

EJEMPLO

El siguiente ejemplo muestra cómo utilizar la clase *CListaCircularSE* y sus derivadas *CPila* y *CCola*. Primeramente creamos una pila y una cola de objetos de la clase *CDatos* y a continuación creamos una lista circular. Para comprobar que las listas se han creado correctamente, mostramos a continuación los contenidos de las mismas. Además, para certificar que cuando se saca un elemento de una pila o de una cola éste es eliminado, intentamos mostrar por segunda vez el contenido de las mismas; el resultado es un mensaje de que están vacías.

En este ejemplo aparece también por primera vez el operador **instanceof** cuya sintaxis es la siguiente:

objeto instanceof clase

El resultado de una expresión como la anterior es **true** si el *objeto* es un ejemplar de la *clase* y **false** en caso contrario. Como ejemplo, observe el método *mostrarLista* de la aplicación siguiente. Este método tiene un parámetro que le permite recibir objetos de la clase *CListaCircularSE* o de sus derivadas: *CPila* y *CCola*. Utiliza el operador **instanceof** para saber con qué clase de objeto está trabajando y enviarle así los mensajes adecuados. Se puede observar que para sacar un elemento del objeto *lista*, si es de la clase *CPila* se le envía el mensaje *sacarDePila* y si es la clase *CCola*, *sacarDeCola*; en otro caso, no se hace nada.

```
//////////  
// Pilas y colas  
//  
public class Test  
{  
    public static void mostrarLista(CListaCircularSE lista)  
    {  
        // Mostrar todos los elementos de la lista  
        int i = 0, tam = lista.tamaño();  
        CDatos obj;  
        while (i < tam)  
        {  
            if (lista instanceof CPila)  
                obj = (CDatos)((CPila)lista).sacarDePila();  
            else if (lista instanceof CCola)  
                obj = (CDatos)((CCola)lista).sacarDeCola();  
            else  
            {  
                i++;  
                continue;  
            }  
            System.out.println(i + ". - " + obj.obtenerNombre() + " " +  
                               obj.obtenerNota());  
            i++;  
        }  
        if (tam == 0) System.out.println("lista vacía");  
    }  
  
    public static void main(String[] args)  
    {  
        // Crear una pila y una cola vacías  
        CPila pila = new CPila();  
        CCola cola = new CCola();  
  
        // Leer datos y añadirlos a ambas  
        String nombre;  
        double nota;  
        int i = 0;
```

```

System.out.println("Introducir datos. Finalizar con Ctrl+Z.");
System.out.print("nombre: ");
while ((nombre = Leer.dato()) != null)
{
    System.out.print("nota:   ");
    nota = Leer.datoDouble();
    pila.meterEnPila(new CDatos(nombre, nota));
    cola.meterEnCola(new CDatos(nombre, nota));
    System.out.print("nombre: ");
}
System.out.println("\n");

// Mostrar la pila
System.out.println("\nPila:");
mostrarLista(pila);
// Mostrar la pila por segunda vez
System.out.println("\nPila:");
mostrarLista(pila);

// Mostrar la cola
System.out.println("\nCola:");
mostrarLista(cola);
// Mostrar la cola por segunda vez
System.out.println("\nCola:");
mostrarLista(cola);

// Crear una lista circular
CListaCircularSE lcse = new CListaCircularSE();
lcse.append(new CDatos("lcse", 10));
// Mostrar la lista circular
System.out.println("\nlcse:");
mostrarLista(lcse);
}
}

```

Si ejecutamos esta aplicación e introducimos los siguientes datos,

```

Introducir datos. Finalizar con Ctrl+Z.
nombre: Alumno 1
nota: 7.5
nombre: Alumno 2
nota: 8.5
nombre: Alumno 3
nota: 9.5
nombre: [Ctrl+Z]

```

se mostrarán los siguientes resultados, los cuales indican que el operador **instanceof** ha funcionando correctamente discriminando la clase de objeto que se desea-

ba mostrar en cada instante, y que las operaciones de sacar en las pilas y colas eliminan el objeto sacado de las mismas.

Pila:

- 0.- Alumno 3 9.5
- 1.- Alumno 2 8.5
- 2.- Alumno 1 7.5

Pila:

lista vacía

Cola:

- 0.- Alumno 1 7.5
- 1.- Alumno 2 8.5
- 2.- Alumno 3 9.5

Cola:

lista vacía

lcse:

También se puede observar en estos resultados, que en las pilas el último objeto en entrar es el primero en salir y en las colas, el primero en entrar es el primero en salir.

LISTA DOBLEMENTE ENLAZADA

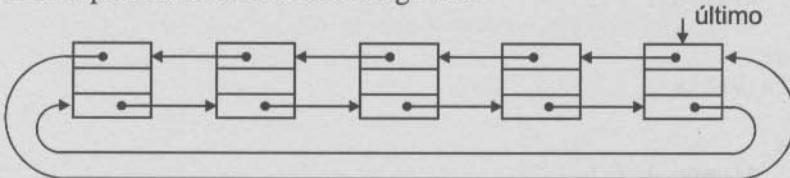
En una lista doblemente enlazada, a diferencia de una lista simplemente enlazada, cada elemento tiene información de dónde se encuentra el elemento posterior y el elemento anterior. Esto permite leer la lista en ambas direcciones. Este tipo de listas es útil cuando la inserción, borrado y movimiento de los elementos son operaciones frecuentes. Una aplicación típica es un procesador de textos, donde el acceso a cada línea individual se hace a través de una lista doblemente enlazada.

Las operaciones sobre una lista doblemente enlazada normalmente se realizan sin ninguna dificultad. Sin embargo, casi siempre es mucho más fácil la manipulación de las mismas cuando existe un doble enlace entre el último elemento y el primero, estructura que recibe el nombre de *lista circular doblemente enlazada*. Para moverse sobre una lista circular, es necesario almacenar de alguna manera un punto de referencia; por ejemplo, mediante una referencia al último elemento de la lista.

En el apartado siguiente se expone la forma de construir y manipular una lista circular doblemente enlazada.

Lista circular doblemente enlazada

Una *lista circular doblemente enlazada (lcde)* es una colección de objetos, cada uno de los cuales contiene datos o una referencia a los datos, una referencia al elemento siguiente en la colección y una referencia al elemento anterior. Gráficamente puede representarse de la forma siguiente:



Para construir una lista de este tipo, primero tendremos que definir la clase de objetos que van a formar parte de la misma. Por ejemplo, cada elemento de la lista puede definirse como una estructura de datos con tres miembros: una referencia al elemento siguiente, otra al elemento anterior y otra al área de datos. El área de datos puede ser de un tipo predefinido o de un tipo definido por el usuario. Según esto, el tipo de cada elemento de la lista puede venir definido de la forma siguiente:

```
private class CElemento
{
    // Atributos
    private Object datos;          // referencia a los datos
    private CElemento anterior;    // anterior elemento
    private CElemento siguiente;   // siguiente elemento

    // Métodos
    private CElemento() {} // constructor
}
```

Podemos automatizar el proceso de implementar una lista circular doblemente enlazada, diseñando una clase *CListaCircularDE* (*Clase Lista Circular Doblemente Enlazada*) que proporcione los atributos y métodos necesarios para crear cada elemento de la lista, así como para permitir el acceso a los mismos. La clase que diseñamos a continuación cubre estos objetivos.

Clase CListaCircularDE

La clase *CListaCircularDE* que vamos a implementar incluirá los atributos *último*, *actual*, *númeroDeElementos* y *posición*. El atributo *último* valdrá **null** cuando la lista esté vacía y cuando no, referenciará siempre a su último elemento; *actual* hace referencia al último elemento accedido; *númeroDeElementos* es el número de elementos que tiene la lista y *posición* indica el índice del elemento referencia-

do por *actual*. Asimismo, incluye una clase interna, *CElemento*, que definirá la estructura de los elementos, y los métodos indicados en la tabla siguiente:

Método	Significado
<i>CListaCircularDE</i>	Es el constructor. Inicia <i>último</i> y <i>actual</i> a null , <i>númeroDeElementos</i> a 0 y <i>posición</i> a -1 (la posición del primer elemento de la lista es la 0).
<i>tamaño</i>	Devuelve el número de elementos de la lista. No tiene parámetros.
<i>insertar</i>	Añade un elemento a continuación del referenciado por <i>actual</i> . El elemento añadido pasa a ser el elemento <i>actual</i> . Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. No devuelve ningún valor.
<i>borrar</i>	Borra el elemento referenciado por <i>actual</i> . No tiene parámetros. Devuelve una referencia al objeto borrado o null si la lista está vacía.
<i>irAlSiguiente</i>	Avanza la posición <i>actual</i> al siguiente elemento. Si esta posición coincide con <i>númeroDeElementos</i> -1, permanece en ella. No tiene parámetros y no devuelve ningún valor.
<i>irAlAnterior</i>	Retrasa la posición <i>actual</i> al elemento anterior. Si esta posición coincide con la 0, permanece en ella. No tiene parámetros y no devuelve ningún valor.
<i>irAlPrincipio</i>	Hace que la posición <i>actual</i> sea la 0. No tiene parámetros y no devuelve ningún valor.
<i>irAlFinal</i>	Hace que la posición <i>actual</i> sea la <i>númeroDeElementos</i> -1. No tiene parámetros y no devuelve ningún valor.
<i>irAl</i>	Avanza la posición <i>actual</i> al elemento de índice <i>i</i> (el primer elemento tiene índice 0). No tiene parámetros y devuelve true si la operación de mover se realiza con éxito o false si la lista está vacía o el índice está fuera de límites.
<i>obtener</i>	Devuelve el elemento referenciado por <i>actual</i> , o bien null si la lista está vacía. No tiene parámetros.
<i>obtener(i)</i>	Devuelve el elemento de la posición <i>i</i> , o bien null si la lista está vacía o el índice está fuera de límites. Tiene un parámetro correspondiente a la posición <i>i</i> del objeto que se desea obtener.
<i>modificar</i>	Establece nuevos datos para el elemento <i>actual</i> . Tiene un parámetro que es una referencia de tipo Object al nuevo objeto. No devuelve ningún valor.

A continuación se presenta el código correspondiente a la definición de la clase *CListaCircularDE*:

```
////////////////////////////////////////////////////////////////
// La clase lista circular doblemente enlazada permite manipular
// los elementos que componen una lista de este tipo.
//
public class CListaCircularDE
{
    private CElemento último;
        // referencia al último elemento de la lista
    private CElemento actual;
        // referencia al elemento actual en el que estamos
    private long númeroDeElementos;
        // número de elementos que tiene la lista
    private long posición;
        // posición del elemento actual

    // Elemento de una lista lineal circular doblemente enlazada
    private class CElemento
    {
        // Atributos
        private Object datos;          // referencia a los datos
        private CElemento anterior;    // anterior elemento
        private CElemento siguiente;   // siguiente elemento

        // Métodos
        private CElemento() {} // constructor
    }

    public CListaCircularDE() // constructor
    {
        actual = último = null;
        númeroDeElementos = 0L;
        posición = -1L; // la posición del primer elemento será la 0
    }

    public long tamaño()
    {
        // Permite saber el tamaño de la lista
        return númeroDeElementos;
    }

    public void insertar(Object obj)
    {
        // Añade un nuevo elemento a la lista a continuación
        // del elemento actual; el nuevo elemento pasa a ser el
        // actual.
        CElemento q;

        if (último == null) // lista vacía
        {
            último = new CElemento();
            últimos.
```

```

// Las dos líneas siguientes inician una lista circular.
último.anterior = último;
último.siguiente = último;
último.datos = obj;      // asignar datos.
actual = último;
posición = 0L;           // ya hay un elemento en la lista.
}
else // existe una lista
{
    q = new CElemento();

    // Insertar el nuevo elemento después del actual.
    actual.siguiente.anterior = q;
    q.siguiente = actual.siguiente;
    actual.siguiente = q;
    q.anterior = actual;
    q.datos = obj;

    // Actualizar parámetros.
    posición++;

    // Si el elemento actual es el último, el nuevo elemento
    // pasa a ser el actual y el último.
    if( actual == último )
        último = q;

    actual = q; // el nuevo elemento pasa a ser el actual.
} // fin else

númeroDeElementos++; // incrementar el número de elementos.
}

public Object borrar()
{
    // El método borrar devuelve los datos del elemento
    // referenciado por actual y lo elimina de la lista
    // (al quedar desreferenciado es enviado a la basura)
    CElemento q;
    Object obj;

    if( último == null ) return ( null ); // lista vacía.
    if( actual == último ) // se trata del último elemento.
    {
        if( númeroDeElementos == 1L ) // hay un solo elemento
        {
            obj = último.datos;
            último = actual = null;
            númeroDeElementos = 0L;
            posición = -1L;
        }
    }
}

```

```
else // hay más de un elemento
{
    actual = último.anterior;
    último.siguiente.anterior = actual;
    actual.siguiente = último.siguiente;
    obj = último.datos;
    último = actual;
    posición--;
    númeroDeElementos--;
} // fin del bloque else
} // fin del bloque if( actual == último )
else // el elemento a borrar no es el último
{
    q = actual.siguiente;
    actual.anterior.siguiente = q;
    q.anterior = actual.anterior;
    obj = actual.datos;
    actual = q;
    númeroDeElementos--;
}
return obj;
}

public void irAlSiguiente()
{
    // Avanza la posición actual al siguiente elemento.
    if (posición < númeroDeElementos - 1)
    {
        actual = actual.siguiente;
        posición++;
    }
}

public void irAlAnterior()
{
    // Retrasa la posición actual al elemento anterior.
    if (posición > 0L)
    {
        actual = actual.anterior;
        posición--;
    }
}

public void irAlPrincipio()
{
    // Hace que la posición actual sea el principio de la lista.
    actual = último.siguiente;
    posición = 0L;
}
```

```

public void irAlFinal()
{
    // El final de la lista es ahora la posición actual.
    actual = último;
    posición = númeroDeElementos - 1;
}

public boolean irAl(long i)
{
    // Posicionarse en el elemento i
    long númeroDeElementos = tamaño();
    if (i >= númeroDeElementos || i < 0) return false;

    irAlPrincipio();
    // Posicionarse en el elemento i
    for (long n = 0; n < i; n++)
        irAlSiguiente();
    return true;
}

public Object obtener()
{
    // El método obtener devuelve la referencia a los datos
    // asociados con el elemento actual.
    if (último == null) return null; // lista vacía

    return actual.datos;
}

public Object obtener(long i)
{
    // El método obtener devuelve la referencia a los datos
    // asociados con el elemento de índice i.
    if (!irAl(i)) return null;
    return obtener();
}

public void modificar(Object pNuevosDatos)
{
    // El método modificar establece nuevos datos para el
    // elemento actual.
    if (último == null) return; // lista vacía

    actual.datos = pNuevosDatos;
}
}

```

Cuando se declara un objeto de la clase *CListaCircularDE*, se ejecuta el constructor de la misma que realiza las siguientes operaciones:

- Crea una lista vacía (*último = actual = null*). En todo momento, el último elemento de la lista está apuntado por *último*, y *actual* apunta al elemento sobre el que se realizará la siguiente operación.
- Asigna un valor 0 al atributo *númeroDeElementos* y un valor -1 a *posición*; el valor de este atributo pasará a ser 0 cuando se añada el primer elemento.

El método *insertar* de la clase *CListaCircularDE* añade un elemento a la lista a continuación del elemento *actual*. Contempla dos casos: que la lista esté vacía, o que la lista ya exista. El elemento insertado pasa a ser el elemento *actual*, y si se añade al final, éste pasa a ser el *último* y el *actual*. Añadir un elemento implica realizar los enlaces con el anterior y siguiente elementos y actualizar los parámetros *actual*, *númeroDeElementos* y *último*, si procede.

El método *borrar* devuelve una referencia al objeto de datos asociado con el elemento *actual*, elemento que será enviado a la basura cuando finalice la ejecución del método por quedar desreferenciado. Contempla dos casos: que el elemento a borrar sea el *último* o que no lo sea. Si el elemento a borrar es el *último*, y sólo quedaba éste, los atributos de la lista deben iniciarse igual que lo hizo el constructor; si quedaban más de uno, el que es ahora el nuevo *último* pasa a ser también el elemento *actual*. Si el elemento a borrar no era el *último*, el elemento siguiente al eliminado pasa a ser el elemento *actual*. El método devuelve **null** si la lista está vacía.

Para el resto de los métodos es suficiente con la explicación dada al principio de este apartado, además de en el código.

Ejemplo

El siguiente ejemplo muestra cómo utilizar la clase *CListaCircularDE*. Primera-mente creamos un objeto *lcde*, correspondiente a una lista circular doblemente enlazada, en la que cada elemento almacenará un referencia a un objeto *CDatos*; y a continuación realizamos varias operaciones de inserción, movimiento y borrado, para finalmente visualizar los elementos de la lista y comprobar si los resultados son los esperados.

```
//////////  
// Lista circular doblemente enlazada  
//  
public class Test  
{  
    public static void mostrarLista(CListaCircularDE lista)  
    {  
        // Mostrar todos los elementos de la lista
```

```

long i = 0, tam = lista.tamaño();
CDatos obj;
while (i < tam)
{
    obj = (CDatos)lista.obtener(i);
    System.out.println(i + ". - " + obj.obtenerNombre() + " " +
                       obj.obtenerNota());
    i++;
}
if (tam == 0) System.out.println("lista vacía");
}

public static void main(String[] args)
{
    // Crear una lista vacía
    CListaCircularDE lcde = new CListaCircularDE();

    // Insertar elementos
    lcde.insertar(new CDatos("alumno1", 7.8));
    lcde.insertar(new CDatos("alumno2", 6.5));
    lcde.insertar(new CDatos("alumno3", 10));
    lcde.insertar(new CDatos("alumno4", 8.6));

    // Ir al elemento de la posición 2
    lcde.irAl(2);

    // Borrar el elemento actual (posición 2)
    lcde.borrar();

    // Ir al anterior
    lcde.irAl(1);
    lcde.insertar(new CDatos("nuevo alumno3", 9.5));

    // Ir al final
    lcde.irAlFinal();
    lcde.insertar(new CDatos("alumno5", 8.5));

    // Ir al anterior
    lcde.irAlAnterior();
    lcde.modificar(new CDatos("alumno4", 5.5));

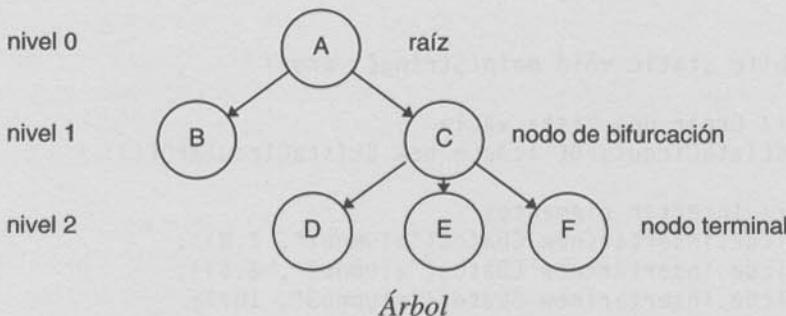
    // Mostrar la lista
    System.out.println("\nLista:");
    mostrarLista(lcde);
}
}

```

ÁRBOLES

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas.

En un árbol existe un nodo especial denominado *raíz*. Así mismo, un nodo del que sale alguna rama, recibe el nombre de *nodo de bifurcación* o *nodo rama* y un nodo que no tiene ramas recibe el nombre de *nodo terminal* o *nodo hoja*.



De un modo más formal, diremos que un árbol es un conjunto finito de uno o más nodos tales que:

- a) Existe un nodo especial llamado *raíz* del árbol, y
- b) los nodos restantes están agrupados en $n > 0$ conjuntos disjuntos A_1, \dots, A_n , cada uno de los cuales es a su vez un árbol que recibe el nombre de *subárbol de la raíz*.

Evidentemente, la definición dada es recursiva; es decir, hemos definido un árbol como un conjunto de árboles, que es la forma más apropiada de definirlo.

De la definición se desprende, que cada nodo de un árbol es la raíz de algún subárbol contenido en la totalidad del mismo.

El número de ramas de un nodo recibe el nombre de *grado* del nodo.

El nivel de un nodo respecto al nodo raíz se define diciendo que la raíz tiene nivel 0 y cualquier otro nodo tiene un nivel igual a la distancia de ese nodo al nodo raíz. El máximo de los niveles se denomina *profundidad* o *altura* del árbol.

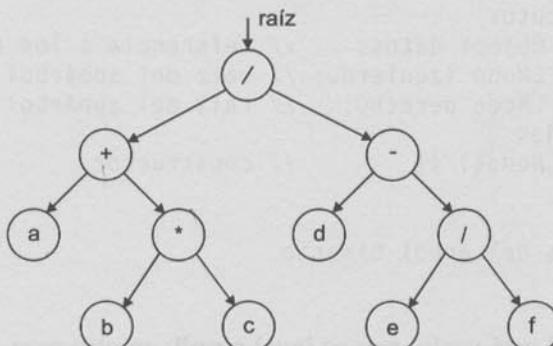
Es útil limitar los árboles en el sentido de que cada nodo sea a lo sumo de grado 2. De esta forma cabe distinguir entre subárbol izquierdo y subárbol derecho de un nodo. Los árboles así formados, se denominan *árboles binarios*.

Árboles binarios

Un árbol binario es un conjunto finito de nodos que consta de un *nodo raíz* que tiene dos subárboles binarios denominados *subárbol izquierdo* y *subárbol derecho*.

Las expresiones algebraicas, debido a que los operadores que intervienen son operadores binarios, nos dan un ejemplo de estructura en árbol binario. La figura siguiente nos muestra un árbol que corresponde a la expresión aritmética:

$$(a + b * c) / (d - e / f)$$



Expresión algebraica

El árbol binario es una estructura de datos muy útil cuando el tamaño de la estructura no se conoce, se necesita acceder a sus elementos ordenadamente, la velocidad de búsqueda es importante o el orden en el que se insertan los elementos es casi aleatorio.

En definitiva, un árbol binario es una colección de objetos (nodos del árbol) cada uno de los cuales contiene datos o una referencia a los datos, una referencia a su subárbol izquierdo y una referencia a su subárbol derecho. Según lo expuesto, la estructura de datos representativa de un nodo puede ser de la forma siguiente:

```

// Nodo de un árbol binario
private class CNodo
{
    // Atributos
    private Object datos;      // referencia a los datos
    private CNodo izquierdo;  // raíz del subárbol izquierdo
    private CNodo derecho;    // raíz del subárbol derecho
    // Métodos
    public CNodo() {}          // constructor
}
  
```

La definición dada de árbol binario, sugiere una forma natural de representar árboles binarios en un ordenador. Una variable *raíz* referenciará el árbol y cada nodo del árbol será un objeto de la clase *CNodo*. Esto es, la declaración genérica de un árbol binario puede ser así:

```
public class CArbolBinario
{
    // Atributos del árbol binario
    private CNodo raíz;           // raíz del árbol

    // Nodo de un árbol binario
    private class CNodo
    {
        // Atributos
        private Object datos;    // referencia a los datos
        private CNodo izquierdo; // raíz del subárbol izquierdo
        private CNodo derecho;   // raíz del subárbol derecho
        // Métodos
        public CNodo() {}        // constructor
    }

    // Métodos del árbol binario
}
```

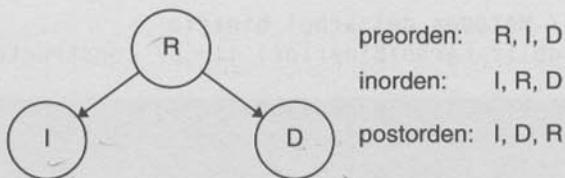
Si el árbol está vacío, *raíz* es igual a **null**; en otro caso, *raíz* es una referencia al nodo raíz del árbol y según se puede observar en el código anterior, este nodo tiene tres atributos: una referencia a los datos y dos referencias más, una a su subárbol izquierdo y otra a su subárbol derecho.

Formas de recorrer un árbol binario

Observe la figura “expresión algebraica” mostrada anteriormente ¿partiendo del nodo raíz, qué orden seguimos para poder evaluar la expresión que representa el árbol? Hay varios algoritmos para el manejo de estructuras en árbol y un proceso que generalmente se repite en estos algoritmos es el de recorrido de un árbol. Este proceso consiste en examinar sistemáticamente los nodos de un árbol, de forma que cada nodo sea visitado solamente una vez.

Básicamente se pueden utilizar tres formas para recorrer un árbol binario: *preorden*, *inorden* y *postorden*. Cuando se utiliza la forma *preorden*, primero se visita la raíz, después el subárbol izquierdo y por último el subárbol derecho; en cambio, si se utiliza la forma *inorden*, primero se visita el subárbol izquierdo, después la raíz y por último el subárbol derecho; y si se utiliza la forma *postorden*, primero se visita el subárbol izquierdo, después el subárbol derecho y por último la raíz.

R: raíz
 I: subárbol izquierdo
 D: subárbol derecho



preorden: R, I, D
 inorder: I, R, D
 postorden: I, D, R

Formas de recorrer un árbol

Evidentemente, las definiciones dadas son definiciones recursivas, ya que, recorrer un árbol utilizando cualquiera de ellas, implica recorrer sus subárboles empleando la misma definición.

Si se aplican estas definiciones al árbol binario de la figura “expresión algebraica” mostrada anteriormente, se obtiene la siguiente solución:

Preorden: / + a * b c - d / e f
 Inorden: a + b * c / d - e / f
 Postorden: a b c * + d e f / - /

El recorrido en preorden produce la notación *prefija*; el recorrido en inorder produce la notación *convencional*; y el recorrido en postorden produce la notación *postfija o inversa*.

Los nombres de preorden, inorder y postorden derivan del lugar en el que se visita la raíz con respecto a sus subárboles. Estas tres formas, se exponen a continuación como tres métodos recursivos de la clase *CArbolBinario*, con un único parámetro *r* que representa la raíz del árbol cuyos nodos se quieren visitar.

```

////////// Clase árbol binario.
// Clase árbol binario.
// 
public class CArbolBinario
{
    // Atributos del árbol binario
    private CNodo raíz;           // raíz del árbol

    // Nodo de un árbol binario
    private class CNodo
    {
        // Atributos
        private Object datos;      // referencia a los datos
        private CNodo izquierdo;   // raíz del subárbol izquierdo
        private CNodo derecho;     // raíz del subárbol derecho
        // Métodos
        public CNodo() {}          // constructor
    }
}
  
```

```

// Métodos del árbol binario
public CArbolBinario() {} // constructor

public void preorden(CNodo r)
{
    if ( r != null )
    {
        // Escribir aquí las operaciones a realizar
        // con el nodo referenciado r
        preorden( r.izquierdo); // se visita el subárbol izquierdo
        preorden( r.derecho); // se visita el subárbol derecho
    }
}

public void inorder(CNodo r)
{
    if ( r != null )
    {
        inorder( r.izquierdo); // se visita el subárbol izquierdo
        // Escribir aquí las operaciones a realizar
        // con el nodo referenciado r
        inorder( r.derecho); // se visita el subárbol derecho
    }
}

public void postorden(CNodo r)
{
    if ( r != null )
    {
        postorden( r.izquierdo); // se visita el subárbol izquierdo
        postorden( r.derecho); // se visita el subárbol derecho
        // Escribir aquí las operaciones a realizar
        // con el nodo referenciado r
    }
}
/////////////////////////////////////////////////////////////////

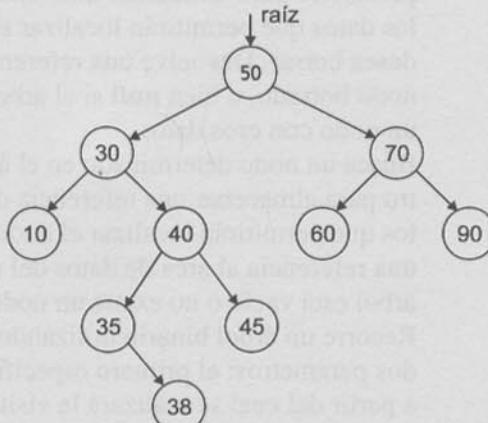
```

ÁRBOLES BINARIOS DE BÚSQUEDA

Un *árbol binario de búsqueda* es un árbol ordenado; esto es, las ramas de cada nodo están ordenadas de acuerdo con las siguientes reglas: para todo nodo a_i , todas las claves del subárbol izquierdo de a_i son menores que la clave de a_i , y todas las claves del subárbol derecho de a_i son mayores que la clave de a_i .

Con un árbol de estas características, encontrar si un nodo de una clave determinada existe o no es una operación muy sencilla. Por ejemplo, observando la figura siguiente, localizar la clave 35 es aplicar la definición de árbol de búsque-

da; esto es, si la clave buscada es menor que la clave del nodo en el que estamos, pasamos al subárbol izquierdo de este nodo para continuar la búsqueda, y si es mayor, pasamos al subárbol derecho. Este proceso continúa hasta encontrar la clave o hasta llegar a un subárbol vacío, árbol cuya raíz tiene un valor **null**.



Árbol binario de búsqueda

En Java podemos automatizar el proceso de implementar un árbol binario de búsqueda diseñando una clase *CArbolBinB* (Clase *Arbol Binario de Búsqueda*) que proporcione los atributos y métodos necesarios para crear cada nodo del árbol, así como para permitir el acceso a los mismos. La clase que diseñamos a continuación cubre estos objetivos.

Clase CArbolBinB

La clase *CArbolBinB* que vamos a implementar incluirá un atributo protegido *raíz* para referenciar la raíz del árbol y cuatro constantes públicas relacionadas con los posibles errores que se pueden dar relativos a un nodo: *CORRECTO*, *NO_DATOS*, *YA_EXISTE* y *NO_EXISTE*. El atributo *raíz* valdrá *null* cuando el árbol esté vacío. Asimismo, incluye una clase interna, *CNodo*, que definirá la estructura de los nodos, y los métodos indicados en la tabla siguiente:

Método	Significado
<i>CArbolBinB</i>	Es el constructor. Crea un árbol vacío (<i>raíz</i> a <i>null</i>).
<i>insertar</i>	Inserta un nodo en el árbol binario basándose en la definición de árbol binario de búsqueda. Tiene un parámetro que es una referencia de tipo Object al objeto a añadir. Devuelve un 0, definido por la constante <i>CORRECTO</i> , si la operación de inserción se realiza satisfactoriamente, o un valor

	distinto de 0, definido por alguna de las constantes siguientes: NO_DATOS si la raíz es igual a null , o YA_EXISTE si el nodo con esos datos ya existe.
<i>borrar</i>	Borra un nodo de un árbol binario de búsqueda. Tiene un parámetro para almacenar una referencia de tipo Object a los datos que permitirán localizar en el árbol el nodo que se desea borrar. Devuelve una referencia al área de datos del nodo borrado, o bien null si el árbol está vacío o no existe un nodo con esos datos.
<i>buscar</i>	Busca un nodo determinado en el árbol. Tiene un parámetro para almacenar una referencia de tipo Object a los datos que permitirán localizar el nodo en el árbol. Devuelve una referencia al área de datos del nodo, o bien null si el árbol está vacío o no existe un nodo con esos datos.
<i>inorden</i>	Recorre un árbol binario utilizando la forma <i>inorden</i> . Tiene dos parámetros: el primero especifica la referencia al nodo a partir del cual se realizará la visita; el valor del primer parámetro sólo será tenido en cuenta si el segundo es false , porque si es true se asume que el primer parámetro es la raíz del árbol. No devuelve ningún valor.
<i>comparar</i>	Método que debe ser redefinido por el usuario en una subclase para especificar el tipo de comparación que se desea realizar con dos nodos del árbol. Debe de devolver un entero indicando el resultado de la comparación (-1, 0 ó 1 si <i>nodo1</i> < <i>nodo2</i> , <i>nodo1</i> == <i>nodo2</i> , o <i>nodo1</i> > <i>nodo2</i> , respectivamente). Este método es invocado por los métodos <i>insertar</i> , <i>borrar</i> y <i>buscar</i> .
<i>proceso</i>	Método que debe ser redefinido por el usuario en una subclase para especificar las operaciones que se desean realizar con el nodo visitado. Es invocado por el método <i>inorden</i> .
<i>visitarInorden</i>	Método sin parámetros que debe ser redefinido por el usuario en una subclase para invocar al método <i>inorden</i> .

A continuación se presenta el código correspondiente a la definición de la clase *CArbolBinB*:

```
///////////////////////////////
// Clase abstracta: árbol binario de búsqueda. Para utilizar los
// métodos proporcionados por esta clase, tendremos que crear
// una subclase de ella y redefinir los métodos:
// comparar, procesar y visitarInorden.
//
public abstract class CArbolBinB
{
```

```

// Atributos del árbol binario
protected CNodo raíz = null; // raíz del árbol

// Nodo de un árbol binario
private class CNodo
{
    // Atributos
    private Object datos;      // referencia a los datos
    private CNodo izquierdo;   // raíz del subárbol izquierdo
    private CNodo derecho;     // raíz del subárbol derecho

    // Métodos
    public CNodo() {}          // constructor
}

// Posibles errores que se pueden dar relativos a un nodo
public static final int CORRECTO = 000;
public static final int NO_DATOS = 100;
public static final int YA_EXISTE = 101;
public static final int NO_EXISTE = 102;

// Métodos del árbol binario
public CArbolBinB() {}        // constructor

// El método siguiente debe ser redefinido en una subclase para
// que permita comparar dos nodos del árbol por el atributo
// que necesitemos en cada momento.
public abstract int comparar(Object obj1, Object obj2);

// El método siguiente debe ser redefinido en una subclase para
// que permita especificar las operaciones que se deseen
// realizar con el nodo visitado.
public abstract void procesar(Object obj);

// El método siguiente debe ser redefinido en una subclase para
// que invoque a "inorden" con los argumentos deseados.
public abstract void visitarInorden();

public Object buscar(Object obj)
{
}

public int insertar(Object obj)
{
}

public Object borrar(Object obj)
{
}

```

```

public void inorder( CNodo r , boolean nodoRaíz)
{
    // El método recursivo inorder visita los nodos del árbol
    // utilizando la forma inorder; esto es, primero se visita
    // el subárbol izquierdo, después se visita la raíz, y por
    // último, el subárbol derecho.
    // Si el segundo parámetro es true, la visita comienza
    // en la raíz independientemente del primer parámetro.

    CNodo actual = null;

    if ( nodoRaíz )
        actual = raíz; // partir de la raíz
    else
        actual = r; // partir de un nodo cualquiera
    if ( actual != null )
    {
        inorder( actual.izquierdo, false ); // visitar subárbol izq.
        // Procesar los datos del nodo visitado
        procesar( actual.datos );
        inorder( actual.derecho, false ); // visitar subárbol dcho.
    }
}
/////////////////////////////////////////////////////////////////

```

Buscar un nodo en el árbol

El método *buscar* cuyo código se muestra a continuación permite acceder a los datos de un nodo del árbol. Su sintaxis es la siguiente:

```
public Object buscar(Object obj)
```

El parámetro *obj* se refiere al objeto de datos, que suponemos apuntado por un nodo del árbol, al que deseamos acceder. Este método devuelve un valor **null** si el objeto referenciado por *obj* no se localiza en el árbol, o bien una referencia al objeto de datos del nodo localizado.

Por definición de árbol de búsqueda, sabemos que sus nodos tienen que estar ordenados utilizando como clave alguno de los atributos de *obj*. Según esto, el método *buscar* se escribe aplicando estrictamente esa definición; esto es, si la clave buscada es menor que la clave del nodo en el que estamos, continuamos la búsqueda en su subárbol izquierdo y si es mayor, entonces continuamos la búsqueda en su subárbol derecho. Este proceso continúa hasta encontrar la clave, o bien hasta llegar a un subárbol vacío (subárbol cuya raíz tiene un valor **null**), en

cuyo caso se inserta en ese lugar un nuevo nodo que almacenará la referencia *obj* al objeto de datos.

Para saber si una clave es igual, menor o mayor que otra invocaremos al método *comparar* pasando como argumentos los objetos de datos que contienen los atributos que se desean comparar. Como tales atributos, dependiendo de la aplicación, pueden ser bien de algún tipo numérico, o bien de tipo alfanumérico o alfabético, la implementación de este método hay que posponerla al diseño de la aplicación que utilice esta clase, razón por la que *comparar* ha sido definido como un método abstracto. Para ello, como veremos un poco más adelante, derivaremos una nueva clase de ésta, y redefiniremos este método.

```
public Object buscar(Object obj)
{
    // El método buscar permite acceder a un determinado nodo.
    CNodo actual = raiz;
    int nComp = 0;

    // Buscar un nodo que tenga asociados los datos dados por obj
    while (actual != null)
    {
        if ((nComp = comparar(obj, actual.datos)) == 0)
            return (actual.datos); // CORRECTO (nodo encontrado)
        else if (nComp < 0)      // buscar en el subárbol izquierdo
            actual = actual.izquierdo;
        else                      // buscar en el subárbol derecho
            actual = actual.derecho;
    }
    return null; // NO_EXISTE
}
```

Insertar un nodo en el árbol

El método *insertar* cuyo código se muestra a continuación permite añadir un nodo que aún no existe en el árbol. Su sintaxis es la siguiente:

```
public int insertar(Object obj)
```

El parámetro *obj* se refiere al objeto de datos que será apuntado por el nodo que se añadirá al árbol. Devuelve un entero *NO_DATOS* si *obj* es **null**, *CORRECTO* si la operación de insertar se ejecuta con éxito, y *YA_EXISTE* si ya hay un nodo con los datos referenciados por *obj*.

El proceso realizado por este método lo primero que hace es verificar si ya hay un nodo con estos datos en el árbol (para realizar esta operación se sigue el

mismo proceso descrito en el método *buscar*), en cuyo caso, como ya se indicó en el párrafo anterior, lo notificará. Si ese nodo no se encuentra, el proceso de búsqueda nos habrá conducido hasta un nodo terminal, posición donde lógicamente debe añadirse el nuevo nodo que almacenará la referencia *obj* a los datos.

```

public int insertar(Object obj)
{
    // El método insertar permite añadir un nodo que aún no está
    // en el árbol.
    CNodo último = null, actual = raíz;
    int nComp = 0;
    if ( obj == null ) return NO_DATOS;

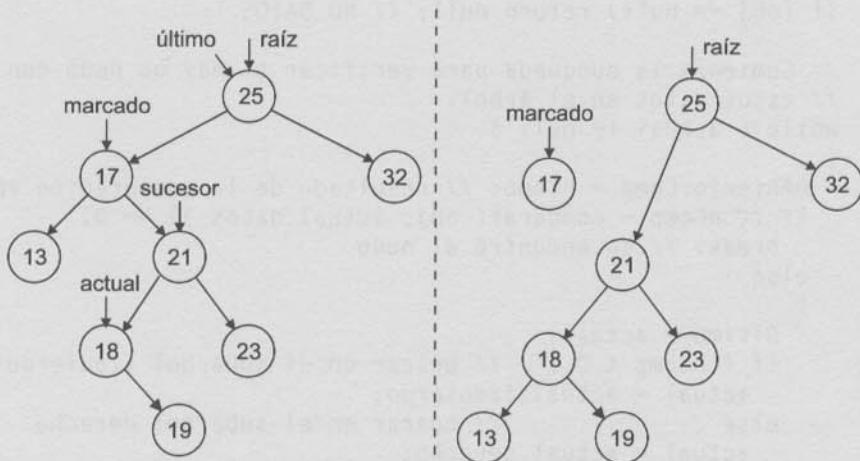
    // Comienza la búsqueda para verificar si ya hay un nodo con
    // estos datos en el árbol
    while (actual != null)
    {
        if ((nComp = comparar( obj, actual.datos )) == 0)
            break; // se encontró el nodo
        else
        {
            último = actual;
            if ( nComp < 0 ) // buscar en el subárbol izquierdo
                actual = actual.izquierdo;
            else // buscar en el subárbol derecho
                actual = actual.derecho;
        }
    }

    if ( actual == null ) // no se encontró el nodo, añadirlo
    {
        CNodo nuevoNodo = new CNodo();
        nuevoNodo.datos = obj;
        nuevoNodo.izquierdo = nuevoNodo.derecho = null;
        // El nodo a añadir pasará a ser la raíz del árbol total si
        // éste está vacío, del subárbol izquierdo de "último" si la
        // comparación fue menor, o del subárbol derecho de "último" si
        // la comparación fue mayor.
        if ( último == null ) // árbol vacío
            raíz = nuevoNodo;
        else if ( nComp < 0 )
            último.izquierdo = nuevoNodo;
        else
            último.derecho = nuevoNodo;
        return CORRECTO;
    } // fin del bloque if ( actual == null )
    else // el nodo ya existe en el árbol
        return YA_EXISTE;
}

```

Borrar un nodo del árbol

A continuación se estudia el problema de borrar un determinado nodo de un árbol que tiene las claves ordenadas. Este proceso es una tarea fácil si el nodo a borrar es un nodo terminal o si tiene un único descendiente. La dificultad se presenta cuando deseamos borrar un nodo que tiene dos descendientes (en la figura, 17), ya que con una sola referencia no se puede apuntar en dos direcciones. En este caso, el lugar en el árbol del nodo a borrar será reemplazado por su sucesor presentándose dos casos: si tomamos como sucesor la raíz de su subárbol izquierdo (13), su subárbol derecho (21) lo será ahora del nodo más a la derecha (13) en el subárbol izquierdo (13), y si tomamos como sucesor la raíz de su subárbol derecho (21), su subárbol izquierdo (13) lo será ahora del nodo más a la izquierda (18) en el subárbol derecho (21).



Borrar el nodo con clave 17

En el ejemplo que muestra la figura anterior, se desciende por el árbol hasta encontrar el nodo a borrar (17). La variable *actual* representa la raíz del subárbol en el que continúa la búsqueda; inicialmente su valor es *raíz*. La variable *marcado* apunta al nodo a borrar una vez localizado, el cual es sustituido por su *sucesor* a la derecha (21) pasando su subárbol izquierdo (13) a serlo ahora del nodo más a la izquierda (18) en el subárbol derecho (21). Cuando finalice la ejecución del método, el nodo que queda referenciado por *marcado* será enviado a la basura, ya que *marcado* es una variable local y desaparecerá. El proceso detallado, se presenta a continuación y contempla los casos mencionados:

1. El nodo a borrar es un nodo terminal; no tiene descendientes.
2. El nodo a borrar no tiene subárbol izquierdo.
3. El nodo a borrar no tiene subárbol derecho.

4. El nodo a borrar tiene subárbol izquierdo y derecho.

Resumiendo, el método *borrar* primero localiza el nodo a borrar y lo marca (queda referenciado por *marcado*). Despues analiza si éste tiene descendientes y cuántos; en función de esto obtiene su sucesor (queda referenciado por *sucesor*). Finalmente, rehace los enlaces dejando fuera el nodo marcado para borrar. Dicho nodo será enviado a la basura en el instante en que finaliza el método *borrar*.

```

public Object borrar(Object obj)
{
    // El método borrar permite eliminar un nodo del árbol.
    CNodo ultimo = null, actual = raíz;
    CNodo marcado = null, sucesor = null;
    int nAnteriorComp = 0, nComp = 0;

    if (obj == null) return null; // NO_DATOS

    // Comienza la búsqueda para verificar si hay un nodo con
    // estos datos en el árbol.
    while (actual != null)
    {
        nAnteriorComp = nComp; // resultado de la comparación anterior
        if ((nComp = comparar( obj, actual.datos )) == 0)
            break; // se encontró el nodo
        else
        {
            último = actual;
            if (nComp < 0) // buscar en el subárbol izquierdo
                actual = actual.izquierdo;
            else // buscar en el subárbol derecho
                actual = actual.derecho;
        }
    } // fin del bloque while (actual != null)

    if (actual != null) // se encontró el nodo
    {
        marcado = actual;
        if ((actual.izquierdo == null && actual.derecho == null))
            // se trata de un nodo terminal (no tiene descendientes)
            sucesor = null;
        else if (actual.izquierdo == null) // nodo sin subárbol izq.
            sucesor = actual.derecho;
        else if (actual.derecho == null) // nodo sin subárbol derecho
            sucesor = actual.izquierdo;
        else // nodo con subárbol izquierdo y derecho
        {
            // Referencia del subárbol derecho del nodo a borrar
            sucesor = actual.derecho;
        }
    }
}

```

```

// Descender al nodo más a la izquierda en el subárbol
// derecho de este nodo (el de valor más pequeño) y hacer
// que el subárbol izquierdo del nodo a borrar sea ahora
// el subárbol izquierdo de este nodo.
while ( actual.izquierdo != null )
    actual = actual.izquierdo;
actual.izquierdo = marcado.izquierdo;
}

// Eliminar el nodo y rehacer los enlaces
if ( último != null )
{
    if ( nAnteriorComp < 0 )
        último.izquierdo = sucesor;
    else
        último.derecho = sucesor;
}
else
    raíz = sucesor;

return marcado.datos;; // CORRECTO
// "marcado" será enviado a la basura
}
else // el nodo buscado no está en el árbol
    return null; // NO_EXISTE
}

```

Utilización de la clase CArbolBinB

La clase *CArbolBinB* es una clase abstracta; por lo tanto, para hacer uso del soporte que proporciona para la construcción y manipulación de árboles binarios de búsqueda, tendremos que derivar una clase de ella y redefinir los métodos abstractos heredados: *comparar*, *procesar* y *visitarInorden*. La redefinición de estos métodos está condicionada a la clase de objetos que formarán parte del árbol.

Como ejemplo, vamos a construir un árbol binario de búsqueda en el que cada nodo haga referencia a un objeto de la clase *CDatos* ya utilizada anteriormente en este mismo capítulo. Esto sugiere pensar en la clave de ordenación que se utilizará para construir el árbol. En nuestro ejemplo vamos a ordenar los nodos del árbol por el atributo *nombre* de *CDatos*. Se trata entonces de una ordenación alfabética; por tanto, el método *comparar* debe ser redefinido para que devuelva -1, 0 ó 1 según sea el *nombre* de un objeto *CDatos*, menor, igual o mayor, respectivamente, que el *nombre* del otro objeto con el que se compara.

Pensemos ahora en el proceso que deseamos realizar con cada nodo accedido. En el ejemplo, simplemente nos limitaremos a mostrar los datos *nombre* y *nota*.

Según esto, el método *procesar* obtendrá los datos *nombre* y *nota* del objeto *CDatos* pasado como argumento y los mostrará.

Finalmente, escribiremos el método *visitarInorden* para que permita recorrer, en nuestro caso, el árbol en su totalidad.

```
///////////
// Clase derivada de la clase abstracta CArbolBinB. Redefine los
// métodos: comparar, procesar y visitarInorden.
//
public class CArbolBinarioDeBusqueda extends CArbolBinB
{
    // Permite comparar dos nodos del árbol por el atributo nombre.
    public int comparar(Object obj1, Object obj2)
    {
        String str1 = new String(((CDatos)obj1).obtenerNombre());
        String str2 = new String(((CDatos)obj2).obtenerNombre());
        return str1.compareTo(str2);
    }

    // Permite mostrar los datos del nodo visitado.
    public void procesar(Object obj)
    {
        String nombre = new String(((CDatos)obj).obtenerNombre());
        double nota = ((CDatos)obj).obtenerNota();
        System.out.println(nombre + " " + nota);
    }

    // Visitar los nodos del árbol.
    public void visitarInorden()
    {
        // Si el segundo argumento es true, la visita comienza
        // en la raíz independientemente del primer argumento.
        inorder(null, true);
    }
}
///////
```

Ahora puede comprobar de una forma clara que los métodos *comparar*, *procesar* y *visitarInorden* dependen del tipo de objetos que almacenemos en el árbol que construyamos. Por esta razón no pudieron ser implementados en la clase *CArbolBinB*, sino que hay que implementarlos para cada caso particular.

Observe que como los parámetros de los métodos *comparar* y *procesar* son genéricos, referencias de tipo **Object**, deben convertirse explícitamente en referencias a la clase de objetos que realmente representan; en nuestro caso a referencias a objetos de la clase *CDatos*, de lo contrario no tendremos acceso a los métodos explícitos de esta clase.

Cuando se declare un objeto de la clase *CArbolBinarioDeBusqueda*, el constructor de esta clase invoca al constructor *CArbolBinB* de su clase base, que creará un árbol vacío (*raíz = null*). El atributo *raíz* apunta siempre a la raíz del árbol.

Finalmente, escribiremos una aplicación *Test* que, utilizando la clase *CArbolBinarioDeBusqueda*, cree un objeto *arbolbb* correspondiente a un árbol binario de búsqueda en el que cada nodo haga referencia a un objeto *CDatos* que encapsule el nombre de un alumno y la nota de una determinada asignatura que está cursando. Con el fin de probar que todos los métodos proporcionados por la clase funcionan adecuadamente (piense en los métodos heredados y en los redefinidos), la aplicación realizará las operaciones siguientes:

1. Creará un objeto *arbolbb* de la clase *CArbolBinarioDeBusqueda*.
2. Solicitará parejas de datos *nombre* y *nota*, a partir de las cuales construirá los objetos *CDatos* que añadiremos como nodos en el *arbolbb*.
3. Durante la construcción del árbol, permitirá modificar la nota de un alumno ya existente, o bien eliminarlo. Para discriminar una operación de otra tomaremos como referencia la nueva nota: si es positiva, entenderemos que deseamos modificar la nota del alumno especificado y si es negativa, que hay que eliminarlo.
4. Finalmente, mostrará los datos almacenados en el árbol para comprobar que todo ha sucedido como esperábamos.

```
//////////  
// Crear un árbol binario de búsqueda  
//  
public class Test  
{  
    public static void main(String[] args)  
    {  
        CArbolBinarioDeBusqueda arbolbb = new CArbolBinarioDeBusqueda();  
  
        // Leer datos y añadirlos al árbol  
        String nombre;  
        double nota;  
        int i = 0, cod;  
  
        System.out.println("Introducir datos. Finalizar con Ctrl+Z.");  
  
        System.out.print("nombre: ");  
        while ((nombre = Leer.dato()) != null)  
        {  
            System.out.print("nota: ");  
            nota = Leer.datoDouble();  
            cod = arbolbb.insertar(new CDatos(nombre, nota));  
        }  
    }  
}
```

```

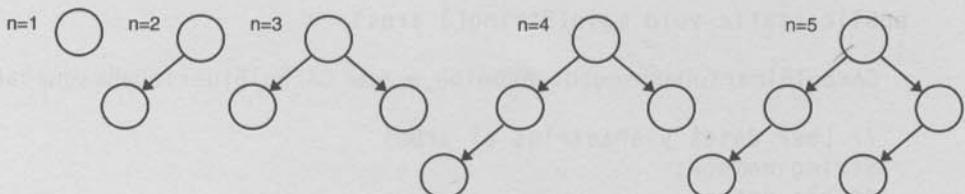
        if (cod == CArbolBinarioDeBusqueda.YAEXISTE)
        {
            // Si ya existe, distinguimos dos casos:
            // 1. nota nueva >= 0; cambiamos la nota
            // 2. nota nueva < 0; borramos el nodo
            CDatos datos = (CDatos)arbolbb.buscar(new CDatos(nombre, nota));
            if (nota >= 0)
                datos.asignarNota(nota);
            else
            {
                if (arbolbb.borrar(new CDatos(nombre, nota)) == null)
                    System.out.println("no borrado porque no existe");
                else
                    System.out.println("nodo borrado");
            }
        }
        System.out.print("nombre: ");
    }
    System.out.println("\n");

    // Mostrar los nodos del árbol
    System.out.println("\nArbol:");
    arbolbb.visitarInorden();
}
}

```

ÁRBOLES BINARIOS PERFECTAMENTE EQUILIBRADOS

Un árbol binario está perfectamente equilibrado si, para todo nodo, el número de nodos en el subárbol izquierdo y el número de nodos en el subárbol derecho, difieren como mucho en una unidad.



Árboles perfectamente equilibrados

Como ejemplo, considere el problema de construir un árbol perfectamente equilibrado siendo los valores de los nodos, n referencias a objetos de la clase *CDatos* implementada anteriormente en este mismo capítulo. Recuerde que cada objeto de esta clase encapsula el nombre de un alumno y la nota de una determinada asignatura que está cursando.

Esto puede realizarse fácilmente distribuyendo los nodos, según se lean, equitativamente a la izquierda y a la derecha de cada nodo. El proceso recursivo que se indica a continuación, es la mejor forma de realizar esta distribución. Para un número dado n de nodos y siendo ni (nodos a la izquierda) y nd (nodos a la derecha) dos enteros, el proceso es el siguiente:

1. Utilizar un nodo para la raíz.
2. Generar el subárbol izquierdo con $ni = n/2$ nodos utilizando la misma regla.
3. Generar el subárbol derecho con $nd = n-ni-1$ nodos utilizando la misma regla.

Cada nodo del árbol consta de los siguientes miembros: *datos*, referencia al subárbol *izquierdo* y referencia al subárbol *derecho*.

```
private class CNodo
{
    // Atributos
    private Object datos;          // referencia a los datos
    private CNodo izquierdo;       // raíz del subárbol izquierdo
    private CNodo derecho;         // raíz del subárbol derecho

    // Métodos
    public CNodo() {}             // constructor
}
```

En Java podemos automatizar el proceso de implementar un árbol binario perfectamente equilibrado diseñando una clase *CArbolBinE* (Clase *Arbol Binario Equilibrado*) que proporcione los atributos y métodos necesarios para crear cada nodo del árbol, así como para permitir el acceso a los mismos.

Clase CArbolBinE

La clase *CArbolBinE* que vamos a implementar incluirá un atributo protegido *raíz* para referenciar la raíz del árbol. El atributo *raíz* valdrá **null** cuando el árbol esté vacío. Asimismo, incluye la clase interna *CNodo* que define la estructura de los nodos, y los métodos indicados en la tabla siguiente:

Método	Significado
<i>CArbolBinE</i>	Es el constructor; como es igual que el constructor por omisión, podría omitirse. Crea un árbol vacío (<i>raíz</i> a null).
<i>construirArbol</i>	Es un método privado que permite construir un árbol binario perfectamente equilibrado. Tiene un parámetro de tipo int que se corresponde con el número de nodos que va a tener el árbol. Devuelve una referencia a la raíz del árbol.

construirArbolEquilibrado

Invoca al método *construirArbol* pasando como argumento el número de nodos y almacena el valor devuelto por él, en el atributo *raíz* de la clase. No devuelve nada. Su misión es evitar que el usuario de la clase tenga que utilizar directamente el atributo *raíz*.

buscar

Busca un nodo determinado en el árbol. Tiene un parámetro para almacenar una referencia de tipo **Object** a los datos que permitirán localizar el nodo en el árbol. Devuelve una referencia al área de datos del nodo, o bien **null** si el árbol está vacío o no existe un nodo con esos datos. Opcionalmente se puede especificar un segundo parámetro correspondiente a la posición del nodo según el orden de acceso seguido por el método *inorden* (consideraremos que la primera posición es la 0).

inorden

Recorre un árbol binario utilizando la forma *inorden*. Tiene dos parámetros: el primero especifica la referencia al nodo a partir del cual se realizará la visita; el valor del primer parámetro sólo será tenido en cuenta si el segundo es **false**, porque si es **true** se asume que el primer parámetro es la raíz del árbol. No devuelve ningún valor.

leerDatos

Método que debe ser redefinido por el usuario en una subclase para que permita leer los datos que serán referenciados por un nodo del árbol. Devuelve el objeto de datos. Es invocado por el método *construirArbol*.

comparar

Método que debe ser redefinido por el usuario en una subclase para especificar el tipo de comparación que se desea realizar con dos nodos del árbol. Debe de devolver un entero indicando el resultado de la comparación (-1, 0 ó 1 si *nodo1 < nodo2*, *nodo1 == nodo2*, o *nodo1 > nodo2*, respectivamente). Este método es invocado por los métodos *insertar*, *borrar* y *buscar*.

proceso

Método que debe ser redefinido por el usuario en una subclase para especificar las operaciones que se desean realizar con el nodo visitado. Es invocado por el método *inorden*.

visitarInorden

Método sin parámetros que debe ser redefinido por el usuario en una subclase para invocar al método *inorden*.

A continuación se presenta el código correspondiente a la definición de la clase *CArbolBinE*:

```
///////////////////////////////
// Clase abstracta: árbol binario perfectamente equilibrado.
// Para utilizar los métodos proporcionados por esta clase,
```

```
// tendremos que crear una subclase de ella y redefinir los
// métodos: leerDatos, comparar, procesar y visitarInorden.
//
public abstract class CArbolBinE
{
    // Atributos del árbol binario
    protected CNodo raíz = null; // raíz del árbol

    // Nodo de un árbol binario
    private class CNodo
    {
        // Atributos
        private Object datos;          // referencia a los datos
        private CNodo izquierdo;       // raíz del subárbol izquierdo
        private CNodo derecho;         // raíz del subárbol derecho
        // Métodos
        public CNodo() {}             // constructor
    }

    // Métodos del árbol binario
    public CArbolBinE() {}          // constructor

    // El método siguiente debe ser redefinido en la subclase para
    // que permita leer los datos que serán referenciados por un
    // nodo del árbol. Devuelve el objeto de datos.
    public abstract Object leerDatos();

    // El método siguiente debe ser redefinido en una subclase para
    // que permita comparar dos nodos del árbol por el atributo
    // que necesitemos en cada momento.
    public abstract int comparar(Object obj1, Object obj2);

    // El método siguiente debe ser redefinido en la subclase para
    // que permita especificar las operaciones que se deseen
    // realizar con el nodo visitado.
    public abstract void procesar(Object obj);

    // El método siguiente debe ser redefinido en la subclase para
    // que invoque a "inorden" con los argumentos deseados.
    public abstract void visitarInorden();

    private CNodo construirArbol(int n)
    {
        // Construye un árbol de n nodos perfectamente equilibrado

        CNodo nodo = null;
        int ni = 0, nd = 0;

        if (n == 0)
            return null;
```

```
    else
    {
        ni = n / 2;      // nodos del subárbol izquierdo
        nd = n - ni - 1; // nodos del subárbol derecho
        nodo = new CNodo();
        nodo.datos = leerDatos();
        nodo.izquierdo = construirArbol(ni);
        nodo.derecho = construirArbol(nd);
        return nodo;
    }
}

public void construirArbolEquilibrado(int n)
{
    raíz = construirArbol(n);
}

private void buscar(Object obj, CNodo r, Object[] datos, int[] pos)
{
    // El método buscar permite acceder a un determinado nodo.
    // Si los datos especificados por "obj" se localizan en el
    // árbol referenciado por "r" a partir de la posición "pos[0]",
    // "buscar" devuelve en datos[0] la referencia a esos datos;
    // en otro caso, devuelve null.
    // Los nodos se consideran numerados (0, 1, 2, ...) según
    // el orden en el que son accedidos por el método "inorden".
    CNodo actual = r;

    if (actual != null && datos[0] == null)
    {
        buscar(obj, actual.izquierdo, datos, pos);
        if (comparar(obj, actual.datos) == 0)
            if (pos[0]-- == 0)
                datos[0] = actual.datos; // nodo encontrado
        buscar(obj, actual.derecho, datos, pos);
    }
}

public Object buscar(Object obj)
{
    return buscar(obj, 0);
}

public Object buscar(Object obj, int posición)
{
    Object[] datos = {null};
    int[] pos = {posición};
    buscar(obj, raíz, datos, pos);
    return datos[0];
}
```

```

public void inorder( CNodo r, boolean nodoRaiz )
{
    // El método recursivo inorder visita los nodos del árbol
    // utilizando la forma inorder; esto es, primero se visita
    // el subárbol izquierdo, después se visita la raíz, y por
    // último, el subárbol derecho.
    // Si el segundo argumento es true, la visita comienza
    // en la raíz independientemente del primer argumento.
    CNodo actual = null;

    if ( nodoRaiz )
        actual = raíz; // partir de la raíz
    else
        actual = r; // partir de un nodo cualquiera
    if ( actual != null )
    {
        inorder( actual.izquierdo, false ); // visitar subárbol izq.
        // Procesar los datos del nodo visitado
        procesar( actual.datos );
        inorder( actual.derecho, false ); // visitar subárbol dcho.
    }
}
}

```

El proceso de construcción lo lleva a cabo el método recursivo denominado *construirArbol*, el cual construye un árbol de n nodos (éste, es a su vez invocado por *construirArbolEquilibrado*). El prototipo de este método es:

```
CNodo construirArbol(int n)
```

Este método tiene un parámetro entero que se corresponde con el número de nodos del árbol y devuelve una referencia al nodo raíz del árbol construido. En realidad diremos que devuelve una referencia a cada subárbol construido lo que permite realizar los enlaces entre nodos. Observe que para cada nodo se ejecutan las dos sentencias siguientes:

```
nodo.izquierdo = construirArbol(ni);
nodo.derecho = construirArbol(nd);
```

que asignan a los atributos *izquierdo* y *derecho* de cada nodo, las referencias de sus subárboles izquierdo y derecho, respectivamente.

El método privado *buscar* también se ha declarado como un método recursivo. Permite acceder a unos datos determinados, comenzando la búsqueda desde cualquier nodo. Para facilitar la labor del usuario de la clase, se ha añadido a la interfaz pública de la misma dos sobrecargas de este método: una con un paráme-

tro a un objeto que encapsule los datos a buscar, y otra con un parámetro más, la posición del nodo a partir de la cual se quiere realizar la búsqueda. De esta forma se puede buscar un nodo aunque su clave de búsqueda esté repetida.

Utilización de la clase CArbolBinE

La clase *CArbolBinE* es una clase abstracta; por lo tanto, para hacer uso del soporte que proporciona para la construcción y manipulación de árboles binarios perfectamente equilibrados, tendremos que derivar una clase de ella y redefinir los métodos abstractos heredados: *leerDatos*, *comparar*, *procesar* y *visitarInorden*. La redefinición de estos métodos está condicionada a la clase de objetos que formarán parte del árbol.

Como ejemplo, vamos a construir un árbol binario perfectamente equilibrado en el que cada nodo haga referencia a un objeto de la clase *CDatos* ya utilizada anteriormente en este mismo capítulo.

El método *leerDatos* obtendrá los datos *nombre* y *nota*, a partir de ellos construirá un objeto *CDatos* y devolverá el objeto construido para su inserción en el árbol. Los métodos *comparar*, *procesar* y *visitarInorden* se definen igual que en la clase *CArbolBinarioDeBusqueda*.

Según lo expuesto, la clase *CArbolBinarioEquilibrado* derivada de *CArbolBinE* puede ser de la forma siguiente:

```
/////////////////////////////
// Clase derivada de la clase abstracta CArbolBinE. Redefine los
// métodos: leerDatos, comparar, procesar y visitarInorden.
//
public class CArbolBinarioEquilibrado extends CArbolBinE
{
    // Leer los datos que serán referenciados por un nodo del árbol.
    public Object leerDatos()
    {
        String nombre;
        double nota;

        System.out.print("nombre: "); nombre = Leer.dato();
        System.out.print("nota:   "); nota = Leer.datoDouble();
        return (Object)(new CDatos(nombre, nota));
    }

    // Permite comparar dos nodos del árbol por el atributo nombre.
    public int comparar(Object obj1, Object obj2)
    {
        String str1 = new String(((CDatos)obj1).obtenerNombre());
```

```

String str2 = new String(((CDatos)obj2).obtenerNombre());
return str1.compareTo(str2);
}

// Permite mostrar los datos del nodo visitado.
public void procesar(Object obj)
{
    String nombre = new String(((CDatos)obj).obtenerNombre());
    double nota = ((CDatos)obj).obtenerNota();
    System.out.println(nombre + " " + nota);
}

// Visitar los nodos del árbol.
public void visitarInorden()
{
    // Si el segundo argumento es true, la visita comienza
    // en la raíz independientemente del primer argumento.
    inorder(null, true);
}
}

```

Cuando se declare un objeto de la clase *CArbolBinarioEquilibrado*, el constructor de esta clase invoca al constructor *CArbolBinE* de su clase base, que creará un árbol vacío (*raíz = null*). El atributo *raíz* apunta siempre a la raíz del árbol.

Finalmente, escribiremos una aplicación *Test* que, utilizando la clase *CArbolBinarioEquilibrado*, cree un objeto *arbolbe* correspondiente a un árbol binario de búsqueda en el que cada nodo haga referencia a un objeto *CDatos*. De forma resumida la aplicación *Test*:

1. Creará un objeto *arbolbe* de la clase *CArbolBinarioEquilibrado*.
2. Construirá el árbol equilibrado de *n* nodos, enviando al objeto *arbolbe* el mensaje *construirArbolEquilibrado*.
3. Buscará un determinado nodo enviando al objeto *arbolbe* el mensaje *buscar*.
4. Finalmente, mostrará los datos almacenados en el árbol para comprobar que todo ha sucedido como esperábamos.

```

// Crear un árbol binario perfectamente equilibrado de n nodos
//
public class Test
{
    public static void main(String[] args)
    {
        CArbolBinarioEquilibrado arbolbe = new CArbolBinarioEquilibrado();
    }
}

```

```

        int numeroDeNodos;
        System.out.print("Número de nodos: ");
        numeroDeNodos = Leer.datoInt();
        arbolbe.construirArbolEquilibrado(numeroDeNodos);
        System.out.println();

        // Buscar datos
        String nombre;
        System.out.print("nombre a buscar: "); nombre = Leer.dato();
        CDatos obj = (CDatos)arbolbe.buscar(new CDatos(nombre, 0));
        if ( obj != null )
            System.out.println(obj.obtenerNombre() + " " +
                               obj.obtenerNota());
        else
            System.out.println("La búsqueda falló");

        // Mostrar los nodos del árbol
        System.out.println("\nArbol:");
        arbolbe.visitarInorden();
    }
}

```

CLASES RELACIONADAS DE LA BIBLIOTECA JAVA

Java soporta diferentes grupos de objetos, entre los que cabe destacar de forma genérica los siguientes:

- *Collection*. Una colección no tiene un orden especial y permite claves duplicadas.
- *List*. Una lista está ordenada y permite claves duplicadas. En unos casos los elementos se colocan en el orden en el que son añadidos y en otros, los mismos elementos asumen un orden natural.
- *Set*. Un conjunto no tiene un orden especial pero no permite claves duplicadas.
- *Map*. Un mapa utiliza un conjunto de claves no duplicadas (un índice) para acceder a los datos almacenados.

Pues bien, además de la clase **LinkedList** para trabajar con listas enlazadas, la biblioteca de Java proporciona en su paquete **java.util** las clases **TreeSet**, **TreeMap**, **HashSet**, **HashMap** y **HashTable**.

La clase **TreeSet** proporciona un conjunto ordenado, utilizando para el almacenamiento de los datos un árbol. Los elementos deben tener un orden asociado (saber qué elemento sigue a cuál) implementando la interfaz **Comparable** o proporcionando una clase **Comparator** para efectuar las comparaciones.

La clase **TreeMap** proporciona un mapa ordenado, utilizando para el almacenamiento de los datos un árbol. Igual que la clase **TreeSet**, los elementos deben tener un orden asociado.

Las clases **Hash...** proporcionan conjuntos de datos que no permiten duplicados y que utilizan algoritmos *hash* para el almacenamiento de los datos y para su posterior acceso.

EJERCICIOS RESUELTOS

- Realizar una aplicación que permita crear una lista lineal de elementos de cualquier tipo clasificados ascendentemente. La lista vendrá definida por un objeto de una clase abstracta que denominaremos *CListaLinealSEO* (Clase *Lista Lineal Simplemente Enlazada Ordenada*) y cada elemento de la lista será un objeto de la clase siguiente:

```
private class CElemento
{
    // Atributos
    private Object datos;
    private CElemento siguiente; // siguiente elemento
    // Métodos
    // ...
}
```

La clase *CListaLinealSEO* debe incluir los atributos:

```
private CElemento p = null;           // elemento de cabecera
private CElemento elemAnterior = null; // elemento anterior
private CElemento elemActual = null;  // elemento actual
```

El atributo *elemActual* hará referencia al elemento accedido y *elemAnterior* al anterior al actual, excepto cuando el elemento actual sea el primero, en cuyo caso ambas referencias señalarán a ese elemento. También incluirá los métodos:

```
public abstract int comparar(Object obj1, Object obj2);
public boolean listaVacia()
public Object buscar(Object obj)
public void añadir(Object obj)
public Object borrar(Object obj)
public Object obtenerPrimero()
public Object obtenerSiguiente()
```

Todos los métodos expuestos, excepto *listaVacia*, deben actualizar las referencias *elemActual* y *elemAnterior*.

El método *comparar* debe ser redefinido por el usuario en una subclase para especificar el tipo de comparación que se desea realizar con dos elementos de la lista. Según esto, debe devolver un entero indicando el resultado de la comparación (-1, 0 ó 1 si *obj1*<*obj2*, *obj1*=*obj2*, o *obj1*>*obj2*, respectivamente). Este método es invocado directamente por el método *buscar* e indirectamente por los métodos *añadir* y *borrar*, que invocan a *buscar*.

El método *listaVacía* devuelve **true** si la lista está vacía y **false** en caso contrario.

El método *buscar* localiza un elemento determinado en la lista. Tiene un parámetro para almacenar una referencia de tipo **Object** a los datos que permitirán localizar el elemento en la lista, y devuelve una referencia al área de datos del elemento, o bien **null** si la lista está vacía o no existe un elemento con esos datos.

El método *añadir* inserta un elemento en la lista en orden ascendente de una clave seleccionada del área de datos. Tiene un parámetro que es una referencia de tipo **Object** al objeto a añadir. No devuelve nada.

El método *borrar* borra un elemento de la lista. Tiene un parámetro para almacenar una referencia de tipo **Object** a los datos que permitirán localizar en la lista el elemento que se desea borrar. Devuelve una referencia al área de datos del elemento borrado, o bien **null** si la lista está vacía.

El método *obtenerPrimero* devuelve una referencia al área de datos del elemento primero, o bien **null** si la lista está vacía.

El método *obtenerSiguiente* devuelve una referencia al área de datos del elemento siguiente al actual, o bien **null** si la lista está vacía.

Según el enunciado, la clase *CListaLinealSEO* puede ser como se muestra a continuación:

```
///////////////////////////////
// Clase abstracta CListaLinealSEO:
//   Lista lineal simplemente enlazada ordenada ascendentemente.
//
public abstract class CListaLinealSEO
{
    // p: referencia al primer elemento de la lista.
    private CElemento p = null;           // elemento de cabecera
    private CElemento elemAnterior = null; // elemento anterior
    private CElemento elemActual = null;   // elemento actual

    // Elemento de una lista lineal simplemente enlazada
```

```

private class CElemento
{
    // Atributos
    private Object datos;
    private CElemento siguiente; // siguiente elemento
    // Métodos
    private CElemento() {} // constructor
    private CElemento(Object d, CElemento s) // constructor
    {
        datos = d;
        siguiente = s;
    }
}

public CListaLinealSE0() {} // constructor

// El método siguiente debe ser redefinido en una subclase para
// que permita comparar dos elementos de la lista por el atributo
// que necesitemos en cada momento.
public abstract int comparar(Object obj1, Object obj2);

public boolean listaVacia()
{
    return p == null;
}

public Object buscar(Object obj)
{
    // Buscar un elemento determinado en una lista ordenada.
    // El método almacena en elemActual la referencia del
    // elemento buscado y en elemAnterior la referencia del
    // elemento anterior.
    elemAnterior = elemActual = null;

    // Si la lista referenciada por p está vacía, retornar.
    if ( listaVacia() ) return null;
    // Si la lista no está vacía, encontrar el elemento.
    elemAnterior = p;
    elemActual = p;
    // Posicionarse en el elemento buscado.
    while (elemActual != null && comparar(obj, elemActual.datos) > 0)
    {
        elemAnterior = elemActual;
        elemActual = elemActual.siguiente;
    }
    if ( elemActual != null )
        return elemActual.datos;
    else
        return null;
}

```

```
public void añadir(Object obj)
{
    // Añadir un elemento en orden ascendente según una clave
    // proporcionada por obj.
    CElemento q = new CElemento(obj, null); // crear el elemento

    // Si la lista referenciada por p está vacía, añadirlo sin más
    if ( listaVacia() )
    {
        // Añadir el primer elemento
        p = q;
        elemAnterior = elemActual = p; // actualizar referencias
        return;
    }

    // Si la lista no está vacía, encontrar el punto de inserción
    buscar(obj); // establece los valores de elemAnterior y elemActual

    // Dos casos:
    // 1) Insertar al principio de la lista
    // 2) Insertar después del anterior (incluye insertar al final)
    if ( elemAnterior == elemActual ) // insertar al principio
    {
        q.siguiente = p;
        p = q; // cabecera
        elemAnterior = elemActual = p; // actualizar referencias
    }
    else // insertar después del anterior
    {
        q.siguiente = elemActual;
        elemAnterior.siguiente = q;
        elemActual = q; // actualizar referencia
    }
}

public Object borrar(Object obj)
{
    // Borrar un determinado elemento.
    // Si la lista está vacía, retornar.
    if ( listaVacia() ) return null;

    // Si la lista no está vacía, buscar el elemento.
    buscar(obj); // establece los valores de elemAnterior y elemActual
    // Dos casos:
    // 1) Borrar el primer elemento de la lista
    // 2) Borrar el siguiente a elemAnterior (elemActual)
    if ( elemActual == p ) // 1)
        p = p.siguiente; // cabecera
    else // 2)
        elemAnterior.siguiente = elemActual.siguiente;
```

```

Object borrado = elemActual.datos;
elemActual = elemActual.siguiente; // actualizar referencia
return borrado; // retornar el elemento borrado.
// El elemento referenciado por borrado será enviado a la
// basura al quedar desreferenciado, por tratarse de una
// variable local.
}

public Object obtenerPrimero()
{
    // Devolver una referencia a los datos del primer elemento.
    // Si la lista está vacía, devolver null.
    if ( listaVacia() ) return null;
    elemActual = elemAnterior = p;
    return p.datos;
}

public Object obtenerSiguiente()
{
    // Devolver una referencia a los datos del elemento siguiente
    // al actual y hacer que éste sea el actual.
    // Si la lista está vacía, devolver null.
    if ( listaVacia() ) return null;
    // Avanzar un elemento
    elemAnterior = elemActual;
    elemActual = elemActual.siguiente;
    if ( elemActual != null )
        return elemActual.datos;
    else
        return null;
}
}
/////////////////////////////////////////////////////////////////

```

En la lista que crearemos a partir de la clase anterior vamos a almacenar objetos de la clase:

```

public class CDatos
{
    // Atributos
    private String nombre;
    private double nota;
    // Métodos
    public CDatos() {} // constructor sin parámetros
    public CDatos(String nom, double n) // constructor con parámetros
    {
        nombre = nom;
        nota = n;
    }
}

```

```

public void asignarNombre(String nom)
{
    nombre = nom;
}

public String obtenerNombre()
{
    return nombre;
}

public void asignarNota(double n)
{
    nota = n;
}

public double obtenerNota()
{
    return nota;
}
}

```

Pero, para utilizar la clase abstracta *CListaLinealSEO* tenemos que derivar de ella otra clase, por ejemplo *CListaLinealSEOrdenada*, que redefina el método *comparar* para que permita comparar dos objetos *CDatos* por el atributo *nombre*:

```

public class CListaLinealSEOrdenada extends CListaLinealSEO
{
    // Permite comparar dos elementos de la lista por
    // el atributo nombre.
    public int comparar(Object obj1, Object obj2)
    {
        String str1 = new String(((CDatos)obj1).obtenerNombre());
        String str2 = new String(((CDatos)obj2).obtenerNombre());
        return str1.compareTo(str2);
    }
}

```

Finalmente, realizamos una aplicación que utilizando la clase anterior cree una lista lineal simplemente enlazada y ordenada, de objetos *CDatos*:

```

///////////////////////////////
// Crear una lista lineal simplemente enlazada
//
public class Test
{
    public static void mostrarLista(CListaLinealSEOrdenada lse)
    {
        // Mostrar todos los elementos de la lista
        CDatos obj = (CDatos)lse.obtenerPrimero();
    }
}

```

```

int i = 1;
while (obj != null)
{
    System.out.println(i++ + ". - " + obj.obtenerNombre() + " " +
        obj.obtenerNota());
    obj = (CDatos)lse.obtenerSiguiente();
}
}

public static void main(String[] args)
{
    // Crear una lista lineal vacía
    CListaLinealSEOrdenada lse = new CListaLinealSEOrdenada();
    // Leer datos y añadirlos a la lista
    CDatos obj;
    String nombre;
    double nota;
    int i = 0;
    System.out.println("Introducir datos. Finalizar con Ctrl+Z.");
    System.out.print("nombre: ");
    while ((nombre = Leer.dato()) != null)
    {
        System.out.print("nota: ");
        nota = Leer.datoDouble();
        lse.agregar(new CDatos(nombre, nota));
        System.out.print("nombre del alumno a borrar: ");
    }
    System.out.println("\n");

    // Borrar un elemento determinado
    System.out.print("nombre del alumno a borrar: ");
    nombre = Leer.dato();
    obj = (CDatos)lse.borrar(new CDatos(nombre, 0));
    if (obj == null)
        System.out.println("Error: elemento no borrado");
    // Modificar un elemento
    System.out.print("nombre del alumno a modificar: ");
    nombre = Leer.dato();
    obj = (CDatos)lse.buscar(new CDatos(nombre, 0));
    System.out.println("Nombre: " + obj.obtenerNombre() +
        ", nota: " + obj.obtenerNota());
    System.out.print("nota nueva: ");
    nota = Leer.datoDouble();
    obj.asignarNota(nota);
    // Mostrar todos
    System.out.println("Lista:");
    mostrarLista(lse);
}
}

```

2. Escribir una aplicación para que utilizando una pila, simule una calculadora capaz de realizar las operaciones de +, -, * y /. La mayoría de las calculadoras aceptan la notación *infija* y unas pocas la notación *postfija*. En estas últimas, para sumar 10 y 20 introduciríamos primero 10, después 20 y por último el +. Cuando se introducen los operandos, se colocan en una pila y cuando se introduce el operador, se sacan dos operandos de la pila, se calcula el resultado y se introduce en la pila. La ventaja de la notación postfija es que expresiones complejas pueden evaluarse fácilmente sin mucho código. La calculadora del ejemplo propuesto utilizará la notación *postfija*.

De forma resumida, el programa realizará las siguientes operaciones:

- Leerá un dato, operando u operador, y lo almacenará en la variable *oper*.
- Analizará *oper*; si se trata de un operando lo mete en la pila y si se trata de un operador saca los dos últimos operandos de la pila, realiza la operación indicada por dicho operador y mete el resultado en la pila para poder utilizarlo como operando en una posible siguiente operación.

Para realizar esta aplicación utilizaremos las clases *CPila* derivada de *CListaCircularSE*, *CDatos* y *CLeer*. Como estas clases ya han sido implementadas, en este ejercicio nos limitaremos a utilizar los recursos que proporcionan.

El programa completo se muestra a continuación:

```
///////////////////////////////
// Calculadora utilizando una pila. Esta aplicación, además de las
// clases necesarias de la biblioteca de Java, utiliza las clases:
// CPila derivada de CListaCircularSE, CDatos y CLeer.
//
public class Test
{
    private static CPila pila = new CPila(); // pila de operandos
    private static double[] operando = {0, 0}; // operando 0 y 1

    public static void obtenerOperandos()
    {
        if (pila.tamaño() < 2) throw new NullPointerException();
        operando[1] = ((Double)pila.sacarDePila()).doubleValue();
        operando[0] = ((Double)pila.sacarDePila()).doubleValue();
    }

    public static void main(String[] args)
    {
        // oper almacena la entrada realizada desde el teclado
        String oper = null;

        System.out.println("Operaciones: + - * /\n");
    }
}
```

```
System.out.println("Forma de introducir los datos:");
System.out.println(">primer operando 1 [Entrar]");
System.out.println(">segundo operando 2 [Entrar]");
System.out.println(">operador [Entrar]\n");
System.out.println("Para salir pulse q\n");

do
{
    try
    {
        System.out.print("> ");
        oper = Leer.dato();           // leer un operando o un operador
        switch (oper.charAt(0))      // verificar el primer carácter
        {
            case '+':
                obtenerOperandos();
                System.out.println(operando[0] + operando[1]);
                pila.meterEnPila(new Double(operando[0]+operando[1]));
                break;
            case '-':
                obtenerOperandos();
                System.out.println(operando[0] - operando[1]);
                pila.meterEnPila(new Double(operando[0]-operando[1]));
                break;
            case '*':
                obtenerOperandos();
                System.out.println(operando[0] * operando[1]);
                pila.meterEnPila(new Double(operando[0]*operando[1]));
                break;
            case '/':
                obtenerOperandos();
                if (operando[1] == 0)
                {
                    System.out.println("\nError: división por cero");
                    break;
                }
                System.out.println(operando[0] / operando[1]);
                pila.meterEnPila(new Double(operando[0]/operando[1]));
                break;
            case 'q':
                // salir
                break;
            default : // es un operando
                pila.meterEnPila(new Double(oper));
        }
    }
    catch(NumberFormatException e)
    {
        System.out.print("Error: dato no es válido. Teclee otro: ");
    }
}
```

```

        catch(NullPointerException e)
        {
            System.out.print("Error: teclee " + (2-pila.tamaño()) +
                            " operando(s) más");
        }
    }
    while (oper.charAt(0) != 'q');
}
}

```

3. Escribir una aplicación que permita calcular la frecuencia con la que aparecen las palabras en un fichero de texto. La forma de invocar al programa será:

`java Palabras fichero_de_texto`

donde *fichero_de_texto* es el nombre del fichero de texto del cual deseamos obtener la estadística.

El proceso de contabilizar las palabras que aparezcan en el texto de un determinado fichero, lo podemos realizar de la forma siguiente:

- Se lee la información del fichero y se descompone en palabras, entendiendo por palabra una secuencia de caracteres delimitada por espacios en blanco, tabuladores, signos de puntuación, etc.
- Cada palabra deberá insertarse por orden alfabético ascendente junto con un contador que indique su número de apariciones, en el nodo de una estructura en árbol. Esto facilitará la búsqueda.
- Una vez construido el árbol de búsqueda, se presentará por pantalla una estadística con el siguiente formato:

```

...
nombre = 1
obtener = 1
palabras = 1
permita = 1
programa = 1
que = 2
queremos = 1
será = 1
estadística = 1
texto = 2
un = 1
una = 1

```

Total palabras: 44

Total palabras diferentes: 35

Según lo expuesto, cada nodo del árbol tendrá que hacer referencia a un área de datos que incluya tanto la palabra como el número de veces que apareció en el texto. Estos datos serán los atributos de una clase *CDatos* definida así:

```
public class CDatos
{
    // Atributos
    private String palabra;
    private int contador;

    // Métodos
    public CDatos() {}           // constructor sin parámetros

    public CDatos(String pal) // constructor con un parámetro
    {
        palabra = pal;
        contador = 0;
    }

    public CDatos(String pal, int cont) // constructor con dos params
    {
        palabra = pal;
        contador = cont;
    }

    public void asignarPalabra(String pal)
    {
        palabra = pal;
    }

    public String obtenerPalabra()
    {
        return palabra;
    }

    public void asignarContador(int cont)
    {
        contador = cont;
    }

    public int obtenerContador()
    {
        return contador;
    }
}
```

El árbol de búsqueda que tenemos que construir será un objeto de la clase *CArbolBinarioDeBusqueda* derivada de *CArbolBinB*. Recuerde que la clase *CArbolBinB* fue implementada anteriormente en este mismo capítulo, al hablar de árboles binarios de búsqueda. La razón de por qué derivamos un clase de *CArbolBinB* es porque esta clase es abstracta, con la intención de redefinir los métodos que procesan información contenida en el área de datos referenciada por cada nodo, que en nuestro caso se corresponde con objetos *CDatos*.

```
///////////////////////////////
// Clase derivada de la clase abstracta CArbolBinB. Redefine los
// métodos: comparar, procesar y visitarInorden.
//
public class CArbolBinarioDeBusqueda extends CArbolBinB
{
    public int totalPalabras = 0;
    public int totalPalabrasDiferentes = 0;

    // Permite comparar dos nodos del árbol por el atributo
    // nombre.
    public int comparar(Object obj1, Object obj2)
    {
        String str1 = new String(((CDatos)obj1).obtenerPalabra());
        String str2 = new String(((CDatos)obj2).obtenerPalabra());
        return str1.compareTo(str2);
    }

    // Permite mostrar los datos del nodo visitado.
    public void procesar(Object obj)
    {
        String palabra = new String(((CDatos)obj).obtenerPalabra());
        int contador = ((CDatos)obj).obtenerContador();
        System.out.println(palabra + " = " + contador);
        totalPalabras += contador;
        totalPalabrasDiferentes++;
    }

    // Visitar los nodos del árbol.
    public void visitarInorden()
    {
        // Si el segundo argumento es true, la visita comienza
        // en la raíz independientemente del primer argumento.
        inorder(null, true);
    }
}
/////////////////////////////
```

Se puede observar que el método *procesar* de la clase *CArbolBinarioDeBusqueda*, además de visualizar la información almacenada en el objeto *CDatos* refe-

renciado por el nodo visitado, contabiliza el número total de palabras del texto procesado y el número total de palabras diferentes (esto es, como si todas hubieran aparecido sólo una vez en el texto). El resto de los métodos ya fueron explicados al hablar de árboles binarios de búsqueda.

Sólo queda construir una aplicación que cree un objeto de la clase *CArbolBinarioDeBusqueda* a partir de las palabras almacenadas en un fichero y presente los resultados pedidos. El código de esta aplicación se va a apoyar en tres métodos: **main**, *leerFichero* y *palabras*.

El método **main** verifica que, cuando se ejecute la aplicación, se haya pasado como parámetro el nombre del fichero de texto, invoca al método *leerFichero* y una vez construido el árbol, lo recorre para visualizar los resultados pedidos.

El método *leerFichero* abre el fichero y lo lee línea a línea. Cada línea leída será pasada como argumento al método *palabras* para su descomposición en palabras con el fin de añadirlas al árbol binario de búsqueda que ha sido declarado como un atributo de la clase aplicación. Para descomponer una línea en palabras utilizaremos los recursos proporcionados por la clase **StringTokenizer** del paquete **util** de Java (esta clase fue explicada en el capítulo 7).

El código completo de la aplicación que hemos denominado *Palabras*, se muestra a continuación:

```
import java.io.*;
import java.util.*;
// Utilizar un árbol de búsqueda para obtener la frecuencia con la
// que aparecen las palabras en un fichero de texto.
// Esta aplicación, además de las clases necesarias de la
// biblioteca de Java, utiliza las clases: CArbolBinarioDeBusqueda
// derivada de CArbolBinB y CDatos.
//
public class Palabras
{
    private static CArbolBinarioDeBusqueda arbolbb =
        new CArbolBinarioDeBusqueda();

    public static void palabras(String linea)
    {
        // Descomponer linea en palabras
        StringTokenizer cadena;
        cadena = new StringTokenizer(linea, " .,:;\n\r\t\f");

        String palabra;
        CDatos obj;
```

```
        while (cadena.hasMoreTokens())
        {
            palabra = cadena.nextToken();
            if ((obj = (CDatos)arbolbb.buscar(new CDatos(palabra))) == null)
                arbolbb.insertar(new CDatos(palabra, 1));
            else
                obj.asignarContador(obj.obtenerContador()+1);
        }
    }

public static void leerFichero(String nombrefich)
{
    // Definiciones de variables
    File fichFuente = new File(nombrefich);
    BufferedReader flujoE = null;

    try
    {
        // Asegurarse de que el fichero, existe y se puede leer
        if (!fichFuente.exists() || !fichFuente.isFile())
        {
            System.err.println("No existe el fichero " + nombrefich);
            return;
        }
        if (!fichFuente.canRead())
        {
            System.err.println("El fichero " + nombrefich +
                               " no se puede leer");
            return;
        }

        // Abrir un flujo de entrada desde el fichero fuente
        FileInputStream fis = new FileInputStream(fichFuente);
        InputStreamReader isr = new InputStreamReader(fis);
        flujoE = new BufferedReader(isr);

        // Buscar cadena en el fichero fuente
        String linea;

        while ((linea = flujoE.readLine()) != null)
        {
            // Si se alcanzó el final del fichero,
            // readLine devuelve null
            palabras(linea);
        }
    }
    catch(IOException e)
    {
        System.out.println("Error: " + e.getMessage());
    }
}
```

```

        finally
    {
        // Cerrar el flujo
        try
        {
            if (flujoE != null) flujoE.close();
        }
        catch(IOException e)
        {
            System.out.println("Error: " + e.toString());
        }
    }
}

public static void main(String[] args)
{
    // main debe recibir un parámetro: el nombre del fichero
    // java Palabras palabras.txt

    if (args.length < 1)
        System.err.println("Sintaxis: java Palabras <fichero_de_texto>");
    else
        leerFichero(args[0]);

    arbolbb.visitarInorden();
    System.err.println();
    System.err.println("Total palabras: " + arbolbb.totalPalabras);
    System.err.println("Total palabras diferentes: " +
                       arbolbb.totalPalabrasDiferentes);
}
}

```

EJERCICIOS PROPUESTOS

1. Se quiere escribir un programa para manipular ecuaciones algebraicas o polinómicas dependientes de las variables x e y . Por ejemplo:

$$2x^3y - xy^3 + 8.25 \text{ más } 5x^5y - 2x^3y + 7x^2 - 3 \text{ igual a } 5x^5y + 7x^2 - xy^3 + 5.25$$

Cada término del polinomio será representado por un objeto de una clase *CTermino* y cada polinomio por un objeto que sea una lista lineal simplemente enlazada ordenada, de elementos *CTermino*.

La clase *CTermino* ya fue desarrollada en el apartado “ejercicios resueltos” del capítulo 10. También se implementó una clase *CPolinomio*. Esta clase debe ser ahora reemplazada por una lista lineal simplemente enlazada ordenada.

2. En un fichero tenemos almacenados los nombres y las notas medias de los alumnos de un determinado curso. La estructura de cada uno de los registros del fichero se corresponde con los atributos de una clase como la siguiente:

```
public class CRegistro
{
    // Atributos
    private String nombre;
    private float nota;
    // Métodos
    // ...
}
```

Queremos leer los datos de este fichero para construir una estructura de datos en memoria que se ajuste a un árbol binario *de búsqueda* perfectamente *equilibrado*. Para ello, es aconsejable ordenar el fichero antes de crear el árbol. Esto facilita la creación del árbol binario con los dos requisitos impuestos: que sea de búsqueda y perfectamente equilibrado. En este ejercicio supondremos que partimos de un fichero ordenado. Más adelante, en el capítulo de “Algoritmos” veremos cómo se ordena un fichero.

Cuando se muestre la información almacenada en el árbol, el listado de los nombres y sus correspondientes notas debe aparecer en orden ascendente. Por definición de árbol de búsqueda, para todo nodo, las claves menores que la del propio nodo forman el subárbol izquierdo y las mayores, el subárbol derecho. Según esto, la clave menor se encuentra en el nodo más a la izquierda y la clave mayor en el nodo más a la derecha del árbol. Por lo tanto, para visualizar los nombres en orden ascendente tendremos que recorrer el árbol en *inorden*. Entonces, si pensamos en el proceso inverso, esto es, si partimos de un fichero con las claves ordenadas y construimos un árbol perfectamente equilibrado tenemos que utilizar la forma *inorden* para conseguir al mismo tiempo que el árbol sea de búsqueda. Es decir, el método que cree el árbol debe incluir los siguientes procesos en el orden mostrado:

```
crear el subárbol izquierdo
leer un registro del fichero y asignárselo al nodo actual
crear el subárbol derecho
```

Vemos que primero hay que construir el subárbol izquierdo, después la raíz y por último el subárbol derecho. Como las claves contenidas en el fichero están ordenadas, la clave menor se almacenará en el nodo más a la izquierda y la mayor en el nodo más a la derecha, dando así lugar a un árbol de búsqueda, además de perfectamente equilibrado.

3. El filtro *sort* lee líneas de texto del fichero estándar de entrada y las presenta en orden alfabético en el fichero estándar de salida. El ejemplo siguiente muestra la forma de utilizar *sort*:

```
sort
lo que puede hacerse
en cualquier momento
no se hará
en ningún momento.
(eof)
en cualquier momento
en ningún momento.
lo que puede hacerse
no se hará
```

Se desea escribir un programa de nombre *Ordenar*, que actúe como el filtro *sort*. Para ordenar las distintas líneas vamos a ir insertándolas en un árbol binario de búsqueda, de tal forma que al recorrerlo podamos presentar las líneas en orden alfabético. El programa se ejecutará utilizando la siguiente sintaxis:

```
java Ordenar fichero_de_texto [-r]
```

Si se especifica el atributo opcional *-r*, las líneas del fichero serán presentadas en orden alfabético descendente; si no se especifica, entonces se presentarán en orden alfabético ascendente.

CAPÍTULO 14

© F.J.Ceballos/RA-MA

ALGORITMOS

En este capítulo vamos a exponer cómo resolver problemas muy comunes en programación. El primero que nos vamos a plantear es la recursión; se trata de un problema cuyo planteamiento forma parte de su solución. El segundo problema que vamos a abordar es la ordenación de objetos en general; la ordenación es tan común que no necesita explicación; algo tan cotidiano como una guía telefónica, es un ejemplo de una lista clasificada. El localizar un determinado teléfono exige una búsqueda por algún método; el problema de búsqueda será el último que resolvemos.

RECUSIVIDAD

Se dice que un proceso es recursivo si forma parte de sí mismo, o sea que se define en función de sí mismo. Ejemplos típicos de recursión los podemos encontrar frecuentemente en problemas matemáticos, en estructuras de datos y en muchos otros problemas.

La recursión es un proceso extremadamente potente, pero consume muchos recursos, razón por la que la analizaremos detenidamente, para saber cuándo y cómo aplicarla. De este análisis deduciremos que aunque un problema por definición sea recursivo, no siempre será el método de solución más adecuado.

En las aplicaciones prácticas, antes de poner en marcha un proceso recursivo es necesario demostrar que el nivel máximo de recursión, esto es, el número de veces que se va a llamar a sí mismo, es no sólo finito, sino realmente pequeño. La razón es que se necesita cierta cantidad de memoria para almacenar el estado del proceso cada vez que se abandona temporalmente, debido a una llamada para ejecutar otro proceso que es él mismo. El estado del proceso de cálculo en curso hay

que almacenarlo para recuperarlo cuando se acabe la nueva ejecución del proceso y haya que reanudar la antigua.

En términos de un lenguaje de programación, un método es recursivo cuando se llama a sí mismo.

Un ejemplo es el método de Ackerman, A , el cual está definido para todos los valores enteros no negativos m y n de la forma siguiente:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m,0) &= A(m-1,1) \quad (m > 0) \\ A(m,n) &= A(m-1,A(m,n-1)) \quad (m,n > 0) \end{aligned}$$

El pseudocódigo que especifica cómo solucionar este problema aplicando la recursión, es el siguiente:

```
<método A(m,n)>
  IF (m es igual a 0) THEN
    devolver como resultado n+1
  ELSE IF (n es igual a 0) THEN
    devolver como resultado A(m-1,1)
  ELSE
    devolver como resultado A(m-1,A(m,n-1))
  ENDIF
END <método A(m,n)>
```

A continuación presentamos este método como parte de una clase *CRecursion*:

```
public class CRecursion
{
  // Método recursivo de Ackerman:
  //   A(0,n) = n+1
  //   A(m,0) = A(m-1,1)           (m > 0)
  //   A(m,n) = A(m-1,A(m,n-1))   (m,n > 0)
  public static int Ackerman(int m, int n)
  {
    if (m == 0)
      return n+1;
    else if (n == 0)
      return Ackerman(m-1, 1);
    else
      return Ackerman(m-1, Ackerman(m,n-1));
  }
}
```

Para probar cómo funciona este algoritmo podemos escribir la aplicación siguiente:

```

public class Test
{
    public static void main(String[] args)
    {
        int m, n, a;
        System.out.println("Cálculo de A(m,n)=A(m-1,A(m,n-1))\n");
        System.out.print("Valor de m: "); m = Leer.datoInt();
        System.out.print("Valor de n: "); n = Leer.datoInt();
        a = CRecursion.Ackerman(m,n);
        System.out.println("\nA(" + m + "," + n + ") = " + a);
    }
}

```

Supongamos ahora que nos planteamos el problema de resolver el método de Ackerman, pero sin aplicar la recursión. Esto nos exigirá salvar las variables necesarias del proceso en curso, cada vez que el método se llame a sí mismo, con el fin de poder reanudarlo cuando finalice el nuevo proceso invocado.

La mejor forma de hacer esto es utilizar una pila, con el fin de almacenar los valores m y n cada vez que se invoque el método para una nueva ejecución y tomar estos valores de la cima de la pila, cuando esta nueva ejecución finalice, con el fin de reanudar la antigua.

El seudocódigo para este método puede ser el siguiente:

```

<método A(m,n)>
Utilizar una pila para almacenar los valores de m y n
Iniciar la pila con los valores m,n
DO
    Tomar los datos de la parte superior de la pila
    IF (m es igual a 0) THEN
        Amn = n+1
        IF (pila no vacía)
            sacar de la pila los valores: m, n
            meter en la pila los valores: m, Amn
        ELSE
            devolver como resultado Amn
        ENDIF
    ELSE IF (n es igual a 0) THEN
        meter en la pila los valores: m-1,1
    ELSE
        meter en la pila los valores: m-1, Amn
        meter en la pila los valores: m,n-1
    ENDIF
WHILE (true)
END <método A(m,n)>

```

A continuación presentamos el código correspondiente a este método que hemos denominado *AckermanNR*. Dicho método se ha incluido en la interfaz de la clase *CRecursion* anterior y utiliza la clase *CPila*, implementada en el capítulo de “Estructuras dinámicas”, para crear una pila que almacene los valores *m* y *n* cada vez que es invocado para una nueva ejecución.

```

public static int AckermanNR(int m, int n)
{
    CPila pila = new CPila(); // pila de elementos (m,n)
    CDatos dato;
    int Ackerman_m_n = 0;
    pila.meterEnPila(new CDatos(m, n));
    while (true)
    {
        // Tomar los datos de la cima de la pila
        dato = (CDatos)pila.sacarDePila();
        m = dato.obtenerM();
        n = dato.obtenerN();
        if (m == 0) // Ackerman(0,n) = n+1
        {
            Ackerman_m_n = n+1;
            if (pila.tamaño() != 0)
            {
                // Sacar m y n de la pila
                dato = (CDatos)pila.sacarDePila();
                m = dato.obtenerM();
                n = dato.obtenerN();
                // Meter m y Ackerman_m_n en la pila
                pila.meterEnPila(new CDatos(m, Ackerman_m_n));
            }
            else
                return Ackerman_m_n;
        }
        else if (n == 0) // Ackerman(m-1,1)
        {
            // Meter m-1 y 1 en la pila
            pila.meterEnPila(new CDatos(m-1, 1));
        }
        else // Ackerman(m-1,Ackerman(m,n-1))
        {
            // Meter m-1 y Ackerman_m_n en la pila
            pila.meterEnPila(new CDatos(m-1, Ackerman_m_n));
            // Meter m y n-1 en la pila
            pila.meterEnPila(new CDatos(m, n-1));
        }
    }
}

```

Según se puede observar, los valores de *m* y *n* son encapsulados por objetos de la clase *CDatos* definida así:

```

public class CDatos
{
    // Atributos
    private int m, n;

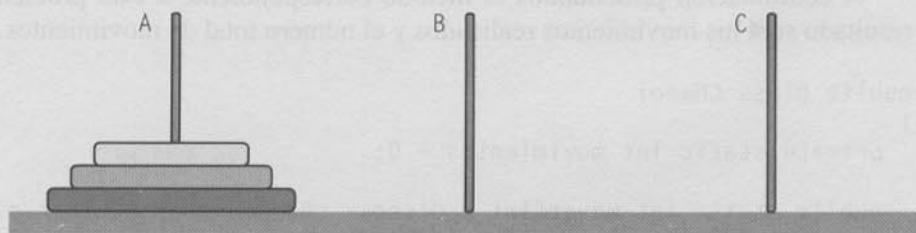
    // Métodos
    public CDatos(int im, int in) // constructor con parámetros
    {
        m = im;
        n = in;
    }

    public int obtenerM()
    {
        return m;
    }

    public int obtenerN()
    {
        return n;
    }
}

```

Un proceso en el que es realmente eficaz aplicar la recursión es el problema de las *torres de Hanoi*. Este problema consiste en tres barras verticales *A*, *B* y *C* y n discos, de diferentes tamaños, apilados inicialmente sobre la barra *A*, en orden de tamaño decreciente.



El objetivo es mover los discos desde la barra *A* a la *C*, conservando su orden, bajo las siguientes reglas:

1. Se moverá un sólo disco cada vez.
2. Un disco no puede situarse sobre otro más pequeño.
3. Se utilizará la barra *B* como pila auxiliar.

Una posible solución, es el algoritmo recursivo que se muestra a continuación:

1. Mover $n-1$ discos de la barra *A* a la *B* (el disco n es el del fondo).

2. Mover el disco n de la barra A a la C , y
3. Mover los $n-1$ discos de la barra B a la C .

Resumiendo estas condiciones en un cuadro obtenemos:

	<i>nº discos</i>	<i>origen</i>	<i>otra torre</i>	<i>destino</i>
<i>inicialmente</i>	n	A	B	C
1	$n-1$	A	C	B
2	1	A	B	C
3	$n-1$	B	A	C

El método a realizar será mover *n discos de origen a destino*:

```
mover(n_discos, origen, otratorre, destino);
```

El seudocódigo para este programa puede ser el siguiente:

```
<método mover(n_discos, A, B, C)>
IF (n_discos es mayor que 0) THEN
    mover(n_discos-1, A, C, B)
    mover(disco_n, A, B, C)
    mover(n_discos-1, B, A, C)
ENDIF
END <método mover>
```

A continuación presentamos el método correspondiente a este problema. El resultado será los movimientos realizados y el número total de movimientos.

```
public class CHanoi
{
    private static int movimientos = 0;

    public static int mover(int n_discos, char a, char b, char c)
    {
        if (n_discos > 0)
        {
            mover(n_discos-1, a, c, b);
            System.out.println("mover disco de " + a + " a " + c);
            movimientos++;
            mover(n_discos-1, b, a, c);
        }
        return movimientos;
    }
}
```

Para probar cómo funciona este método escribimos la aplicación siguiente:

```

public class Test
{
    public static void main(String[] args)
    {
        int n_discos, movimientos;
        System.out.print("Número de discos: ");
        n_discos = Leer.datoInt();
        movimientos = CHanoi.mover(n_discos, 'A', 'B', 'C');
        System.out.println("\nmovimientos efectuados: " + movimientos);
    }
}

```

Si ejecuta la aplicación anterior para $n_discos = 3$, el resultado será el siguiente:

```

Número de discos : 3
mover disco de A a C
mover disco de A a B
mover disco de C a B
mover disco de A a C
mover disco de B a A
mover disco de B a C
mover disco de A a C
movimientos efectuados: 7

```

Como ejercicio se propone realizar el método *mover* sin utilizar recursión.

ORDENACIÓN DE DATOS

Uno de los procedimientos más comunes y útiles en el procesamiento de datos, es la ordenación de los mismos. Se considera ordenar al proceso de reorganizar un conjunto dado de objetos en una secuencia determinada. El objetivo de este proceso generalmente es facilitar la búsqueda de uno o más elementos pertenecientes a un conjunto. Son ejemplos de datos ordenados las listas de los alumnos matriculados en una cierta asignatura, las listas del censo, los índices alfabéticos de los libros, las guías telefónicas, etc. Esto quiere decir que muchos problemas están relacionados de alguna forma con el proceso de ordenación. Es por lo que la ordenación es un problema importante a considerar.

La ordenación, tanto numérica como alfanumérica, sigue las mismas reglas que empleamos nosotros en la vida normal. Esto es, un dato numérico es mayor que otro cuando su valor es más grande, y una cadena de caracteres es mayor que otra cuando está después por orden alfabético.

Podemos agrupar los métodos de ordenación en dos categorías: ordenación de matrices u ordenación interna (cuando los datos se guardan en memoria interna) y ordenación de ficheros u ordenación externa (cuando los datos se guardan en memoria externa; generalmente en discos).

En este apartado no se trata de analizar exhaustivamente todos los métodos de ordenación y ver sus prestaciones de eficiencia, rapidez, etc. sino que simplemente analizamos desde el punto de vista práctico los métodos más comunes para ordenación de matrices y de ficheros.

Método de la burbuja

Hay muchas formas de ordenar datos, pero una de las más conocidas es la ordenación por el método de la burbuja.

Veamos a continuación el algoritmo correspondiente a este método para ordenar una lista de menor a mayor, partiendo de que los datos a ordenar están almacenados en una matriz de n elementos:

1. Comparamos el primer elemento con el segundo, el segundo con el tercero, el tercero con el cuarto, etc. Cuando el resultado de una comparación sea "mayor que", se intercambian los valores de los elementos comparados. Con esto conseguimos llevar el valor mayor a la posición n .
2. Repetimos el punto 1, ahora para los $n-1$ primeros elementos de la lista. Con esto conseguimos llevar el valor mayor de éstos a la posición $n-1$.
3. Repetimos el punto 1, ahora para los $n-2$ primeros elementos de la lista y así sucesivamente.
4. La ordenación estará realizada cuando al repetir el *iésimo* proceso de comparación no haya habido ningún intercambio o, en el peor de los casos, después de repetir el proceso de comparación descrito $n-1$ veces.

El seudocódigo para este algoritmo puede ser el siguiente:

```
<método ordenar(matriz "a" de "n" elementos)>
["a" es un matriz cuyos elementos son  $a_0, a_1, \dots, a_{n-1}$ ]
  n = n-1
  DO WHILE ("a" no esté ordenado y n > 0 )
    i = 1
    DO WHILE ( i <= n )
      IF ( a[i-1] > a[i] ) THEN
        permutar a[i-1] con a[i]
```

```

ENDIF
i = i+1
ENDDO
n = n-1
ENDDO
END <clasificar>

```

La clase siguiente incluye el método *ordenar* que utiliza este algoritmo para ordenar una matriz de tipo **double** o de tipo **String**.

```

///////////////////////////////
// Ordenación por el método de la burbuja. El método "ordenar" se
// sobrecarga dos veces: una para ordenar una matriz de tipo
// double y otra para ordenar una matriz de tipo String.
//
public class CMatriz
{
    public static void ordenar(double[] m)
    {
        double aux;
        int i, número_de_elementos = m.length;
        boolean s = true;

        while (s && (--número_de_elementos > 0))
        {
            s = false; // no permutación
            for (i = 1; i <= número_de_elementos; i++)
                // i el elemento (i-1) es mayor que el (i) ?
                if (m[i-1] > m[i])
                {
                    // permutar los elementos (i-1) e (i)
                    aux = m[i-1];
                    m[i-1] = m[i];
                    m[i] = aux;
                    s = true; // permutación
                }
        }
    }

    public static void ordenar(String[] m)
    {
        String aux;
        int i, número_de_elementos = m.length;
        boolean s = true;

        while (s && (--número_de_elementos > 0))
        {
            s = false; // no permutación
            for (i = 1; i <= número_de_elementos; i++)
                // i el elemento (i-1) es mayor que el (i) ?

```

```

        if (m[i-1].compareTo(m[i]) > 0)
        {
            // permutar los elementos (i-1) e (i)
            aux = m[i-1];
            m[i-1] = m[i];
            m[i] = aux;
            s = true; // permutación
        }
    }
}
/////////////////////////////////////////////////////////////////

```

Observe que *s* inicialmente vale **false** para cada iteración y toma el valor **true** cuando al menos se efectúa un cambio entre dos elementos. Si en una exploración a lo largo de la lista no se efectúa cambio alguno, *s* permanecerá valiendo **false**, lo que indica que la lista está ordenada, terminando así el proceso.

Cuando se analiza un método de ordenación, hay que determinar cuántas comparaciones e intercambios se realizan para el caso más favorable, para el caso medio y para el caso más desfavorable.

En el método de la burbuja se realizan $(n-1)(n/2)=(n^2-n)/2$ comparaciones en el caso más desfavorable, donde *n* es el número de elementos a ordenar. Para el caso más favorable (la lista está ordenada) el número de intercambios es 0. Para el caso medio es $3(n^2-n)/4$; hay tres intercambios por cada elemento desordenado. Y para el caso menos favorable, el número de intercambios es $3(n^2-n)/2$. El análisis matemático que conduce a estos valores, queda fuera del propósito de este libro. El tiempo de ejecución es un múltiplo de n^2 y está directamente relacionado con el número de comparaciones y de intercambios.

La siguiente aplicación ordena una matriz **double** y otra de tipo **String** utilizando el método *ordenar* de la clase *CMatriz*.

```

public class Test
{
    public static void main(String[] args)
    {
        // Matriz numérica
        double[] m = {3,2,1,5,4};
        CMatriz.ordenar(m);
        for (int i = 0; i < m.length; i++)
            System.out.print(m[i] + " ");
        System.out.println();

        // Matriz de cadenas de caracteres
        String[] s = {"ccc","bbb","aaa","eee","ddd"};

```

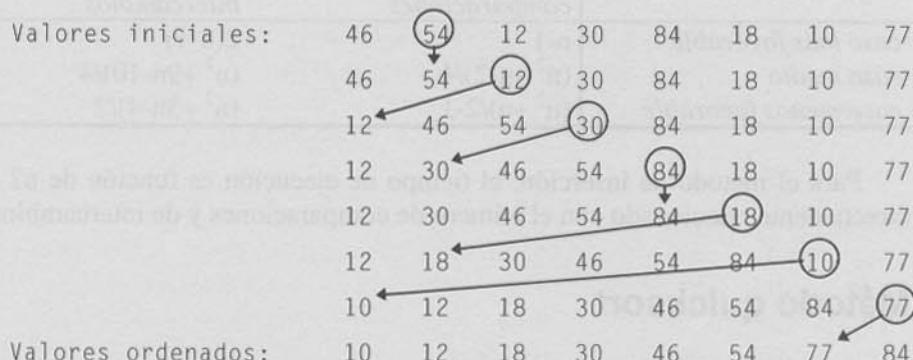
```

    CMatriz.ordenar(s);
    for (int i = 0; i < s.length; i++)
        System.out.print(s[i] + " ");
    System.out.println();
}
}

```

Método de inserción

El algoritmo para este método de ordenación es el siguiente: inicialmente, se ordenan los dos primeros elementos de la matriz, luego se inserta el tercer elemento en la posición correcta con respecto a los dos primeros, a continuación se inserta el cuarto elemento en la posición correcta con respecto a los tres primeros elementos ya ordenados y así sucesivamente hasta llegar al último elemento de la matriz. Por ejemplo:



El pseudocódigo para este algoritmo puede ser el siguiente:

```

<método inserción(matriz "a" de "n" elementos)>
["a" es un matriz cuyos elementos son  $a_0, a_1, \dots, a_{n-1}$ ]
  i = 1
  DO WHILE ( i < n )
    x = a[i]
    insertar x en la posición correcta entre  $a_0$  y  $a_i$ 
  ENDDO
END <inserción>

```

La programación de este algoritmo, para el caso concreto de ordenar numéricamente una lista de valores, es la siguiente:

```

public static void insercion(double[] m)
{
    int i, k, n_elementos = m.length;
    double x;
}

```

```

// Desde el segundo elemento
for (i = 1; i < n_elementos; i++)
{
    x = m[i];
    k = i-1;
    // Para k=-1, se ha alcanzado el extremo izquierdo.
    while (k >=0 && x < m[k])
    {
        m[k+1] = m[k]; // hacer hueco para insertar
        k--;
    }
    m[k+1] = x; // insertar x en su lugar
}
}

```

Análisis del método de inserción directa:

	<i>comparaciones</i>	<i>intercambios</i>
<i>caso más favorable</i>	$n-1$	$2(n-1)$
<i>caso medio</i>	$(n^2 + n - 2)/4$	$(n^2 + 9n - 10)/4$
<i>caso menos favorable</i>	$(n^2 + n)/2 - 1$	$(n^2 + 3n - 4)/2$

Para el método de inserción, el tiempo de ejecución es función de n^2 y está directamente relacionado con el número de comparaciones y de intercambios.

Método quicksort

El método de ordenación *quicksort*, está generalmente considerado como el mejor algoritmo de ordenación disponible actualmente. El proceso seguido por este algoritmo es el siguiente:

1. Se selecciona un valor perteneciente al rango de valores de la matriz. Este valor se puede escoger aleatoriamente o haciendo la media de un pequeño conjunto de valores tomados de la matriz. El valor óptimo sería la mediana (el valor que es menor o igual que los valores correspondientes a la mitad de los elementos de la matriz y mayor o igual que los valores correspondientes a la otra mitad). No obstante, incluso en el peor de los casos (el valor escogido está en un extremo), *quicksort* funciona correctamente.
2. Se divide la matriz en dos partes: una con todos los elementos menores que el valor seleccionado y otra con todos los elementos mayores o iguales.
3. Se repiten los puntos 1 y 2 para cada una de las partes en la que se ha dividido la matriz, hasta que esté ordenada.

El proceso descrito es esencialmente recursivo. Según lo expuesto, el seudocódigo para este algoritmo puede ser el siguiente:

```
<método qs(matriz "a")>
Se elige un valor x de la matriz
DO WHILE ( "a" no esté dividido en dos partes )
  [dividir "a" en dos partes: a_inf y a_sup]
  a_inf con los elementos a_i < x
  a_sup con los elementos a_i >= x
ENDDO
IF ( existe a_inf ) THEN
  qs( a_inf )
ENDIF
IF ( existe a_sup ) THEN
  qs( a_sup )
ENDIF
END <qs>
```

A continuación se muestra una versión de este algoritmo, que selecciona el elemento medio de la matriz para proceder a dividirla en dos partes. Esto resulta fácil de implementar, aunque no siempre da lugar a una buena elección. A pesar de ello, funciona correctamente.

```
public static void quicksort(String[] m)
{
    qs(m, 0, m.length - 1);
}

// Método recursivo qs
private static void qs(String[] m, int inf, int sup)
{
    int izq, der;
    String mitad, x;
    izq = inf; der = sup;
    mitad = m[(izq + der) / 2];
    do
    {
        while (m[izq].compareTo(mitad) < 0 && izq < sup) izq++;
        while (mitad.compareTo(m[der]) < 0 && der > inf) der--;
        if (izq <= der)
        {
            x = m[izq]; m[izq] = m[der]; m[der] = x;
            izq++; der--;
        }
    }
    while (izq <= der);
    if (inf < der) qs(m, inf, der);
    if (izq < sup) qs(m, izq, sup);
}
```

Observamos que cuando el valor *mitad* se corresponde con uno de los valores de la lista, las condiciones *izq < sup* y *der > inf* de las sentencias

```
while (m[izq].compareTo(mitad) < 0 && izq < sup) izq++;
while (mitad.compareTo(m[der]) < 0 && der > inf) der--;
```

no serían necesarias. En cambio, si el valor *mitad* es un valor no coincidente con un elemento de la lista, pero que está dentro del rango de valores al que pertenecen los elementos de la misma, esas condiciones son necesarias para evitar que se puedan sobrepasar los límites de los índices de la matriz.

Para experimentarlo, pruebe como ejemplo la lista de valores *1 1 3 1 1* y elija *mitad = 2* fijo. En este caso las sentencias anteriores serían así:

```
while (m[izq] < mitad && izq < sup) izq++;
while (mitad < m[der] && der > inf) der--;
```

En el método *quicksort*, en el caso más favorable, esto es, cada vez se selecciona la mediana obteniéndose dos particiones iguales, se realizan $n \log n$ comparaciones y $n/6 \log n$ intercambios, donde n es el número de elementos a ordenar; en el caso medio, el rendimiento es inferior al caso óptimo en un factor de $2 \log 2$; y en el caso menos favorable, esto es, cada vez se selecciona el valor mayor obteniéndose una partición de $n-1$ elementos y otra de un elemento, el rendimiento es del orden de $n.n=n^2$. Con el fin de mejorar el caso menos favorable, se sugiere elegir, cada vez, un valor aleatoriamente o un valor que sea la mediana de un pequeño conjunto de valores tomados de la matriz.

El método *qs* sin utilizar la recursión puede desarrollarse de la forma siguiente:

```
public static void quicksortNR(String[] m)
{
    qsNR(m, 0, m.length - 1);
}

// Método no recursivo qs
private static void qsNR(String[] m, int inf, int sup)
{
    CPila pila = new CPila(); // pila de elementos (inf,sup)
    CDatos dato;           // encapsula los atributos inf y sup

    int izq, der, p;
    String mitad, x;
    // Iniciar la pila con los valores: inf, sup
    pila.meterEnPila(new CDatos(inf, sup));
    do
    {
```

```

// Tomar los datos inf, sup de la parte superior de la pila
dato = (CDatos)pila.sacarDePila();
inf = dato.obtenerInf(); sup = dato.obtenerSup();
do
{
    // División de la matriz en dos partes
    izq = inf; der = sup;
    mitad = m[(izq + der) / 2];
    do
    {
        while (m[izq].compareTo(mitad) < 0 && izq < sup) izq++;
        while (mitad.compareTo(m[der]) < 0 && der > inf) der--;
        if (izq <= der)
        {
            x = m[izq]; m[izq] = m[der]; m[der] = x;
            izq++; der--;
        }
    }
    while (izq <= der);
    if (izq < sup)
    {
        // Meter en la pila los valores: izq, sup
        pila.meterEnPila(new CDatos(izq, sup));
    }
    /* inf = inf; */ sup = der;
}
while (inf < der);
}
while (pila.tamaño() != 0);
}

```

En esta solución observamos que después de cada paso se generan dos nuevas sublistas. Una de ellas la tratamos en la siguiente iteración y la otra la posponemos, guardando sus límites *inf* y *sup* en una pila. El método *qsNR* utiliza la clase *CPila*, implementada en el capítulo de “Estructuras dinámicas”, para crear una pila que almacene los valores *inf* y *sup* a los que nos hemos referido anteriormente. Estos límites son los atributos de objetos de una clase *CDatos* definida así:

```

public class CDatos
{
    // Atributos
    private int inf, sup;
    // Métodos
    public CDatos(int i, int s) // constructor con parámetros
    {
        inf = i;
        sup = s;
    }
}

```

```

public int obtenerInf()
{
    return inf;
}

public int obtenerSup()
{
    return sup;
}
}

```

Comparación de los métodos expuestos

Si medimos los tiempos consumidos por los métodos de ordenación estudiados anteriormente, observaremos que el método de la burbuja es el peor de los métodos; el método de inserción directa mejora considerablemente y el método *quicksort* es el más rápido y mejor método de ordenación de matrices con diferencia.

BÚSQUEDA DE DATOS

El objetivo de ordenar un conjunto de objetos es, generalmente, facilitar la búsqueda de uno o más elementos pertenecientes a un conjunto, aunque es posible realizar dicha búsqueda sin que el conjunto de objetos esté ordenado, pero esto trae como consecuencia un mayor tiempo de proceso.

Búsqueda secuencial

Este método de búsqueda, aunque válido, es el menos eficiente. Se basa en comparar el valor que se desea buscar con cada uno de los valores de la matriz. La matriz no tiene por qué estar ordenada.

El pseudocódigo para este método de búsqueda puede ser el siguiente:

```

<método búsqueda_S( matriz a, valor que queremos buscar)>
    i = 0
    DO WHILE ( no encontrado )
        IF ( valor = a[i] )
            encontrado
        ENDIF
        i = i+1
    ENDDO
END <búsqueda_S>

```

Como ejercicio, escribir el código correspondiente a un método que permita buscar un valor previamente leído, en un matriz.

Búsqueda binaria

Un método eficiente de búsqueda, que puede aplicarse a las matrices clasificadas, es la *búsqueda binaria*. Si partimos de que los elementos de la matriz están almacenados en orden ascendente, el proceso de búsqueda binaria puede describirse así: se selecciona el elemento del centro o aproximadamente del centro de la matriz. Si el valor a buscar no coincide con el elemento seleccionado y es mayor que él, se continúa la búsqueda en la segunda mitad de la matriz. Si, por el contrario, el valor a buscar es menor que el valor del elemento seleccionado, la búsqueda continúa en la primera mitad de la matriz. En ambos casos, se halla de nuevo el elemento central, correspondiente al nuevo intervalo de búsqueda, repitiéndose el ciclo. El proceso se repite hasta que se encuentra el valor a buscar, o bien hasta que el intervalo de búsqueda sea nulo, lo que querrá decir que el elemento buscado no figura en la matriz.

El seudocódigo para este algoritmo puede ser el siguiente:

```
<método búsquedaBin( matriz a, valor que queremos buscar )>
DO WHILE ( existe un intervalo donde buscar )
    x = elemento mitad del intervalo de búsqueda
    IF ( valor > x ) THEN
        buscar "valor" en la segunda mitad del intervalo de búsqueda
    ELSE
        buscar "valor" en la primera mitad del intervalo de búsqueda
    ENDIF
ENDDO
IF ( se encontró valor ) THEN
    retornar su índice
ELSE
    retornar -1
ENDIF
END <búsquedaBin>
```

A continuación se muestra el código correspondiente a este método.

```
public static int búsquedaBin(double[] m, double v)
{
    // El método búsquedaBin devuelve como resultado la posición
    // del valor. Si el valor no se localiza devuelve -1.

    if (m.length == 0) return -1;
    int mitad, inf = 0, sup = m.length - 1;
```

```

do
{
    mitad = (inf + sup) / 2;
    if (v > m[mitad])
        inf = mitad + 1;
    else
        sup = mitad - 1;
}
while (m[mitad] != v && inf <= sup);

if (m[mitad] == v)
    return mitad;
else
    return -1;
}

```

Búsqueda de cadenas

Uno de los métodos más eficientes en la búsqueda de cadenas dentro de un texto es el algoritmo *Boyer y Moore*. La implementación básica de este método construye una tabla *delta* que se utilizará en la toma de decisiones durante la búsqueda de una subcadena. Dicha tabla contiene un número de entradas igual al número de caracteres del código que se esté utilizando. Por ejemplo, si se está utilizando el código de caracteres ASCII la tabla será de 256 entradas. Cada entrada contiene el valor *delta* asociado con el carácter que representa. Por ejemplo, el valor *delta* asociado con *A* estará en la entrada 65 y el valor *delta* asociado con el *espacio en blanco*, en la entrada 32. El valor *delta* para un carácter, es la posición de la ocurrencia más a la derecha de ese carácter respecto a la posición final en la cadena buscada. Las entradas correspondientes a los caracteres que no pertenecen a la cadena a buscar, tienen un valor igual a la longitud de esta cadena.

Por lo tanto, para definir la tabla *delta* para una determinada subcadena a buscar, construimos una matriz con todos sus elementos iniciados a la longitud de dicha cadena, y luego, asignamos el valor *delta* para cada carácter de la subcadena, así:

```

for (i = 0; i < longitud_cadena_patrón; i++)
    delta[cadena_patrón[i]] = longitud_cadena_patrón - i - 1;

```

En el algoritmo de *Boyer y Moore* la comparación se realiza de derecha a izquierda, empezando desde el principio del texto. Es decir, se empieza comparando el último carácter de la cadena que se busca con el correspondiente carácter en el texto donde se busca; si los caracteres no coinciden, la cadena que se busca se desplaza hacia la derecha un número de caracteres igual al valor indicado por la entrada en la tabla *delta* correspondiente al carácter del *texto* que no coincide. Si

el carácter no aparece en la cadena que se busca, su valor *delta* es la longitud de la cadena que se busca.

Veamos un ejemplo. Suponga que se desea buscar la cadena "cien" en el texto "Más vale un ya que cien después se hará". La búsqueda comienza así:

Texto: Más vale un ya que cien después se hará
 Cadena a buscar: cien

El funcionamiento del algoritmo puede comprenderse mejor situando la cadena a buscar paralela al texto. La comparación es de derecha a izquierda; por lo tanto, se compara el último carácter en la cadena a buscar (*n*) con el carácter que está justamente encima en el texto (*espacio*). Como *n* es distinto de *espacio*, la cadena que se busca debe desplazarse a la derecha un número de caracteres igual al valor indicado por la entrada en la tabla *delta* que corresponde al carácter del *texto* que no coincide. Para la cadena "cien",

```
delta['c'] = 3
delta['i'] = 2
delta['e'] = 1
delta['n'] = 0
```

El resto de las entradas valen 4 (longitud de la cadena). Según esto, la cadena que se busca se desplaza cuatro posiciones a la derecha (el espacio en blanco no aparece en la cadena que se busca).

Texto: Más vale un ya que cien después se hará
 Cadena a buscar: cien

Ahora, *n* no coincide con *e*; luego la cadena se desplaza una posición a la derecha (*e* tiene un valor asociado de *uno*).

Texto: Más vale un ya que cien después se hará
 Cadena a buscar: cien

n no coincide con *espacio*; se desplaza la cadena cuatro posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
 Cadena a buscar: cien

n no coincide con *y*; se desplaza la cadena cuatro posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
 Cadena a buscar: cien

n no coincide con *u*; se desplaza la cadena cuatro posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
 Cadena a buscar: cien

n no coincide con *i*; se desplaza la cadena dos posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
 Cadena a buscar: cien

Todos los caracteres de la cadena coinciden con los correspondientes caracteres en el texto. Para encontrar la cadena se han necesitado sólo $7+3$ comparaciones (7 hasta que se dio la coincidencia del carácter *c*; más 3 para verificar que coincidían los 3 caracteres restantes). El algoritmo directo habría realizado $20+3$ comparaciones, que en el peor de los casos, serían $i * longCadBuscar$, donde *i* es la posición más a la izquierda de la primera ocurrencia de la cadena a buscar en el texto (20 en el ejemplo anterior, suponiendo que la primera posición es la 1) y *longCadBuscar* es la longitud de la cadena a buscar (4 en el ejemplo anterior). En cambio, el algoritmo *Boyer y Moore* emplearía $k * (i + longCadBuscar)$ comparaciones, donde $k < 1$.

El algoritmo *Boyer y Moore* es más rápido porque tiene información sobre la cadena que se busca, en la tabla *delta*. El carácter que ha causado la no coincidencia en el texto indica cómo mover la cadena respecto del texto. Si el carácter no coincidente en el texto no existe en la cadena, ésta puede moverse sin problemas a la derecha un número de caracteres igual a su longitud, pues es un gasto de tiempo comparar la cadena con un carácter que ella no contiene. Cuando el carácter no coincidente en el texto está presente en la cadena, el valor *delta* para ese carácter alinea la ocurrencia más a la derecha de ese carácter en la cadena, con el carácter en el texto.

A continuación se muestra el código correspondiente al algoritmo *Boyer y Moore*. El método *buscarCadena* es el que realiza el proceso descrito. Este método devuelve la posición de la cadena en el texto o -1 si la cadena no se encuentra (la primera posición es la 0).

```
public static int buscarCadena(String stexto, String scadena)
{
    // Buscar una "cadena" en un "texto"
    char[] texto = stexto.toCharArray();
    char[] cadena = scadena.toCharArray();

    // Construir la tabla "delta"
    int[] delta = new int[256];
    int i, longCad = cadena.length;
```

```

// Iniciar la tabla "delta"
for (i = 0; i < 256; i++)
    delta[i] = longCad;
// Asignar valores a la tabla
for (i = 0; i < longCad; ++i)
    delta[cadena[i]] = longCad - i - 1;

// Algoritmo Boyer-Moore
int j, longTex = texto.length;
i = longCad - 1; // i es el índice dentro del texto
while (i < longTex)
{
    j = longCad - 1; // índice dentro de la cadena a buscar
    // Mientras haya coincidencia de caracteres
    while (cadena[j] == texto[i])
    {
        if (j > 0)
        {
            // Siguiente posición a la izquierda
            j--; i--;
        }
        else
        {
            // Se llegó al principio de la cadena, luego se encontró.
            return i;
        }
    }
    // Los caracteres no coinciden. Mover i lo que indique el
    // valor "delta" del carácter del texto que no coincide
    i += delta[texto[i]];
}
return -1;
}

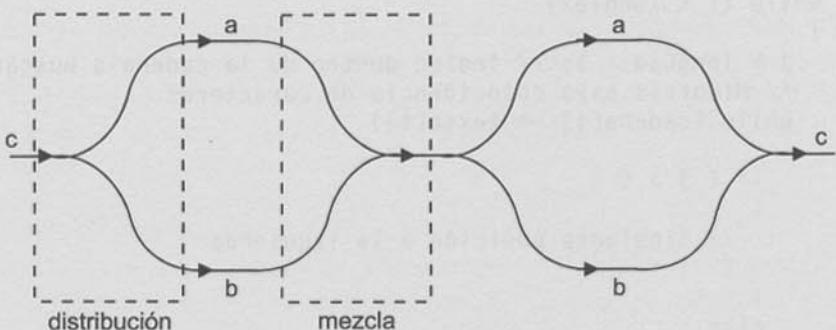
```

ORDENACIÓN DE FICHEROS EN DISCO

Para ordenar un fichero, dependiendo del tamaño del mismo, podremos proceder de alguna de las dos formas siguientes. Si el fichero es pequeño, tiene pocos registros, se puede copiar en memoria en una matriz y utilizando las técnicas vistas anteriormente, ordenamos dicha matriz y a continuación copiamos la matriz ordenada de nuevo en el fichero. Sin embargo, muchos ficheros son demasiado grandes y no cabrían en una matriz en memoria, por lo que para ordenarlos recurriremos a técnicas que actúen sobre el propio fichero.

Ordenación de ficheros. Acceso secuencial

El siguiente programa desarrolla un algoritmo de ordenación de un fichero utilizando el acceso secuencial, denominado *mezcla natural*. La secuencia inicial de los elementos viene dada en el fichero *c* y se utilizan dos ficheros auxiliares denominados *a* y *b*. Cada pasada consiste en una *fase de distribución* que reparte equitativamente los tramos ordenados del fichero *c* sobre los ficheros *a* y *b*, y una *fase que mezcla* los tramos de los ficheros *a* y *b* sobre el fichero *c*.



Este proceso se ilustra en el ejemplo siguiente. Partimos de un fichero *c*. Con el fin de ilustrar el método de *mezcla natural*, separaremos los tramos ordenados en los ficheros por un guión (-).

fichero c: 18 32 - 10 60 - 14 42 44 68 - 12 24 30 48

Fase de distribución:

fichero a: 18 32 - 14 42 44 68
fichero b: 10 60 - 12 24 30 48

Fase de mezcla:

fichero c: 10 18 32 60 - 12 14 24 30 42 44 48 68

Fase de distribución:

fichero a: 10 18 32 60
fichero b: 12 14 24 30 42 44 48 68

Fase de mezcla:

fichero c: 10 12 14 18 24 30 32 42 44 48 60 68

Para dejar ordenado el fichero del ejemplo hemos necesitado realizar dos pasadas. El proceso finaliza, tan pronto como el número de tramos ordenados del fichero *c*, sea 1. Una forma de reducir el número de pasadas es distribuir los tramos ordenados sobre más de dos ficheros.

Según lo expuesto, el algoritmo de ordenación *mezcla natural* podría ser así:

```
<método mezcla_natural()>
n_tramos = 0;
DO
    [Crear y abrir los ficheros temporales a y b]
    n_tramos = distribución();
    n_tramos = mezcla();
    WHILE (n_tramos != 1);
END <mezcla_natural()>
```

La estructura de la aplicación que permite ordenar un fichero utilizando el algoritmo descrito puede ser de la forma siguiente:

```
public class CMezclaNatural
{
    public static void mezclaNatural(File fichFuente)
        throws IOException
    {
        int nro_tramos = 0;
        // a y b son dos ficheros temporales

        do
        {
            distribuir(fichFuente, a, b);
            nro_tramos = mezclar(a, b, fichFuente);
        }
        while (nro_tramos != 1);
    }

    public static int distribuir(File fuente, File destinoA,
                                File destinoB) throws IOException
    {
        // Distribuir los tramos ordenados de fuente entre
        // destinoA y destinoB
    }

    public static int mezclar(File fuenteA, File fuenteB,
                            File destino) throws IOException
    {
        // Fusionar ordenadamente los tramos de destinoA y destinoB
        // en fuente
    }
}
```

```

public static void main(String[] args)
{
    File nombreFichero = new File(args[0]);
    mezclaNatural(nombreFichero);
}
}

```

La aplicación completa y comentada se muestra a continuación.

```

import java.io.*;

///////////////////////////////
// Ordenar un fichero utilizando el método de mezcla natural.
// Se trata de un fichero de texto que almacena una lista de
// nombres.
// El nombre del fichero se recibe a través de la línea de órdenes.
// La ordenación se realiza en orden alfabético ascendente.
// La aplicación está soportada por la clase CMezclaNatural.
// Métodos:
//     mezclaNatural
//     distribuir
//     mezclar
//     main
//
public class CMezclaNatural
{
    // Mezcla natural ///////////////////////////////
    public static void mezclaNatural(File fichFuente)
        throws IOException
    {
        // Definición de variables
        File a = new File("ftempa.tmp"); // fichero temporal
        File b = new File("ftempb.tmp"); // fichero temporal

        int nro_tramos;
        do
        {
            nro_tramos = distribuir(fichFuente, a, b);

            if (nro_tramos <= 1)
            {
                // Proceso finalizado. Borrar los ficheros temporales.
                a.delete(); b.delete();
                return;
            }
            nro_tramos = mezclar(a, b, fichFuente);
        }
        while (nro_tramos != 1);
    } // mezclaNatural
}

```

```
// Fase de distribución /////////////////////////////////
public static int distribuir(File fuente, File destinoA,
                               File destinoB) throws IOException
{
    // Abrir un flujo de entrada desde fuente que permita
    // leer la información línea a línea.
    FileInputStream fis = new FileInputStream(fuente);
    InputStreamReader isr = new InputStreamReader(fis);
    BufferedReader fc = new BufferedReader(isr);

    // Abrir un flujo de salida hacia destinoA
    FileOutputStream fosA = new FileOutputStream(destinoA);
    OutputStreamWriter osrA = new OutputStreamWriter(fosA);
    BufferedWriter fa = new BufferedWriter(osrA);

    // Abrir un flujo de salida hacia destinoB
    FileOutputStream fosB = new FileOutputStream(destinoB);
    OutputStreamWriter osrB = new OutputStreamWriter(fosB);
    BufferedWriter fb = new BufferedWriter(osrB);

    BufferedWriter faux = fa; // faux será fa o fb
    String linea;           // última linea leída
    String linea_ant;       // linea anterior a la última leída
    int nro_tramos = 1;     // número total de tramos ordenados

    // Leer la primera linea (linea anterior)
    if ((linea_ant = fc.readLine()) != null)
    {
        // Escribe en fa la linea leída más el separador de linea
        fa.write(linea_ant); fa.newLine();
    }
    else
    {
        faux = null; fc.close(); fa.close(); fb.close();
        return 0;
    }

    // Leer la siguiente linea (linea actual)
    while ((linea = fc.readLine()) != null)
    {
        if (linea.compareTo(linea_ant) < 0)
        {
            // Cambiar al otro fichero
            faux = (faux == fa) ? fb : fa;
            ++nro_tramos;
        }
        linea_ant = linea;
        // Escribe en faux la linea leída más el separador de linea
        faux.write(linea); faux.newLine();
    }
}
```

```
faux = null; fc.close(); fa.close(); fb.close();
return nro_tramos;
} // distribuir

// Fase de mezcla /////////////////////////////////
public static int mezclar(File fuenteA, File fuenteB,
                           File destino) throws IOException
{
    // Abrir un flujo de entrada desde fuenteA que permita
    // leer la información línea a línea.
    FileInputStream fisA = new FileInputStream(fuenteA);
    InputStreamReader isrA = new InputStreamReader(fisA);
    BufferedReader fa = new BufferedReader(isrA);

    // Abrir un flujo de entrada desde fuenteB que permita
    // leer la información línea a línea.
    FileInputStream fisB = new FileInputStream(fuenteB);
    InputStreamReader isrB = new InputStreamReader(fisB);
    BufferedReader fb = new BufferedReader(isrB);

    // Abrir un flujo de salida hacia destino
    FileOutputStream fos = new FileOutputStream(destino);
    OutputStreamWriter osr = new OutputStreamWriter(fos);
    BufferedWriter fc = new BufferedWriter(osr);

    String lineaDeFa, lineaDeFb, lineaDeFa_ant, lineaDeFb_ant;
    int nro_tramos = 1;

    // Leemos las dos primeras líneas, una de fa y otra de fb
    lineaDeFa = fa.readLine();
    lineaDeFa_ant = lineaDeFa;
    lineaDeFb = fb.readLine();
    lineaDeFb_ant = lineaDeFb;

    // Vamos leyendo y comparando hasta que se acabe alguno de los
    // ficheros. La fusión se realiza entre pares de tramos
    // ordenados. Un tramo de fa y otro de fb darán lugar a un
    // tramo ordenado sobre fc.
    while (lineaDeFa != null && lineaDeFb != null)
    {
        if (lineaDeFa.compareTo(lineaDeFb) < 0)           // if 1
        {
            if (lineaDeFa.compareTo(lineaDeFa_ant) < 0) // if 2
                // Encontrado el final del tramo de fa
            {
                lineaDeFa_ant = lineaDeFa;
                // Copiamos el tramo ordenado de fb
                do
                {
                    fc.write(lineaDeFb); fc.newLine();
                    lineaDeFb = fb.readLine();
                }
                while (lineaDeFb.compareTo(lineaDeFa_ant) <= 0);
                lineaDeFa_ant = lineaDeFb;
            }
        }
    }
}
```

```
    lineaDeFb_ant = lineaDeFb;
}
while ((lineaDeFb = fb.readLine()) != null &&
       lineaDeFb.compareTo(lineaDeFb_ant) > 0):
    ++nro_tramos;
}
else // de if 2
{
    // Copiamos la cadena leída de fa
    lineaDeFa_ant = lineaDeFa;
    fc.write(lineaDeFa); fc.newLine();
    lineaDeFa = fa.readLine();
}
}
else // de if 1
{
    if (lineaDeFb.compareTo(lineaDeFb_ant) < 0) // if 3
        // Encontrado el final del tramo de fb
    {
        lineaDeFb_ant = lineaDeFb;
        // Copiamos el tramo ordenado de fa
        do
        {
            fc.write(lineaDeFa); fc.newLine();
            lineaDeFa_ant = lineaDeFa;
        }
        while ((lineaDeFa = fa.readLine()) != null &&
               lineaDeFa.compareTo(lineaDeFa_ant) > 0);
        ++nro_tramos;
    }
    else // de if 3
    {
        // Copiamos la cadena leída de fb
        lineaDeFb_ant = lineaDeFb;
        fc.write(lineaDeFb); fc.newLine();
        lineaDeFb = fb.readLine();
    }
}
}
} // de while

// En el caso de acabarse primero los datos de fb
if (lineaDeFb == null)
{
    fc.write(lineaDeFa); fc.newLine();
    while ((lineaDeFa = fa.readLine()) != null)
    {
        fc.write(lineaDeFa); fc.newLine();
    }
}
// En el caso de acabarse primero los datos de fa
```

```

else if (líneaDeFa == null)
{
    fc.write(líneaDeFb); fc.newLine();
    while ((líneaDeFb = fb.readLine()) != null)
    {
        fc.write(líneaDeFb); fc.newLine();
    }
}
fc.close(); fa.close(); fb.close();
return nro_tramos;
} // de mezclar

public static void main(String[] args)
{
    // main debe recibir un parámetro: el fichero a ordenar.
    if (args.length != 1)
        System.err.println("Sintaxis: java CMezclaNatural " +
                           "<nombre_fichero>");
    else
    {
        File nombreFichero = new File(args[0]);
        try
        {
            // Asegurarse de que "nombreFichero" existe y se puede leer
            if (!nombreFichero.exists() || !nombreFichero.isFile())
                throw new IOException("No existe el fichero " +
                                      nombreFichero);
            if (!nombreFichero.canRead())
                throw new IOException("El fichero " + nombreFichero +
                                      " no se puede leer");
            mezclaNatural(nombreFichero); // realizar la ordenación
            // Mostrar el contenido del fichero
            char resp;
            System.out.print("¿Desea ver el contenido del fichero? s/n: ");
            resp = Leer.carácter(); Leer.limpiar();
            if (resp == 's')
            {
                // Abrir un flujo de entrada desde nombreFichero
                // que permita leer la información línea a línea.
                FileInputStream fis = new FileInputStream(nombreFichero);
                InputStreamReader isr = new InputStreamReader(fis);
                BufferedReader fc = new BufferedReader(isr);

                // Leer el fichero y mostrarlo
                String linea;
                while ((linea = fc.readLine()) != null)
                    System.out.println(linea);
                fc.close();
            }
        }
    }
}

```

```

        catch(IOException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
///////////////////////////////////////////////////////////////////

```

El flujo **BufferedReader** (**BufferedInputStream**) cuando tratemos con bytes en lugar de con caracteres; ver el capítulo 5) es uno de los más eficientes. Se deriva de **Reader** y añade un *buffer* que actúa como una memoria intermedia desde la que el programa obtendrá los datos. Como su tamaño, generalmente, es bastante más grande que el tamaño del bloque físico asociado con el dispositivo desde el cual se obtiene la información, el número de accesos a este dispositivo por parte de la aplicación disminuirá, ya que el *buffer* que se llenó cuando la aplicación realizó una operación de lectura, no necesitará volverse a llenar mientras la aplicación no procese los datos almacenados en el mismo. Resumiendo, la utilización de un *buffer* repercute en una mejora de la velocidad de ejecución y además, desacopla el tamaño de las unidades de información que requiere el programa, del tamaño de las unidades de información asociadas con el dispositivo.

De la clase **BufferedReader**, cabe destacar los métodos: **readLine** que permite leer una línea de texto sin almacenar el carácter delimitador de la misma (normalmente \n, \r, o \r\n), **mark** que permite poner una marca en la posición actual del flujo y **reset** que permite reanudar la lectura desde la marca más reciente.

Análogamente **BufferedWriter** (**BufferedOutputStream**) también añade un *buffer* que no será volcado al dispositivo asociado con el flujo hasta que no esté lleno, disminuyendo el número de accesos al dispositivo, lo que repercute en una mejora de la velocidad de ejecución, además de desacoplar el tamaño de las unidades de información que requiere el programa, del tamaño de las unidades de información asociadas con el dispositivo.

De la clase **BufferedWriter**, cabe destacar los métodos: **write** que permite escribir una línea de texto pero sin escribir el carácter delimitador de la misma (normalmente \n, \r, o \r\n) y **newLine** que permite escribir el carácter delimitador de línea; este carácter es definido por el sistema y no tiene que ser necesariamente el carácter \n.

Ordenación de ficheros. Acceso aleatorio

El acceso aleatorio a un fichero, a diferencia del secuencial, permite ordenar la información contenida en el mismo sin tener que copiarla sobre otro fichero, para lo cual aplicaremos un proceso análogo al aplicado a las matrices, lo que simplifica enormemente el proceso ordenación. Esto quiere decir que los métodos expuestos para ordenar matrices, pueden ser aplicados también para ordenar ficheros utilizando el acceso aleatorio.

Como ejemplo, vamos a añadir á la clase *CListaTfnos* de la aplicación realizada en el apartado “Un ejemplo de acceso aleatorio a un fichero” del capítulo 12, un método denominado *quicksort* para ordenar el fichero “lista de teléfonos” encapsulado por la misma, en el que cada registro estaba formado por los campos: *nombre*, *dirección* y *teléfono*. La ordenación del fichero la realizaremos por el campo *nombre*, de tipo alfabético, empleando el método *quicksort* explicado anteriormente en este mismo capítulo.

```
///////////////////////////////
// Definición de la clase CListaTfnos.
//
import java.io.*;
public class CListaTfnos
{
    private RandomAccessFile fes; // flujo
    private int nregs;           // número de registros
    private int tamañoReg = 140; // tamaño del registro en bytes

    public CListaTfnos(File fichero) throws IOException
    {
        // ...
    }

    public void cerrar() throws IOException { fes.close(); }

    public int longitud() { return nregs; } // número de registros

    public boolean ponerValorEn( int i, CPersona objeto )
        throws IOException
    {
        // Escribir objeto en el registro de la posición i
    }

    public CPersona valorEn( int i ) throws IOException
    {
        // Obtener los datos del registro i
    }
}
```

```

public void añadir(CPersona obj) throws IOException
{
    // Añadir un registro al final del fichero
}

public boolean eliminar(long tel) throws IOException
{
    // Eliminar el registro especificado por tel
}

public int buscar(String str, int pos) throws IOException
{
    // Buscar un determinado registro a partir de pos
}

// Método quicksort para ordenar el fichero ///////////////////////////////
public void quicksort() throws IOException
{
    qs(0, nregs - 1);
}

private void qs(int inf, int sup) throws IOException
{
    int izq = inf, der = sup;
    String mitad;

    // Obtener del registro mitad, el campo por el que
    // se va a ordenar el fichero.
    mitad = campo((izq + der)/2);
    do
    {
        while (campo(izq).compareTo(mitad) < 0 && izq < sup) izq++;
        while (mitad.compareTo(campo(der)) < 0 && der > inf) der--;
        if (izq <= der)
        {
            permutarRegistros(izq, der);
            izq++; der--;
        }
    }
    while (izq <= der);
    if (inf < der) qs(inf, der);
    if (izq < sup) qs(izq, sup);
}

// Permutar los registros de las posiciones i y j
private void permutarRegistros(int i, int j) throws IOException
{
    CPersona x, y;
    // Leer los registros de las posiciones i y j
    x = valorEn(i);
}

```

```

y = valorEn(j);
// Escribirlos en las posiciones j e i
ponerValorEn(j, x);
ponerValorEn(i, y);
}

// Obtener el campo utilizado para ordenar, del registro nreg
private String campo(int nreg) throws IOException
{
    fes.seek(nreg * tamañoReg); // situar el puntero de L/E
    return fes.readUTF();       // devuelve el nombre
}

```

El método *quicksort* realiza la ordenación de los *nregs* registros del fichero vinculado con el flujo *fes*. Para ello invoca al método recursivo *qs*.

El método *permutarRegistros* es llamado por *qs (quicksort)* cuando hay que permutar dos registros del fichero para que queden correctamente ordenados.

El método *campo* es llamado por *qs (quicksort)* cada vez que es necesario obtener el campo *nombre*, utilizado para realizar la ordenación, de un registro.

Como ejercicio, puede añadir al menú presentado por la aplicación *Test* a la que nos hemos referido anteriormente, que creaba una lista de teléfonos a partir de la clase *CListaTfnos*, una opción más, *ordenar*, que permita ordenar el fichero:

```
File fichero = new File("listatfnos.dat");
listatfnos = new CListaTfnos(fichero);
// ...
case 6: // ordenar
    listatfnos.quicksort();
    break;
```

ALGORITMOS HASH

Los algoritmos *hash* son métodos de búsqueda, que proporcionan una longitud de búsqueda pequeña y una flexibilidad superior a la de otros métodos, como puede ser, el método de *búsqueda binaria* que requiere que los elementos de la matriz estén ordenados.

Por *longitud de búsqueda* se entiende el número de accesos que es necesario efectuar sobre una matriz para encontrar el elemento deseado.

Este método de búsqueda permite, como operaciones básicas, además de la búsqueda de un elemento, insertar un nuevo elemento y eliminar un elemento existente.

Matrices hash

Una matriz producto de la aplicación de un algoritmo *hash* se denomina *matriz hash* y son las matrices que se utilizan con mayor frecuencia en los procesos donde se requiere un acceso rápido a los datos. Gráficamente estas matrices tienen la siguiente forma:

Clave	Contenido
5040	
3721	
...	
6375	

La matriz se organiza con elementos formados por dos miembros: *clave* y *contenido*. La *clave* constituye el medio de acceso a la matriz. Aplicando a la *clave* una función de acceso *fa*, previamente definida, obtenemos un número entero positivo *i* correspondiente a la posición del elemento en la matriz.

$$i = fa(\text{clave})$$

Conociendo la posición, tenemos acceso al *contenido*. El miembro *contenido* puede albergar directamente la información, o bien una referencia a dicha información, cuando ésta sea muy extensa. El acceso, tal cual lo hemos definido, recibe el nombre de *acceso directo*.

Como ejemplo, suponer que la *clave* de acceso se corresponde con el número del documento nacional de identidad (*dni*) y que el contenido son los datos correspondientes a la persona que tiene ese *dni*. Una función de acceso, $i = fa(dni)$, que haga corresponder la posición del elemento en la matriz con el *dni*, es inmediata: $i = dni$. Esta función así definida presenta un inconveniente y es que el número de valores posibles de *i* es demasiado grande para utilizar una matriz. Para solucionar este problema, siempre es posible, dada una matriz de longitud *L*, crear una función de acceso, *fa*, que genere un valor comprendido entre 0 y *L*, más co-

múnmente entre 1 y L . En este caso puede suceder que dos o más claves den lugar a un mismo valor de i :

$$i = fa(clave_1) = fa(clave_2)$$

El método *hash* está basado en esta técnica; el acceso a la matriz es directo a través del número i y cuando se produce una *colisión* (dos claves diferentes dan un mismo número i) este elemento se busca en una zona denominada *área de desbordamiento*.

Método hash abierto

Éste es uno de los métodos más utilizados. El algoritmo para acceder a un elemento de la matriz de longitud L , es el siguiente:

1. Se calcula $i = fa(clave)$.
2. Si la posición i de la matriz está libre, se inserta la *clave* y el *contenido*. Si no está libre y la *clave* es la misma, error: “clave duplicada”. Si no está libre y la clave es diferente, incrementamos i en una unidad y repetimos el proceso descrito en este punto 2. Como ejemplo, vea la tabla siguiente:

Clave	Contenido
5040	0
3721	1
	2
6883	3
4007	4
3900	5
	6
6375	7

En la figura, se observa que se quiere insertar la clave 6383. Supongamos que aplicando la función de acceso, obtenemos un valor 3; esto es:

$$i = fa(6383) = 3$$

Como la posición 3 está ocupada y la clave es diferente, tenemos que incrementar i y volver de nuevo al punto 2 del algoritmo.

La *longitud media de búsqueda* en una *matriz hash abierta* viene dada por la expresión:

$$\text{accesos} = (2-k)/(2-2k)$$

siendo k igual al número de elementos existentes en la matriz dividido por L . Por ejemplo, si existen 60 elementos en una matriz de longitud $L=100$, el número medio de accesos para localizar un elemento será:

$$\text{accesos} = (2-60/100)/(2-2*60/100) = 1,75$$

En el método de *búsqueda binaria*, el número de accesos viene dado por el valor $\log_2 N$, siendo N el número de elementos de la matriz.

Para reducir al máximo el número de colisiones y, como consecuencia, obtener una longitud media de búsqueda baja, es importante elegir bien la función de acceso. Una *función de acceso* o *función hash* bastante utilizada y que proporciona una distribución de las claves uniforme y aleatoria es la *función mitad del cuadrado* que dice: "dada una clave C , se eleva al cuadrado (C^2) y se cogen n bits del medio, siendo $2^n \leq L$ ". Por ejemplo, supongamos:

$L = 256$ lo que implica $n = 8$

$C = 625$

$C^2 = 390625 \quad (0 \leq C^2 \leq 2^{32}-1)$

$390625_{10} = 000000000000101111010111100001_2$

n bits del medio: $01011111_2 = 95_{10}$

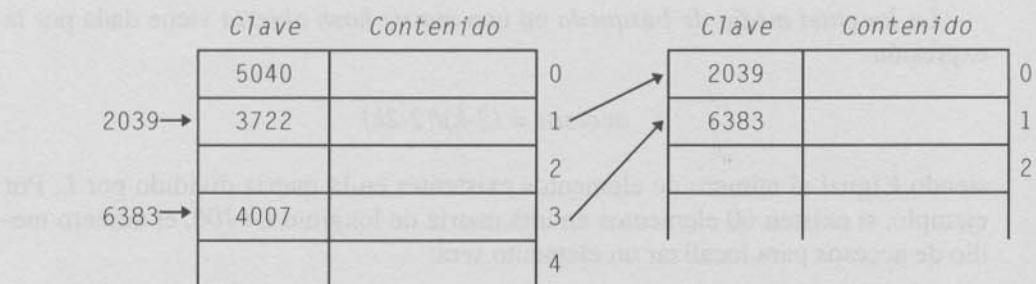
Otra función de acceso muy utilizada es la *función módulo* (resto de una división entera):

$$i = \text{módulo}(\text{clave}/L)$$

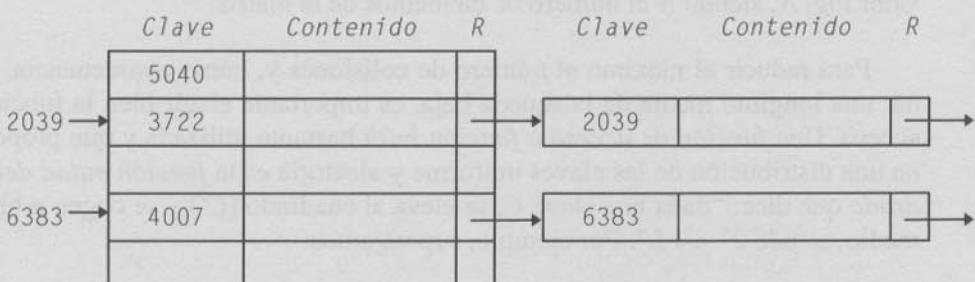
Cuando se utilice esta función es importante elegir un número primo para L , con la finalidad de que el número de colisiones sea pequeño. Esta función es llevada a cabo en Java por medio del operador %.

Método hash con desbordamiento

Una alternativa al método anterior es la de disponer de otra matriz separada, para insertar las claves que producen colisión, denominada *matriz de desbordamiento*, en la que se almacenan todas estas claves de forma consecutiva.



Otra forma alternativa más normal es organizar una lista encadenada por cada posición de la matriz donde se produzca una colisión.



Cada elemento de esta estructura incorpora un nuevo miembro *R*, el cual es una referencia a la lista encadenada de desbordamiento.

Eliminación de elementos

En el método *hash* la eliminación de un elemento no es tan simple como dejar vacío dicho elemento, ya que esto daría lugar a que los elementos insertados por colisión no puedan ser accedidos. Por ello, se suele utilizar un miembro complementario que sirva para poner una marca de que dicho elemento está eliminado. Esto permite acceder a otros elementos que dependen de él por colisiones, ya que la clave se conserva y también permite insertar un nuevo elemento en esa posición cuando se dé una nueva colisión.

Clase CHashAbierto

Como ejercicio escribimos a continuación una clase denominada *CHashAbierto* que proporciona los métodos necesarios para trabajar con matrices *hash* utilizando el método *hash abierto*, cuyo pseudocódigo se expone a continuación:

```

<método hash(matriz, elemento x)>
[La matriz está iniciada a cero]
i = matrizula módulo número_elementos
DO WHILE (no insertado y haya elementos libres)
    IF (elemento "i" está libre) THEN
        copiar elemento x en la posición i
    ELSE
        IF (clave duplicada) THEN
            error: clave duplicada
        ELSE
            [se ha producido una colisión]
            [avanzar al siguiente elemento]
            i = i+1
            IF (i = número_elementos) THEN
                i = 0
            ENDIF
        ENDIF
    ENDIF
ENDDO
END <hash>

```

La clase *CHashAbierto* que vamos a implementar incluirá un atributo privado *matrizhash* para referenciar la *matriz hash*. Un objeto *CHashAbierto* encapsula una *matriz hash* de 101 elementos por omisión, que son referencias a objetos de tipo **Object**, lo que permitirá almacenar elementos de cualquier clase. Asimismo, incluye los métodos indicados en la tabla siguiente:

Método	Significado
<i>CHashAbierto</i>	Es el constructor; está sobrecargado dos veces, una sin parámetros, que crea una matriz de 101 elementos, y otra con un parámetro, que especifica el número de elementos. Inicialmente todas los elementos almacenan el valor null .
<i>númeroDeElementos</i>	Método que devuelve el número de elementos de la matriz.
<i>númeroPrimo</i>	Método que devuelve un número primo a partir de un número pasado como argumento. Como vamos a utilizar la función de acceso <i>módulo</i> es importante elegir un número primo como longitud de la matriz, con la finalidad de que el número de colisiones sea pequeño.
<i>fa</i>	Se trata de un método abstracto con la intención de redefinirlo en una clase derivada y escribir la función de acceso en función de los datos manipulados.
<i>comparar</i>	Método que debe ser redefinido por el usuario en una subclase para permitir comparar las claves de dos objetos de los referenciados por la matriz. Debe de devolver un entero indicando el resultado de la comparación (0 para ==).

<i>hashIn</i>	Método <i>hash abierto</i> para añadir un elemento a la matriz. No devuelve nada.
<i>hashOut</i>	Método <i>hash abierto</i> para buscar un objeto con una clave determinada. Si se encuentra, devuelve una referencia de tipo Object al mismo; en otro caso devuelve null .

A continuación se presenta el código correspondiente a la definición de la clase *CHashAbierto*:

```
///////////////////////////////
// Clase abstracta CHashAbierto: método hash abierto.
// Para utilizar los métodos proporcionados por esta clase,
// tendremos que crear una subclase de ella y redefinir los
// métodos: fa (función de acceso) y comparar.
//
public abstract class CHashAbierto
{
    // Atributos
    private Object[] matrizhash;

    // Métodos
    public CHashAbierto()
    {
        matrizhash = new Object[101];
    }

    public CHashAbierto(int númeroDeElementos)
    {
        if (númeroDeElementos < 1)
            númeroDeElementos = 101;
        else
            númeroDeElementos = númeroPrimo(númeroDeElementos);
        matrizhash = new CALumno[númeroDeElementos];
    }

    public int númeroDeElementos() { return matrizhash.length; }

    // Buscar un número primo a partir de un número dado //////////////////
    public int númeroPrimo(int n)
    {
        boolean primo = false;
        int i, r = (int)Math.sqrt((double)n);

        if (n % 2 == 0) n++;
        while (!primo)
        {
            primo = true;
            for (i = 3; i <= r; i += 2)
                if (n % i == 0) primo = false;
        }
    }
}
```

```
        if (!primo) n += 2; // siguiente impar
    }
    return n;
}

// Función de acceso /////////////////////////////////
// Este método debe ser redefinido en una subclase para poder
// definir la función de acceso que el usuario desee aplicar.
public abstract int fa(Object obj);

// Método comparar /////////////////////////////////
// Este método debe ser redefinido en una subclase para que
// permita comparar dos elementos de la matriz hash por el
// atributo que necesitemos en cada momento.
public abstract int comparar(Object obj1, Object obj2);

// Método hash abierto ///////////////////////////////
public void hashIn(Object x)
{
    int i;           // índice para acceder a un elemento
    int conta = 0; // contador
    boolean insertado = false;

    i = fa(x);      // función de acceso
    while (!insertado && conta < matrizhash.length)
    {
        if (matrizhash[i] == null) // elemento libre
        {
            matrizhash[i] = x;
            insertado = true;
        }
        else // clave duplicada
            if (comparar(x, matrizhash[i]) == 0)
            {
                System.out.println("error: clave duplicada");
                insertado = true;
            }
        else // colisión
        {
            // Siguiente elemento libre
            i++; conta++;
            if (i == matrizhash.length) i = 0;
        }
    }
    if (conta == matrizhash.length)
        System.out.println("error: matriz llena\n");
}

public Object hashOut(Object x)
{
```

```

// x proporcionará el atributo utilizado para buscar. El resto
// de los atributos no interesan (son los que se desea conocer)

int i;           // índice para acceder a un elemento
int conta = 0; // contador
boolean encontrado = false;

i = fa(x);      // función de acceso
if (matrizhash[i] == null) return null;
while (!encontrado && conta < matrizhash.length)
{
    if (comparar(x, matrizhash[i]) == 0)
    {
        x = matrizhash[i];
        encontrado = true;
    }
    else // colisión
    {
        // Siguiente elemento libre
        i++; conta++;
        if (i == matrizhash.length) i = 0;
    }
}
if (conta == matrizhash.length) // no existe
    return null;
else
    return x;
}
}
/////////////////////////////////////////////////////////////////

```

Un ejemplo de una matriz hash

Como ya hemos indicado, para utilizar esta clase tenemos que derivar otra de ella y redefinir los métodos *fa* y *comparar* en función de la información encapsulada por los objetos de datos que deseemos manipular. Por ejemplo, supongamos que deseamos construir una *matriz hash* de objetos de la clase *CAlumno*:

```

/////////////////////////////////////////////////////////////////
// Definición de la clase CAlumno
//
public class CAlumno
{
    // Atributos
    private int matricula;
    private String nombre;

    // Métodos
}

```

```

public CALumno() {}

public CALumno(String nom, int mat)
{
    nombre = nom;
    matrícula = mat;
}

public void asignarNombre(String nom)
{
    nombre = nom;
}

public String obtenerNombre()
{
    return nombre;
}

public void asignarMatrícula(int mat)
{
    matrícula = mat;
}

public long obtenerMatrícula()
{
    return matrícula;
}
}

```

Los objetos *CALumno* serán almacenados en la matriz utilizando como clave el número de matrícula. Según esto, derivamos la clase *HashAbierto* de la clase abstracta *CHashAbierto* y redefinimos los métodos *fa* y *comparar* así:

```

///////////////////////////////
// Clase derivada de la clase abstracta CHashAbierto. Redefine
// los métodos: fa (función de acceso) y comparar.
//
public class HashAbierto extends CHashAbierto
{
    public HashAbierto(int nElementos)
    {
        super(nElementos);
    }

    public int fa(Object obj)
    {
        return (int)((CALumno)obj).obtenerMatrícula() % númeroDeElementos();
    }
}

```

```

public int comparar(Object obj1, Object obj2)
{
    if (((CAAlumno)obj1).obtenerMatricula() ==
        ((CAAlumno)obj2).obtenerMatricula())
        return 0;
    else
        return 1;
}
/////////////////////////////////////////////////////////////////

```

Observe que para definir la función de acceso *módulo* (%) necesitamos utilizar un valor numérico. Esto no quiere decir que la clave tenga que ser numérica, como sucede en nuestro ejemplo, sino que puede ser alfanumérica. Cuando se trabaje con claves alfanuméricas o alfábéticas, por ejemplo *nombre*, antes de aplicar la función de acceso es necesario convertir dicha clave en un valor numérico utilizando un algoritmo adecuado.

Finalmente, escribiremos una aplicación *Test* que permita crear un objeto *HashAbierto*, envoltorio de la *matriz hash*. Para probar su correcto funcionamiento escribiremos código que permita tanto añadir como buscar objetos en dicha matriz.

```

import java.io.*;
/////////////////////////////////////////////////////////////////
// Aplicación para trabajar con una matriz hash
//
public class Test
{
    public static void main(String[] args)
    {
        // Definición de variables
        PrintStream flujoS = System.out;
        int n_elementos; // número de elementos de la matriz hash
        int i;

        String nombre;
        int matricula;
        CAAlumno x;

        // Crear un objeto HashAbierto (encapsula la matriz hash)
        System.out.println("número de elementos: ");
        n_elementos = Leer.datToInt();
        HashAbierto m = new HashAbierto(n_elementos);
        flujoS.println("Se construye una matriz de " +
                      m.númeroDeElementos() + " elementos");

        // Introducir datos

```

```

flujoS.println("Introducir datos. " +
                "Para finalizar, matrícula = 0\n");
flujoS.print("matrícula: "); matrícula = Leer.datoInt();
while (matrícula != 0)
{
    flujoS.print("nombre:      "); nombre = Leer.dato();
    m.hashIn(new CALumno(nombre, matrícula));
    flujoS.print("matrícula:  "); matrícula = Leer.datoInt();
}

// Buscar datos
flujoS.println("Buscar datos. " +
                "Para finalizar, matrícula = 0\n");
flujoS.print("matrícula: "); matrícula = Leer.datoInt();
while (matrícula != 0)
{
    x = (CALumno)m.hashOut(new CALumno("", matrícula));
    if (x != null)
        flujoS.println("nombre: " + x.obtenerNombre());
    else
        flujoS.println("No existe");
    flujoS.print("matrícula:  "); matrícula = Leer.datoInt();
}
}

```

Como alternativa, la biblioteca de Java proporciona en su paquete `java.util` las clases **HashSet**, **HashMap** y **HashTable** que proporcionan conjuntos de datos que no permiten duplicados y que utilizan algoritmos *hash* para el almacenamiento de los datos y para su posterior acceso.

EJERCICIOS RESUELTOS

1. Comparar las dos siguientes versiones del método *búsqueda binaria* e indicar cuál de ellas es más eficaz.

```
public static int búsquedaBin1(double[] m, double v)
{
    // El método búsquedaBin devuelve como resultado la posición
    // del valor. Si el valor no se localiza devuelve -1.

    if (m.length == 0) return -1;
    int mitad, inf = 0, sup = m.length - 1;

    do
    {
        mitad = (inf + sup) / 2;
```

```

        if (v > m[mitad])
            inf = mitad + 1;
        else
            sup = mitad - 1;
    }
    while( m[mitad] != v && inf <= sup);

    if (m[mitad] == v)
        return mitad;
    else
        return -1;
}

public static int búsquedaBin2(double[] m, double v)
{
    // El método búsquedaBin devuelve como resultado la posición
    // del valor. Si el valor no se localiza devuelve -1.

    if (m.length == 0) return -1;
    int mitad, inf = 0, sup = m.length - 1;

    do
    {
        mitad = (inf + sup) / 2;
        if (v > m[mitad])
            inf = mitad + 1;
        else
            sup = mitad;
    }
    while ( inf < sup );

    if (m[inf] == v)
        return inf;
    else
        return -1;
}

```

En cada iteración, en ambos casos, se divide en partes iguales el intervalo entre los índices *inf* y *sup*. Por ello, el número necesario de comparaciones es como mucho $\log_2 n$, siendo *n* el número de elementos de la matriz. Hasta aquí el comportamiento de ambas versiones es el mismo, pero ¿qué pasa con la condición de la sentencia **while**? Se observa que en la primera versión dicha sentencia realiza dos comparaciones frente a una que realiza en la segunda versión, lo que se traducirá en un mayor tiempo de ejecución, resultando, por tanto, ser más eficiente la versión segunda.

La aplicación siguiente permite ver de una forma práctica que la versión segunda emplea menos tiempo de ejecución que la primera. Esta aplicación crea una matriz *y*, utilizando primero una versión y después la otra, realiza una búsqueda por cada uno de sus elementos y dos búsquedas más para dos valores no pertenecientes a la matriz, uno menor que el menor y otro mayor que el mayor. El tiempo de ejecución medido en milisegundos se obtiene por diferencia de los tiempos devueltos por el método **currentTimeMillis** de la clase **System** al iniciar cada proceso de búsqueda y al finalizarlo.

```
class prueba
{
    public static int búsquedaBin1(double[] m, double v)
    {
        // Versión 1
    }

    public static int búsquedaBin2(double[] m, double v)
    {
        // Versión 2
    }

    public static void main(String[] args)
    {
        double[] a = new double[100000];
        long ti, n = a.length;
        int i;
        for (i = 0; i < n; i++)
            a[i] = i+1;

        // Versión 1
        ti = System.currentTimeMillis();
        i = búsquedaBin1(a,0);
        for (i = 0; i < n; i++)
            i = búsquedaBin1(a,i+1);
        i = búsquedaBin1(a,n+1);
        System.out.println((System.currentTimeMillis()-ti) + " milisegundos");

        // Versión 2
        ti = System.currentTimeMillis();
        i = búsquedaBin2(a,0);
        for (i = 0; i < n; i++)
            i = búsquedaBin2(a,i+1);
        i = búsquedaBin2(a,n+1);
        System.out.println((System.currentTimeMillis()-ti) + " milisegundos");
    }
}
```

2. Un centro numérico es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números cuyas sumas son iguales. El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1, 2, 3, 4, 5) y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595. Escribir un programa que calcule los centros numéricos entre 1 y n .

El ejemplo (1 a 5) 6 (7 a 8), donde se observa que 6 es un centro numérico, sugiere ir probando si los valores 3, 4, 5, 6, ..., cn , ..., $n-1$ son centros numéricos. En general cn es un centro numérico si la suma de todos los valores enteros desde 1 a $cn-1$ coincide con la suma desde $cn+1$ a lim_sup_grupo2 (límite superior del grupo segundo de números). Para que el programa sea eficiente, buscaremos el valor lim_sup_grupo2 entre los valores $cn+1$ y $n-1$ utilizando el método de *búsqueda binaria*. Recuerde que la suma de los valores enteros entre 1 y x viene dada por la expresión $(x * (x + 1)) / 2$.

El programa completo se muestra a continuación:

```
import java.util.*;
///////////////////////////////
// Calcular los centros numéricos entre 1 y n.
//
public class Test
{
    // Método de búsqueda binaria
    //
    // cn: centro numérico
    // (1 a cn-1) cn (cn+1 a mitad)
    // suma_grupol = suma de los valores desde 1 a cn-1
    // suma_grupo2 = suma de los valores desde cn+1 a mitad
    //
    // El método devuelve como resultado el valor mitad.
    // Si cn no es un centro numérico devuelve un valor 0.
    //
    public static long búsquedaBin(long cn, long n)
    {
        if (cn <= 0 || n <= 0) return 0;

        long suma_grupol = ((cn-1) * ((cn-1) + 1)) / 2;
        long suma_grupo2 = 0;
        long mitad = 0;

        long inf = cn+1; // límite inferior del grupo 2
        long sup = n;    // límite superior del grupo 2

        // Búsqueda binaria
```

```

do
{
    mitad = (inf + sup) / 2;
    suma_grupo2 = (mitad * (mitad + 1)) / 2 - suma_grupol - cn;
    if (suma_grupol > suma_grupo2)
        inf = mitad + 1;
    else
        sup = mitad - 1;
}
while ( suma_grupol != suma_grupo2 && inf <= sup);

if (suma_grupo2 == suma_grupol)
    return mitad;
else
    return 0;
}

public static void main(String[] args)
{
    long n;           // centros numéricos entre 1 y n
    long cn;          // posible centro numérico
    long lim_sup_grupo2; // límite superior del grupo 2

    System.out.print("Centros numéricos entre 1 y ");
    n = Leer.datoLong();
    System.out.println();
    for (cn = 3; cn < n; cn++)
    {
        lim_sup_grupo2 = búsquedabin(cn, n);
        if (lim_sup_grupo2 != 0)
            System.out.println(cn + " es centro numérico de 1 a " +
                (cn-1) + " y " + (cn+1) + " a " +
                lim_sup_grupo2);
    }
}
}

```

EJERCICIOS PROPUESTOS

1. Escribir un programa que calcule los centros numéricos entre *1* y *n* utilizando el algoritmo de búsqueda secuencial en lugar del de *búsqueda binaria*. Utilice el método **currentTimeMillis** de la clase **System** para comparar cuánto más lento es el método de *búsqueda secuencial* que el de *búsqueda binaria*.
2. Modificar la aplicación “lista de teléfonos” expuesta en el apartado “Ordenación de ficheros. Acceso aleatorio” de este mismo capítulo, para que el método *eliminar* de la clase **CListaTfnos** utilice el algoritmo de *búsqueda binaria*, lo que exigirá que los registros del fichero estén clasificados.

3. Añadir a la clase *CListaTfnos* de la aplicación realizada en el apartado “Ordenación de ficheros. Acceso aleatorio” de este mismo capítulo, un método denominado *inserción* que permita ordenar el fichero “lista de teléfonos” encapsulado por la misma, en orden descendente por el campo *nombre*.
4. Escribir una clase que incluya un método *ordenarMatriz2D* que permita ordenar ascendenteamente una matriz de dos dimensiones en la que cada elemento sea un objeto de la clase *CAlumno* expuesta anteriormente en este mismo capítulo.

Para realizar el proceso de ordenación emplee el método que quiera, pero hágalo directamente sobre la matriz por el campo *nombre*.

Finalmente, escriba una aplicación que pueda recibir el nombre de un fichero como argumento en la línea de órdenes, cuyos registros sean de la clase *CAlumno*, almacene los registros en una matriz de dos dimensiones y utilizando el método *ordenar* que acaba de escribir, ordene la matriz y la visualice una vez ordenada.

5. Realizar un programa que cree una lista dinámica a partir de una serie de números cualesquiera introducidos por el teclado. A continuación, ordenar la lista ascendente utilizando el método *quicksort*.
6. Escribir una aplicación que permita:

- a) Crear un fichero con la información de elementos del tipo:

```
public class CAalumno
{
    // Atributos
    private int matrícula;
    private String nombre;
    private double nota;

    // Métodos
    // ...
}
```

- b) Almacenar los registros en el fichero utilizando el *método hash abierto*.
- c) Obtener un registro por el número de matrícula utilizando el *método hash abierto*.

CAPÍTULO 15

© F.J.Ceballos/RA-MA

HILOS

Uno de los pasos importantes que la informática dio en favor de los desarrolladores de software fue colocar un nivel de software por encima del hardware de un ordenador. Este nivel de software, conocido como sistema operativo, es en esencia una interfaz fácil de utilizar que nos permite controlar todas las partes del hardware, en la mayoría de los casos, sin un profundo conocimiento del mismo.

A su vez, los sistemas operativos también han experimentado un gran avance, pasando de los sistemas de un único procesador a los actuales sistemas operativos distribuidos o de red, o a los sistemas operativos con multiprocesadores. Esta evolución ha desembocado en un mejor aprovechamiento de todos los recursos disponibles, permitiéndonos ejecutar cada vez más tareas en menos tiempo.

El concepto central de cualquier sistema operativo es el de *proceso*. Cualquier ordenador hoy en día es capaz de hacer varias cosas simultáneamente; por ejemplo, puede estar imprimiendo un documento por la impresora y ejecutando un programa del usuario. Esto requiere que la UCP (unidad central de proceso) alterne de un programa a otro en muy cortos espacios de tiempo, lo que conocemos como tiempo compartido. De esta forma, todos los programas, incluyendo los que componen el sistema operativo, que tengan que ejecutarse simultáneamente (multiprogramación) se organizan en varios procesos secuenciales.

CONCEPTO DE PROCESO

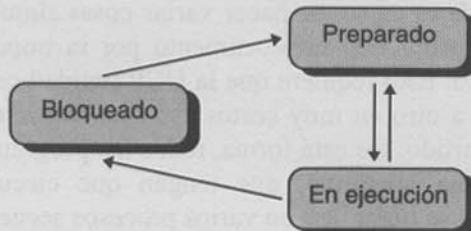
Un proceso es un ejemplar en ejecución de un programa. Cada proceso consta de bloques de código y de datos cargados desde un fichero ejecutable o desde una biblioteca dinámica. También es propietario de otros recursos que se crean durante la vida de dicho proceso y se destruyen cuando finaliza. Por ejemplo, un proceso posee:

- su propio espacio de direcciones,
- su memoria
- sus variables,
- ficheros abiertos,
- procesos hijo,
- contador de programa, registros, pila, señales, semáforos, etc.

Lo anterior, es equivalente a decir que cada proceso tiene su propia UCP virtual, lo que nos permite comprender mejor cómo un sistema puede ejecutar varios procesos simultáneamente, aunque la realidad sea que la UCP alterna entre esos procesos.

Según lo expuesto, sería un error confundir un programa con un proceso. Para evitar este posible malentendido, considere el siguiente ejemplo: cuando instalamos un juego en nuestro ordenador lo hacemos siguiendo las instrucciones adjuntas. En este caso, las instrucciones serían el programa, la actividad que hay que desarrollar para realizar la instalación (leer las instrucciones, introducir el CD-ROM, etc.) el proceso y nosotros la UCP. Si mientras estamos desarrollando esta actividad, alguien solicita nuestra colaboración para otra cosa, registramos el punto en el que nos encontramos y acudimos a resolver lo propuesto. En este caso, la UCP alterna de un proceso a otro.

De lo anterior se deduce que un proceso puede estar en *ejecución* (está utilizando la UCP), *preparado* (está detenido temporalmente para que se ejecute otro proceso) o *bloqueado* (no se puede ejecutar debido a que ocurrió algún evento al que hay que responder adecuadamente). Entre estos tres estados son posibles, como muestra la figura siguiente, cuatro transiciones:



Si un proceso en *ejecución* no puede continuar, pasa al estado de *bloqueado* o también, si puede continuar y el planificador decide que ya ha sido ejecutado el tiempo suficiente, pasa al estado de *preparado*. Si el proceso está *bloqueado* pasará a *preparado* cuando se dé el evento externo por el que se bloqueó y si está *preparado*, pasa a *ejecución* cuando el planificador lo decida porque los demás procesos ya han tenido su parte de tiempo de UCP.

En la UCP puede haber varios programas con varios procesos ejecutándose concurrentemente. En este caso se utilizan distintos mecanismos para la sincronización y comunicación entre procesos. Tales conceptos son parte del estudio de sistemas operativos.

HILOS

Un *hilo* (*thread* - llamado también proceso ligero o subproceso) es la unidad de ejecución de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila. Cuando se crea un proceso, el sistema operativo crea su primer hilo (hilo primario) el cual puede a su vez, crear hilos adicionales. Esto pone de manifiesto que un proceso no se ejecuta, sino que es sólo el espacio de direcciones donde reside el código que es ejecutado mediante uno o más hilos.

Según se ha expuesto en el apartado anterior, en un sistema operativo tradicional, cada proceso tiene un espacio de direcciones y un único *hilo de control*. Por ejemplo, considere un programa que incluya la siguiente secuencia de operaciones para actualizar el saldo de una cuenta bancaria cuando se efectúa un nuevo ingreso:

```
saldo = Cuenta.ObtenerSaldo();
saldo += ingreso;
Cuenta.EstablecerSaldo( saldo );
```

Este modelo de programación, en el que se ejecuta un solo *hilo*, es en el que estamos acostumbrados a trabajar habitualmente. Pero, continuando con el ejemplo anterior, piense en un banco real; en él, varios cajeros pueden actuar simultáneamente. Ejecutar el mismo programa por cada uno de los cajeros tiene un coste elevado (recuerde los recursos que necesita). En cambio, si el programa permitiera lanzar un hilo por cada petición de un cajero para actualizar una cuenta, estaríamos en el caso de múltiples hilos ejecutándose concurrentemente (*multi-threading*). Esta característica ya es una realidad en los sistemas operativos modernos de hoy y como consecuencia contemplada en los lenguajes de programación actuales.

Como ya hemos indicado, cada hilo se ejecuta en forma estrictamente secuencial y tiene su propia pila, el estado de los registros de la UCP y su propio contador de programa. En cambio, comparten el mismo espacio de direcciones, lo que significa compartir también las mismas variables globales, el mismo conjunto de ficheros abiertos, procesos hijos (no hilos hijo), señales, semáforos, etc.

Entonces ¿qué ventajas aporta un hilo respecto a un proceso? Los hilos comparten un espacio de memoria, el código y los recursos, por lo que el lanzamiento

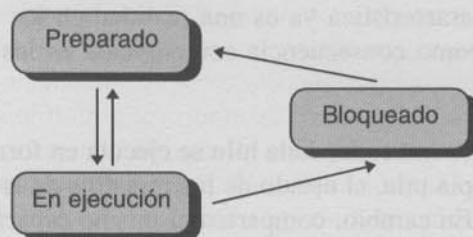
y la ejecución de un hilo es mucho más económica que el lanzamiento y la ejecución de un proceso. Por otra parte, muchos problemas pueden ser resueltos mejor con múltiples hilos; y si no, piense cómo escribiría un programa con un solo hilo de control para mostrar animación, sonido, visualizar documentos y traer ficheros de *Internet*, al mismo tiempo. No obstante, habrá situaciones en las que la mejor solución para ayudar en el trabajo sea crear un nuevo proceso (proceso hijo).

Los hilos comparten la UCP de la misma forma que lo hacen los procesos, pueden crear hilos hijo y se pueden bloquear. No obstante, mientras un hilo esté bloqueado se puede ejecutar otro hilo del mismo proceso, en el caso de hilos soportados por el *kernel* (núcleo del sistema operativo: programas en ejecución que hacen que el sistema funcione), no sucediendo lo mismo con los hilos soportados por una aplicación (por ejemplo, en Windows NT todos los hilos son soportados por el *kernel*). Un ejemplo, imaginemos que alguien llega a un cajero para depositar dinero en una cuenta y casi al mismo tiempo, un segundo cliente inicia la misma operación sobre la misma cuenta en otro cajero. Para que los resultados sean correctos, el segundo cajero quedará bloqueado hasta que el registro que está siendo actualizado por el primer cajero, quede liberado.

Resumiendo, sabemos que en la UCP puede haber varios programas con varios procesos ejecutándose concurrentemente, habilidad que se denomina multitarea, y a su vez, un proceso puede crear varios hilos y ejecutarlos de forma concurrente, lo que se traduce básicamente en una multitarea dentro de multitarea: el usuario sabe que puede ejecutar varias aplicaciones simultáneamente, y el programador sabe que cada aplicación puede ejecutar varios hilos a la vez.

Estados de un hilo

Igual que los procesos con un solo hilo de control, los *hilos* pueden encontrarse en uno de los siguientes estados:



- *Nuevo*. El hilo ha sido creado pero aún no ha sido activado. Cuando se active pasará al estado *preparado*.
- *Preparado*. El hilo está activo y está a la espera de que le sea asignada la UCP.

- *En ejecución.* El hilo está activo y le ha sido asignada la UCP (sólo los hilos activos, *preparados*, pueden ser ejecutados).
- *Bloqueado.* El hilo espera que otro elimine el bloqueo. Un hilo *bloqueado* puede estar:
 - ◊ *Dormido.* El hilo está bloqueado durante una cantidad de tiempo determinada, después de la cual despertará y pasará al estado *preparado*.
 - ◊ *Esperando.* El hilo está esperando a que ocurra alguna cosa: un mensaje **notify**, una operación de E/S o adquirir la propiedad de un método sincronizado. Cuando ocurra, pasará al estado *preparado*.
- *Muerto.* El hilo ha finalizado (está muerto) pero todavía no ha sido recogido por su padre. Los hilos *muertos* no pueden alcanzar ningún otro estado.

Observar que en la figura no se muestran los estados *nuevo* y *muerto* ya que no son estados de transición durante la vida del hilo; esto es, no se puede transitar al estado *nuevo* ni desde el estado *muerto*.

La transición entre estados está controlada por un planificador: parte del *kernel* encargada de que todos los hilos que esperan ejecutarse tenga su porción de tiempo de UCP. Si un hilo *en ejecución* no puede continuar, pasará al estado *bloqueado*; o también, si puede continuar y el planificador decide que ya ha sido ejecutado el tiempo suficiente, pasará al estado *preparado*. Si el proceso está *bloqueado* pasará a *preparado* cuando se dé el evento por el que espera; por ejemplo, puede estar esperando a que otro hilo elimine el bloqueo, o bien si está *dormido*, esperará a que pase el tiempo por el que fue enviado a este estado para ser activado; y si está *preparado*, pasará a *ejecución* cuando el planificador lo decide porque los demás hilos ya han tenido su parte de tiempo de UCP.

Cuándo se debe crear un hilo

Según lo expuesto anteriormente, cada vez que se crea un proceso, el sistema operativo crea un hilo primario. Para muchos procesos éste es el único hilo necesario. Sin embargo, un proceso puede crear otros hilos para ayudarse en su trabajo, utilizando la UCP al máximo posible. Por ejemplo, supongamos el diseño de una aplicación procesador de texto ¿Sería acertado crear un hilo separado para manipular cualquier tarea de impresión? Esto permitiría al usuario continuar utilizando la aplicación mientras se está imprimiendo. En cambio, ¿qué sucederá si los datos del documento cambian mientras se imprime? Éste es un problema que habría que resolver, quizás creando un fichero temporal que contenga los datos a imprimir.

Es evidente que los hilos son extraordinariamente útiles, pero también es evidente que si no se utilizan adecuadamente pueden introducir nuevos problemas mientras tratamos de resolver otros más antiguos. Por lo tanto, es un error pensar

que la mejor forma de desarrollar una aplicación es dividirla en partes que se ejecuten cada una como un hilo.

Cómo se crea un hilo

La mayoría del soporte que Java proporciona para trabajar con hilos reside en la clase **Thread** del paquete **java.lang**, aunque también la clase **Object**, la interfaz **Runnable** y las clases **ThreadGroup** y **ThreadDeath** del mismo paquete, así como la máquina virtual, proporcionan algún tipo de soporte.

Los hilos en Java se pueden crear de dos formas: escribiendo una nueva clase derivada de **Thread**, o bien haciendo que una clase existente implemente la interfaz **Runnable**.

La clase **Thread**, que implementa la interfaz **Runnable**, de forma resumida, está definida así:

```
public class Thread extends Object implements Runnable
{
    // Atributos
    static int MAX_PRIORITY;
        // Prioridad máxima que un hilo puede tener.
    static int MIN_PRIORITY;
        // Prioridad mínima que un hilo puede tener.
    static int NORM_PRIORITY;
        // Prioridad asignada por omisión a un hilo.

    // Constructores
    Thread([argumentos])

    // Métodos
    static Thread currentThread()
        // Devuelve una referencia al hilo que actualmente está
        // en ejecución.
    void destroy()
        // Destruye este hilo, sin realizar ninguna operación de
        // limpieza.
    String getName()
        // Devuelve el nombre del hilo.
    int getPriority()
        // Devuelve la prioridad del hilo.
    void interrupt()
        // Envía este hilo al estado de preparado.
    boolean isAlive()
        // Verifica si este hilo está vivo (no ha terminado).
```

```

boolean isDaemon()
    // Verifica si este hilo es un demonio. Se da este nombre a
    // un hilo que se ejecuta en segundo plano, realizando una
    // operación específica en tiempos predefinidos, o bien en
    // respuesta a ciertos eventos.
boolean isInterrupted()
    // Verifica si este hilo ha sido interrumpido.
void join([milisegundos[, nanosegundos]])
    // Espera indefinidamente o el tiempo especificado, a que este
    // hilo termine (a que muera).
void run()
    // Contiene el código que se ejecutará cuando el hilo pase
    // al estado de ejecución. Por omisión no hace nada.
void setDaemon(boolean on)
    // Define este hilo como un demonio o como un hilo de usuario.
void setName(String nombre)
    // Cambia el nombre de este hilo.
void setPriority(int nuevaPrioridad)
    // Cambia la prioridad de este hilo. Por omisión es normal
    // (NORM_PRIORITY).
static void sleep(long milisegundos[, int nanosegundos])
    // Envía este hilo a dormir por el tiempo especificado.
void start()
    // Inicia la ejecución de este hilo: la máquina virtual de Java
    // invoca al método run de este hilo.
static void yield()
    // Detiene temporalmente la ejecución de este hilo para
    // permitir la ejecución de otros.

// Métodos heredados de la clase Object: notify, notifyAll y wait
void notify()
    // Despierta un hilo de los que están esperando por el
    // monitor de este objeto.
void notifyAll()
    // Despierta todos los hilos que están esperando por el
    // monitor de este objeto.
void wait([milisegundos[, nanosegundos]])
    // Envía este hilo al estado de espera hasta que otro hilo
    // invoque al método notify o notifyAll, o hasta que transcurra
    // el tiempo especificado.
}

```

Una clase que implemente la interfaz **Runnable** tiene que sobreescribir el método **run** aportado por ésta, de ahí que **Thread** proporcione este método aunque no haga nada. El método **run** contendrá el código que debe ejecutar el hilo.

Los métodos **stop**, **suspend**, **resume** y **runFinalizersOnExit** incluidos en versiones anteriores al *jdk1.2* han sido desaprobados porque son intrínsecamente inseguros. Para más detalles vea la ayuda suministrada por *Sun*.

Hilo derivado de Thread

Según hemos dicho anteriormente, un hilo puede ser un objeto de una subclase de la clase **Thread**. Entonces, para que una aplicación pueda lanzar un determinado hilo de ejecución, el primer paso es escribir la clase del hilo derivada de **Thread** y sobreescribir el método **run** heredado por ésta, con el fin de especificar la tarea que tiene que realizar dicho hilo.

Por ejemplo, supongamos que queremos escribir una aplicación elemental que en un instante determinado de su ejecución lance un hilo que realice un simple conteo. La clase del hilo puede ser la siguiente:

```
public class ContadorAdelante extends Thread
{
    public ContadorAdelante(String nombre) // constructor
    {
        if (nombre != null) setName(nombre);
        start(); // el hilo ejecuta su propio método run
    }
    public ContadorAdelante() { this(null); } // constructor

    public void run()
    {
        for (int i = 1; i <= 1000; i++)
        {
            System.out.print(getName() + " " + i + "\r");
        }
        System.out.println();
    }
}
```

La clase *ContadorAdelante* es una subclase de **Thread** y sobreescribe el método **run** heredado. Lo que hace este método es escribir el nombre del hilo seguido de un contador de 1 a 1000.

Para poder lanzar un hilo de la clase *ContadorAdelante*, primero tenemos que construir un objeto de esa clase y después enviar a dicho objeto el mensaje **start**; De esto último se encarga el constructor de la clase. La siguiente aplicación muestra un ejemplo:

```
public class Test
{
    public static void main(String[] args)
    {
        ContadorAdelante cuentaAdelante = new ContadorAdelante("Contador+");
    }
}
```

El operador **new** crea un hilo *cuentaAdelante* (el hilo está en el estado *nuevo*). El método **start** cambia el estado del hilo a *preparado*. De ahora en adelante y hasta que finalice la ejecución del hilo *cuentaAdelante*, será el *planificador de hilos* el que determine cuándo éste pasa al estado de *ejecución* y cuándo lo abandona para permitir que se ejecuten simultáneamente otros hilos.

Según lo expuesto, el método **start** no hace que se ejecute inmediatamente el método **run** del hilo, sino que lo sitúa en el estado *preparado* para que compita por la UCP junto con el resto de los hilos que haya en este estado. Sólo el *planificador* puede asignar tiempo de UCP a un hilo y lo hará con *cuentaAdelante* en cualquier instante después de que haya recibido el mensaje **start**. Por lo tanto, un hilo durante su tiempo de vida, gasta parte de él en ejecutarse y el resto en permanecer en alguno de los estados distintos al de *ejecución*. Más adelante aprenderá cómo un hilo transita entre los diferentes estados.

Lo que no se debe de hacer es llamar directamente al método **run**; esto ejecutaría el código de este método sin que intervenga el *planificador*. Quiere esto decir que es el método **start** el que registra el hilo en el *planificador de hilos*.

Hilo asociado con una clase

Cuando sea necesario que un hilo ejecute el método **run** de un objeto de cualquier otra clase que no esté derivada de **Thread**, los pasos a seguir son los siguientes:

1. El objeto debe ser de una clase que implemente la interfaz **Runnable**, ya que es esta la que aporta el método **run**.
2. Sobreescribir el método **run** con las sentencias que tiene que ejecutar el hilo.
3. Crear un objeto de esa clase.
4. Crear un objeto de la clase **Thread** pasando como argumento al constructor, el objeto cuya clase incluye el método **run**.
5. Invocar al método **start** del objeto **Thread**.

Por ejemplo, la siguiente clase implementa la interfaz **Runnable** y sobrescribe el método **run** proporcionado por ésta:

```
public class ContadorAtras implements Runnable
{
    private Thread cuentaAtrás;
    public ContadorAtras(String nombre) // constructor
    {
        cuentaAtrás = new Thread(this); // objeto de la clase Thread
        if (nombre != null) cuentaAtrás.setName(nombre);
        cuentaAtrás.start(); // el hilo ejecuta el método run de
                            // ContadorAtras
    }
}
```

```

public ContadorAtras() { this(null); } // constructor

public void run()
{
    for (int i = 1000; i > 0; i--)
    {
        System.out.print("\t\t" + cuentaAtrás.getName() + " " + i + "\r");
    }
    System.out.println();
}
}

```

La clase *ContadorAtras* no se deriva de **Thread**. Sin embargo, tiene un método **run** proporcionado por la interfaz **Runnable**. Por ello, cualquier objeto *ContadorAtras* puede ser pasado como argumento cuando se invoque al constructor de la clase **Thread** cuya sintaxis es:

`Thread(Runnable objeto)`

Para poder lanzar un hilo asociado con la clase *ContadorAtras*, primero tenemos que construir un objeto de la misma, después un objeto de la clase **Thread** pasando como argumento el objeto de la clase *ContadorAtras* y finalmente, enviar al objeto **Thread** el mensaje **start**; de estas dos últimas operaciones se encarga el constructor *ContadorAtras*. La siguiente aplicación muestra un ejemplo:

```

public class Test
{
    public static void main(String[] args)
    {
        ContadorAtras objCuentaAtrás = new ContadorAtras("Contador-");
    }
}

```

Esta forma de lanzar un hilo quizás sea un poco más complicada. Sin embargo, hay razones suficientes para hacer este pequeño esfuerzo. Si el método **run** es parte de la interfaz de una clase cualquiera, tiene acceso a todos los miembros de esa clase, cosa que no ocurre si pertenece a una subclase de **Thread**. Otra razón es que Java no permite la herencia múltiple; entonces, si escribimos una clase derivada de **Thread**, esa clase no puede ser a la vez una subclase de cualquier otra. Por ejemplo, para poder asociar un hilo con la clase *CCuentaAhorro* derivada de *CCuenta* (capítulo 10), la única forma de hacerlo es que *CCuentaAhorro* implemente la interfaz **Runnable**.

Finalmente, a pesar de que en ocasiones hablamos en términos similares a: “la clase *ContadorAtras* es un hilo”, desde el punto de vista de la programación orientada a objetos no es correcto expresarse así, a pesar de entendernos. Lo único

que es correcto es: "la clase *ContadorAtras* está asociada con un hilo". Observe en el ejemplo anterior que el hilo es el objeto de la clase **Thread**, no el objeto de la clase *ContadorAtras*. Entonces, siempre que necesitemos que una clase tenga el comportamiento de un hilo, deberemos implementar en la misma la interfaz **Runnable** y sobreescribir el método **run**.

Como ejercicio, pruebe a ejecutar la aplicación siguiente y podrá observar como los dos hilos, *cuentaAdelante* y *cuentaAtrás*, se ejecutan simultáneamente.

```
public class Test
{
    public static void main(String[] args)
    {
        ContadorAdelante cuentaAdelante = new ContadorAdelante("Contador+");
        ContadorAtras objCuentaAtrás = new ContadorAtras("Contador-");
    }
}
```

Cuando ejecute la aplicación anterior, el sistema lanza la ejecución del hilo primario (hilo padre) el cual, al ejecutarse, lanza la ejecución del hilo *cuentaAdelante* y la ejecución del hilo *cuentaAtrás*, finalizando así la ejecución de **main**; no obstante, este método no retornará hasta que no hayan finalizado los hilos hijo; esto es, el hilo primario no termina mientras no terminen sus hilos hijo.

Demonios

Un *demonio*, a diferencia de los hilos tradicionales, no forma parte de la esencia del programa, sino de la máquina Java. Los *demonios* son usados generalmente para prestar servicios en segundo plano a todos los programas que puedan necesitar el tipo de servicio proporcionado. Por ejemplo, el recolector de basura de Java es un ejemplo de este tipo de hilos.

Para crear un hilo *demonio* simplemente hay que crear un hilo normal y enviarle el mensaje **setDaemon**:

```
hilo.setDaemon(true);
```

Si un hilo es un *demonio*, entonces cualquier hilo que el cree será automáticamente un *demonio*.

Para saber si un hilo es un *demonio* simplemente hay que enviarle el mensaje **isDaemon**. El método que se ejecuta devolverá **true** si el hilo es un *demonio* y **false** en caso contrario:

```
boolean b = hilo.isDaemon();
```

El intérprete Java normalmente permanece en ejecución hasta que todos los hilos en el sistema finalizan su ejecución. Sin embargo, los *demonios* son una excepción, ya que su labor es proporcionar servicios a otros programas. Por lo tanto, no tiene sentido continuar ejecutándolos cuando ya no haya programas en ejecución. Por esta razón, el intérprete Java finalizará cuando todos los hilos que queden en ejecución sean *demonios*. El siguiente ejemplo muestra cómo implementar un hilo demonio:

```
//////////  
// Hilo demonio. Suena "bip" aproximadamente cada segundo  
//  
public class CDemonio extends Thread  
{  
    public CDemonio()  
    {  
        setDaemon(true);  
        start();  
    }  
  
    public void run()  
    {  
        char bip = '\u0007';  
        while (true)  
        {  
            try  
            {  
                sleep(1000); // 1 segundo  
            }  
            catch (InterruptedException e) {}  
            System.out.print(bip);  
        }  
    }  
}//////////
```

Para iniciar un demonio, *dbip*, de la clase *CDemonio* basta con escribir una sentencia como la siguiente:

```
CDemonio dbip = new CDemonio();
```

Finalizar un hilo

Un hilo termina de forma natural cuando su método **run** devuelve el control. Cuando esto sucede el hilo pasa al estado *muerto* (ha terminado) y no hay forma de salir de este estado. Esto es, una vez que el hilo está muerto, no puede ser arrancado otra vez; si deseamos ejecutar otra vez la tarea desempeñada por el hilo

hay que construir un nuevo objeto hilo y enviarle el mensaje **start**, pero sí se puede invocar a sus métodos.

Por ejemplo, supongamos una clase *ContadorAdelante* que muestra un contador ascendente que será detenido cuando el atributo *continuar* sea **false**. La clase, además de este atributo y del método **run** que muestra la cuenta, tiene un método **terminar** que pone el atributo *continuar* a **false**, y dos constructores: el primero inicia el hilo con el nombre asignado por omisión y el segundo, también lo inicia pero con el nombre pasado como argumento.

```
//////////  
// Clase que define un hilo que cuenta ascendente mente mientras  
// que el atributo continuar sea true.  
//  
public class ContadorAdelante extends Thread  
{  
    private boolean continuar = true;  
  
    public ContadorAdelante()  
    {  
        start();  
    }  
  
    public ContadorAdelante(String nombreHilo)  
    {  
        setName(nombreHilo);  
        start();  
    }  
  
    public void run()  
    {  
        int i = 1;  
        while (continuar)  
        {  
            System.out.print(getName() + " " + i++ + "\r");  
        }  
        System.out.println();  
    }  
  
    public void terminar()  
    {  
        continuar = false;  
    }  
}//////////
```

La siguiente aplicación inicia un demonio de la clase *CDemonio* expuesta anteriormente y un hilo de la clase *ContadorAdelante*. Mientras este hilo muestra

un contador ascendente en la pantalla, el demonio hace sonar un *bip* cada segundo. El contador se detendrá cuando el usuario pulse la tecla *Entrar*.

```

import java.io.*;
///////////////
// Terminar un hilo.
//
public class Test
{
    public static void main(String[] args)
    {
        // Lanzar el demonio dbip
        CDemonio dbip = new CDemonio();

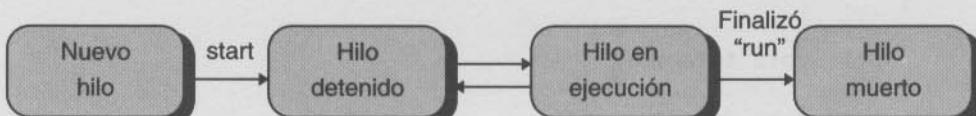
        // Lanzar el hilo cuentaAdelante
        ContadorAdelante cuentaAdelante = new ContadorAdelante("Contador+");

        System.out.println("Pulse [Entrar] para finalizar");
        InputStreamReader is = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(is);
        try
        {
            br.readLine(); // ejecución detenida hasta pulsar [Entrar]
        }
        catch (IOException e) {}
        // Permitir al hilo cuentaAdelante finalizar
        cuentaAdelante.terminar();
    }
}
/////////////

```

Controlar un hilo

Ahora que ya hemos visto cómo realizar una determinada tarea utilizando un hilo, podemos deducir fácilmente que su ciclo de vida evoluciona según muestra la figura siguiente:



Un hilo, durante su ciclo de vida está transitando por los estados: *nuevo*, *preparado*, *en ejecución*, *bloqueado* y *muerto*, estudiados anteriormente. El estado *en ejecución* se corresponde en la figura con el bloque “hilo en ejecución”, el cual se alcanza desde el estado *preparado* al que pasa el hilo después de que haya sido

creado y haber recibido el mensaje **start**, y cuando el hilo está vivo y no está en ejecución es que está detenido, bloque “hilo detenido”.

Precisamente, el método **isAlive** de **Thread** devuelve **true** si el hilo que recibe este mensaje ha sido arrancado (**start**) y todavía no ha muerto.

Normalmente es el planificador el que controla cuándo un hilo debe estar en ejecución y cuándo pasa a estar detenido, pero en ocasiones tendremos que ser nosotros los que programemos las circunstancias bajo las cuales un hilo pueda pasar a ejecución, o bien deba pasar de ejecución a algunos de los estados *preparado* o *bloqueado* (bloqueado porque esté dormido, esté esperando a que otro hilo lo desbloquee, o esperando a que termine una operación de E/S, o bien esperando a apropiarse de un método sincronizado).

Preparado

A un hilo en ejecución se le puede enviar un mensaje **yield** para que se mueva al estado *preparado* y ceda así la UCP a otros hilos que estén compitiendo por ella (hilos que están en el estado *preparado*). Si el planificador observa que no hay ningún hilo esperando por la UCP, permitirá que el hilo que iba a ceder la UCP continúe ejecutándose.



El método **yield** es **static**, por lo tanto, opera sobre el hilo que actualmente se esté ejecutando. Cuando necesite invocarlo basta con que escriba: *Thread.yield()*.

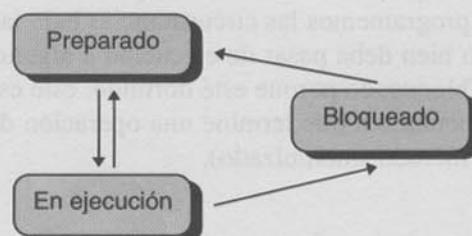
Bloqueado

Muchos métodos que ejecutan operaciones de entrada tienen que esperar por alguna circunstancia en el mundo exterior antes de que ellos puedan proseguir; este comportamiento se conoce como *bloqueo*. Por ejemplo, la sentencia siguiente lee un byte de la entrada estándar lanzando un hilo que ejecuta **read**:

```
n = System.in.read();
```

Si en la entrada estándar hay un byte disponible la sentencia anterior se ejecuta satisfactoriamente y la ejecución del método que la contiene continúa. Sin embargo, si no hay un byte disponible, **read** tiene que esperar hasta que haya uno.

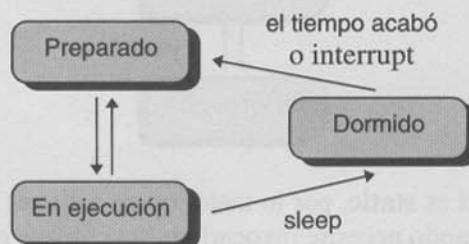
Si el hilo que ejecuta **read** se mantuviera en el estado de ejecución, la UCP quedaría ocupada y no se podría realizar nada más. En general, si un método necesita esperar una cantidad de tiempo indeterminada hasta que la ocurrencia que lo detiene tenga lugar, el hilo en ejecución debe salir de este estado. Todos los métodos Java que permiten leer datos se comportan de esta forma. Un hilo que gentilmente abandona el estado de ejecución hasta que se dé la ocurrencia que lo detiene se dice que está *bloqueado*.



Java implementa muchos de los bloqueos que ocurren durante una operación de E/S llamando a los métodos **sleep** y **wait** que vemos a continuación.

Dormido

Un hilo dormido pasa tiempo sin hacer nada, por lo tanto, no utiliza la UCP.



Una llamada al método **sleep** solicita que el hilo actualmente en ejecución cese durante un tiempo específico. Hay dos formas de llamar a este método:

```
Thread.sleep(milisegundos);
Thread.sleep(milisegundos, nanosegundos);
```

Se puede observar que el método **sleep**, igual que **yield**, es **static**. Ambos métodos operan sobre el hilo que actualmente se esté ejecutando.

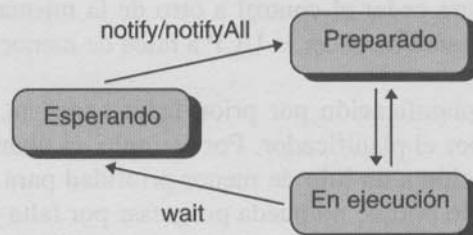
La figura anterior indica que cuando un hilo despierta (el tiempo que tenía que dormir ha transcurrido) no continúa la ejecución, sino que se mueve al estado

preparado. Pasará a ejecución cuando el planificador lo indique. Esto significa que una llamada a `sleep` bloqueará un hilo por un tiempo superior al especificado.

La clase **Thread** proporciona también un método **interrupt**. Cuando un hilo dormido recibe este mensaje, pasa automáticamente al estado *preparado*, y cuando pase a *ejecución*, ejecutará su manejador **InterruptedException**.

Esperando

El método **wait** mueve un hilo en ejecución al estado *esperando* y el método **notify** o **notifyAll** mueven un hilo que esté *esperando* al estado *preparado*; **notifyAll** mueve todos los hilos que estén *esperando* al estado *preparado*. Estos métodos, que veremos más adelante con más detalle, se utilizan para sincronizar hilos.



Planificación de hilos

Muchos ordenadores tienen sólo una UCP, así que los hilos que requieran ejecutarse deben compartirla. La ejecución de múltiples hilos sobre una única UCP, en cierto orden, es llamada *planificación*. La máquina Java (*Java Runtime Environment - JRE*: máquina virtual de Java, incluido el planificador, clases del núcleo central de Java y los ficheros de soporte) soporta un algoritmo de planificación determinista (en cualquier momento se puede saber qué hilo se está ejecutando o cuánto tiempo continuará ejecutándose) muy simple, conocido como *fixed priority scheduling* (planificación por prioridad: el hilo que se elige para su ejecución es el de prioridad más alta). Esto es, la planificación de la UCP es totalmente por derecho de prioridad (*preemptive*).

Lo anteriormente expuesto significa que cada hilo Java tiene asignado una prioridad definida por un valor numérico entre `MIN_PRIORITY` y `MAX_PRIORITY` (constantes definidas en la clase **Thread**), de forma que cuando varios hilos estén *preparados*, será elegido para su ejecución el de mayor prioridad. Solamente cuando la ejecución de ese hilo se detenga por cualquier causa, podrá ejecutarse un hilo de menor prioridad; y cuando un hilo con prioridad más alta que el

que actualmente se está ejecutando se mueva al estado *preparado*, pasará automáticamente a ejecutarse.

Según lo expuesto, la máquina Java no reemplazará el hilo actual en ejecución por otro hilo de la misma prioridad. En otras palabras, la máquina Java no aplica una planificación por *cuantos* (*time-slice -- quanto* o rodaja de tiempo --: tiempo máximo que un hilo puede retener la UCP; esta planificación da lugar a un sistema no determinista), aunque la implementación del sistema de hilos que subyace en la clase **Thread** puede soportar *cuantos*. Por lo tanto, no escribir código que dependa de *cuantos* porque como se indica a continuación, en unas máquinas virtuales puede funcionar (en Windows y MacIntosh) y en otras no (en Solaris).

Después de lo dicho, sería bueno al programar que nuestros hilos cedieran voluntariamente el control algunas veces. Un hilo dado puede renunciar a su derecho de ejecutarse para ceder el control a otro de la misma prioridad llamando al método **yield**. Un intento de ceder la UCP a hilos de menor prioridad se ignorará.

La política de planificación por prioridades expuesta, puede verse en algún momento alterada por el planificador. Por ejemplo, el planificador de hilos puede elegir para su ejecución a un hilo de menor prioridad para evitar que quede completamente bloqueado porque no pueda progresar por falta de los recursos necesarios para ello (puede morir por falta de recursos: inanición -- *starvation*). Por esta razón, la exactitud de los algoritmos programados no debe basarse en la prioridad de los hilos.

¿Qué ocurre con los hilos que tengan igual prioridad?

Cuando todos los hilos que compiten por la UCP tienen la misma prioridad, el planificador elige para su ejecución al siguiente según el orden resultante de aplicar el algoritmo *round-robin* (no *preemptive*). En este caso, la cola de hilos listos para ejecutarse se trata como una cola circular *FIFO*. La UCP será cedida a otro hilo bien porque:

- un hilo de prioridad más alta ha alcanzado el estado de *preparado*;
- cede la UCP, o su método **run** finaliza;
- se supera el *cuanto* (*quantum*): tiempo máximo que un hilo puede retener la UCP. Esta tercera condición sólo es aplicable en sistemas que soporten la planificación por *cuantos*. En este aspecto, la especificación Java da mucha libertad. Cada máquina virtual implementa como quiere este agujero en la definición, cumpliendo perfectamente el estándar. Por ejemplo, las plataformas Windows 9x/NT/2000 y MacIntosh admiten planificación por *cuantos* para hilos con la misma prioridad; en cambio, Solaris planifica por *cooperación*, es decir, los hilos deben ceder voluntariamente la UCP.

Resumiendo, cuando se ejecuta un proceso que tiene varios hilos *preparados*, la máquina Java asigna la UCP en función de la prioridad que tenga asignada el hilo activo: de mayor prioridad a menor prioridad. En Java, los hilos tienen asignadas prioridades de 1 a 10 (10 es la prioridad más alta: *MAX_PRIORITY*). Por otra parte, cuando el sistema asigna la UCP a un hilo, trata de igual forma a todos los hilos de la misma prioridad. Esto es, asigna un *cuanto* al primer hilo *preparado* de prioridad 10, cuando éste finaliza su intervalo de tiempo, asigna otro *cuanto* al siguiente hilo *preparado* de prioridad 10 y así sucesivamente. Cuando todos los hilos de prioridad 10 han tenido su intervalo de tiempo, se empieza otra vez por el primero.

Según lo expuesto ¿cómo permitir la ejecución de hilos con prioridad inferior? La respuesta está en saber que muchos hilos del sistema son detenidos de vez en cuando, por motivos diferentes. Así, cuando todos los hilos de prioridad 10 estén detenidos, el sistema asigna *cuantos* a los hilos *preparados* de prioridad 9. Un razonamiento análogo nos conduce a pensar que los hilos de prioridad 8 sólo pueden ejecutarse cuando los hilos de prioridades 10 y 9 estén detenidos. Parece entonces, que los procesos de prioridad 1 nunca se ejecutarán, o que se ejecutarán de tarde en tarde. Pero, la verdad es que no es así. La mayoría de los hilos consumen su tiempo durmiendo, lo que permite la ejecución de los hilos de prioridades bajas con una frecuencia, probablemente, un poco inferior.

Asignar prioridades a los hilos

Cuando se crea un hilo Java, éste hereda su prioridad del hilo que lo crea. No obstante, es posible aumentar o disminuir esta prioridad. Para modificar la prioridad de un hilo utilice el método **setPriority**:

```
hilo.setPriority(nuevaPrioridad);
```

La siguiente tabla muestra las constantes correspondientes a las prioridades definidas en la clase **Thread**:

Constante	Valor
MIN_PRIORITY	1
NORM_PRIORITY	5 (valor por omisión)
MAX_PRIORITY	10

Para obtener la prioridad que tiene un hilo utilice el método **getPriority**:

```
int p = getPriority();
```

El valor que devuelve este método está comprendido entre `MIN_PRIORITY` y `MAX_PRIORITY`.

El siguiente ejemplo implementa una aplicación que visualiza n contadores. Por ejemplo, para 2 contadores la pantalla mostraría una línea con el siguiente formato: *nombre del hilo, prioridad y cuenta*.

```
Thread-1, P-2 2410 Thread-2, P-3 465771
```

Cada contador es un hilo. Las prioridades asignadas a cada hilo son diferentes (2, 3, etc.). La clase que da lugar a cada objeto hilo contador es la siguiente:

```
public class Contador extends Thread
{
    public int cuenta;
    private double suma = 0;

    public void run()
    {
        for (cuenta = 0; cuenta < 500000; cuenta++)
        {
            // Realizar algunos cálculos
            suma += Math.random();
        }
    }

    // Otros métodos
}
```

El método `run` de la clase simplemente genera, cuenta y acumula 500000 números aleatorios. El miembro `cuenta` es público porque otro hilo *Cuentas* utilizará ese valor para mostrar el progreso de cada uno de los contadores.

La clase *Cuentas* se implementa también como un hilo encargado de lanzar las cuentas. Su método `run` contiene un bucle que se ejecutará mientras los hilos contadores estén vivos; en cada iteración mostrará por cada hilo contador su nombre, prioridad y estado de la cuenta, y esperará durante n Milisegundos. La prioridad del hilo *Cuentas* será $(nCuentas+2)\%Thread.MAX_PRIORITY$, donde $nCuentas$ es el número de hilos contador. Por ejemplo, para 2 hilos contador la prioridad del primer hilo, `Contador[0]`, será 2, la del segundo, `Contador[1]`, será 3 y la del hilo de la clase *Cuentas*, 4. De esta forma, mientras el hilo *Cuentas* duerme los hilos contador compiten por la UCP (lógicamente finalizará antes la cuenta el de mayor prioridad) y cuando despierte, por ser el hilo de mayor prioridad obtendrá inmediatamente la UCP y mostrará los resultados actuales de las cuentas. A continuación se muestra el código correspondiente a esta clase:

```
public class Cuentas extends ... ead
{
    private static int nCuentas;
    private Contador[] cuenta;

    public Cuentas(int n)
    {
        nCuentas = n; // número de hilos contadores

        // Establecer la prioridad de este hilo
        setPriority((nCuentas+2)%Thread.MAX_PRIORITY);

        // Crear y establecer las prioridades de los hilos contador
        cuenta = new Contador[nCuentas];
        for (int i = 0; i < nCuentas; i++)
        {
            cuenta[i] = new Contador();
            cuenta[i].setPriority((i+3)%Thread.MAX_PRIORITY-1);
        }
    }

    public void run()
    {
        int i;
        boolean hayaHilosVivos;

        // Mostrar el nombre y la prioridad de este hilo
        System.out.println(this.getName() + ", P-" +
                           this.getPriority());
        // Lanzar los hilos contadores para su ejecución
        for (i = 0; i < nCuentas; i++)
            cuenta[i].start();

        do
        {
            // Mostrar nombre del hilo, prioridad y estado de la cuenta
            for (i = 0; i < nCuentas; i++)
                System.out.print(cuenta[i].getName() +
                                ", P-" + cuenta[i].getPriority() + " " +
                                cuenta[i].cuenta + " ");

            System.out.print("\r");
            // ¿Hay hilos vivos?
            hayaHilosVivos = cuenta[0].isAlive();
            for (i = 1; i < nCuentas; i++)
                hayaHilosVivos = hayaHilosVivos || cuenta[i].isAlive();
            // Ahora el hilo dormirá nMilisegundos, mientras los hilos
            // contadores siguen su curso.
            try
            {
                int nMilisegundos = (int)(10 * Math.pow(2,nCuentas));
            }
        }
    }
}
```

```
    sleep(nMilisegundos);
}
catch (InterruptedException e) { }
}
while (hayAHIlosVivos);
}
```

La aplicación *Test* que muestre los resultados perseguidos puede ser la siguiente:

```
public class Test
{
    public static void main(String[] args)
    {
        int nCuentas = 2; // número de contadores
        // Crear y lanzar el hilo Cuentas
        Cuentas hiloCuentas = new Cuentas(nCuentas);
        hiloCuentas.start();
    }
}
```

En este ejemplo que acabamos de realizar, la política de planificación es por derecho de prioridad.

¿Qué pasará si eliminamos la sentencia *sleep(nMilisegundos)*? Pues que la política de planificación seguida se ve alterada por el planificador para evitar que el hilo de mayor prioridad se apodere de la UCP (hilo egoísta); pruébelo (evitar *starvation*). Sistemas como Windows 9x/NT/2000 pelean contra los “hilos egoístas” con la estrategia de asignar la UCP por *cuantos (time-slicing)*.

¿Qué sucede si asignamos a todos los hilos contador la misma prioridad? En esta situación, el planificador elegirá el siguiente para ejecución según el modelo *round-robin* y en el caso de Windows asignará, además, la UCP por *cuantos*.

SINCRONIZACIÓN DE HILOS

En los ejemplos que hemos visto hasta ahora cada hilo contenía todo lo que necesitaba para su ejecución: datos y métodos. Además, cada uno de ellos se podía ejecutar sin que interfiriera en la ejecución de cualquier otro hilo que se ejecutara concurrentemente con él. Estamos en el caso de *hilos independientes*.

Sin embargo, hay muchas situaciones en las que dos o más hilos ejecutándose concurrentemente deben acceder a los mismos recursos y/o datos. Como ejemplo, imagine la situación donde dos hilos acceden al mismo fichero de datos; un hilo

puede escribir en el fichero mientras el otro simultáneamente lee del mismo. Estamos en el caso de *hilos cooperantes*. Este tipo de situación puede crear resultados impredecibles, además de indeseables. En estos casos, simplemente se debe tomar el control de la situación y asegurar que cada hilo acceda a los recursos de una manera previsible, sincronizando las actividades que desarrollan cada uno de ellos. Para realizar operaciones de sincronización Java proporciona los siguientes elementos de sincronización: *secciones críticas*, *wait* y *notify*.

En general un hilo se sincroniza con otro hilo poniéndose él mismo a dormir. No obstante, antes de ponerse a dormir, debe poner en conocimiento del sistema qué evento debe ocurrir para reanudar su ejecución. De esta forma, cuando se produzca ese evento, el sistema despertará al hilo permitiéndole continuar la ejecución. Por ejemplo, si un hilo padre necesita esperar hasta que uno o más hilos hijo finalicen, se pone él mismo a dormir hasta que el hilo o hilos hijo pasen al estado *muerto*.

Cuando un hilo se pone a dormir (se bloquea), no entra en la planificación del sistema; esto es, el planificador no le asigna tiempo de UCP y, por consiguiente, detiene su ejecución.

Secciones críticas

Supongamos una aplicación en la que dos hilos de un proceso acceden a una única matriz de datos con la intención de registrar los resultados obtenidos durante un experimento. El programa podría simularse así:

- Creamos un objeto *datos* que envuelva una matriz unidimensional con el propósito de almacenar los datos adquiridos a través de una tarjeta que actúa como interfaz entre nuestra aplicación y el medio utilizado para realizar el experimento. En nuestro ejemplo, simularemos cada uno de los datos adquiridos con un valor obtenido a partir de unos sencillos cálculos.
- Creamos uno o más hilos para que tomen los datos y los vayan almacenando en la matriz hasta llenarla, instante en el que su ejecución finalizará.

Implementemos una clase *CDatos* para manipular una matriz unidimensional de tipo **double** con *n* elementos. Dicha clase incluirá los atributos:

- *datos*: matriz de tipo **double**.
- *ind*: índice del siguiente elemento vacío.
- *tamaño*: número de elementos de la matriz.

y los métodos:

- *CDatos*: es el constructor de la clase. Crea la matriz con n elementos, valor que se pasa como argumento, o con 10 si el valor pasado no es válido; también inicia *tamaño* con el número de elementos.
- *obtener*: devuelve el valor de un determinado elemento.
- *asignar*: asigna un valor a un determinado elemento.
- *cálculos*: obtiene el siguiente valor a almacenar en la matriz. Recibe como argumento el nombre del hilo en ejecución y devuelve el índice del siguiente elemento vacío.

```

public class CDatos
{
    // Atributos
    private double[] dato;
    private int ind = 0;
    public int tamaño;

    // Métodos
    public CDatos(int n)
    {
        if (n < 1) n = 10;
        tamaño = n;
        dato = new double[n];
    }

    public double obtener(int i)
    {
        return dato[i];
    }

    public void asignar(double x, int i)
    {
        dato[i] = x;
    }

    public int cálculos(String hilo)
    {
        if (ind >= tamaño) return tamaño;
        double x = Math.random();
        System.out.println(hilo + " muestra " + ind);
        asignar(x, ind);
        ind++;
        return ind;
    }
}

```

Uno o más hilos serán los encargados de adquirir los datos. Quiere esto decir que cuando se lancen estos hilos, el constructor de cada uno de ellos debe de recibir como argumento el objeto *CDatos* donde serán almacenados los datos que se

adquirirán, ejecutando el método *cálculos* del objeto *CDatos*. La clase de los hilos aludidos puede ser así:

```
public class CAdquirirDatos extends Thread
{
    private CDatos m; // objeto para almacenar los datos
    public CAdquirirDatos(CDatos mdatos) // constructor
    {
        m = mdatos;
    }

    public void run()
    {
        int i = 0;

        do
        {
            i = m.cálculos(getName()); // adquirir datos
        }
        while (i < m.tamaño);
    }
}
```

Para lanzar los hilos que adquirirán los datos, implementaremos una aplicación como la siguiente:

```
public class Test
{
    public static void main(String[] args)
    {
        CDatos datos = new CDatos(10);
        CAdquirirDatos adquirirDatos_0 = new CAdquirirDatos(datos);

        adquirirDatos_0.start();
    }
}
```

En la aplicación anterior observamos que **main** crea el objeto *datos* donde el hilo *AdquirirDatos_0* almacenará los datos. Si ejecuta esta aplicación el resultado será el siguiente:

```
Thread-0 tomó la muestra 0
Thread-0 tomó la muestra 1
Thread-0 tomó la muestra 2
Thread-0 tomó la muestra 3
Thread-0 tomó la muestra 4
Thread-0 tomó la muestra 5
Thread-0 tomó la muestra 6
Thread-0 tomó la muestra 7
```

```
Thread-0 tomó la muestra 8
Thread-0 tomó la muestra 9
```

Observando los resultados vemos que todo se ha desarrollado normalmente. Modifiquemos la aplicación *Test* para que ahora utilice dos hilos en lugar de uno, para adquirir los datos:

```
public class Test
{
    public static void main(String[] args)
    {
        CDatos datos = new CDatos(10);

        CADquirirDatos adquirirDatos_0 = new CADquirirDatos(datos);
        CADquirirDatos adquirirDatos_1 = new CADquirirDatos(datos);

        adquirirDatos_0.start();
        adquirirDatos_1.start();
    }
}
```

Ahora el método **main** de la aplicación *Test* lanza dos hilos: *adquirirDatos_0* y *adquirirDatos_1*. Cuando se lanza un hilo, el retorno al proceso padre es inmediato. Por eso podemos suponer que la ejecución del hilo *AdquirirDatos_1* se inicia paralelamente a la de *adquirirDatos_0*. Si ahora ejecutamos la aplicación, el resultado obtenido será similar al siguiente:

```
Thread-0 tomó la muestra 0
Thread-0 tomó la muestra 1
Thread-0 tomó la muestra 2
Thread-1 tomó la muestra 0
Thread-1 tomó la muestra 4
Thread-0 tomó la muestra 4
Thread-1 tomó la muestra 5
Thread-0 tomó la muestra 6
Thread-1 tomó la muestra 7
Thread-0 tomó la muestra 9
Thread-1 tomó la muestra 9
java.lang.ArrayIndexOutOfBoundsException: 10
    at CDatos.asignar(CDatos.java:21)
    at CDatos.cálculos(CDatos.java:29)
    at CADquirirDatos.run(CADquirirDatos.java, Compiled Code)
```

Analicemos los resultados. Cuando se ejecutó sólo un hilo, la matriz se llenó totalmente sin problemas; esto es, no faltaron muestras, tampoco se perdieron por realizar almacenamientos consecutivos en el mismo elemento y no hubo accesos a elementos fuera de los límites establecidos (el número de muestra coincide con el

índice del elemento de la matriz donde está almacenada). En cambio, al ejecutarse los dos hilos concurrentemente, sí se han dado esos problemas.

En sistemas que soporten la planificación por *cuantos*, un hilo en ejecución puede ser interrumpido después de cualquier línea del método siguiente; por ejemplo, supongamos según el código siguiente que uno de los hilos se interrumpe después de la línea 6; no se incrementó *ind*. Si esto ocurre, cuando se ejecute el otro hilo, almacenará la muestra adquirida en el último elemento utilizado; y si suponemos que este hilo es interrumpido después de la línea 7, se incrementa el índice, cuando se ejecute de nuevo el hilo que se interrumpió en la línea 6, volverá a incrementar el índice dejando un elemento vacío.

```

1. public int cálculos(String hilo)
2. {
3.   if (ind >= tamaño) return tamaño;
4.   double x = Math.random();
5.   System.out.println(hilo + " muestra " + ind);
6.   asignar(x, ind);
7.   ind++;
8.   return ind;
9. }
```

Lógicamente los problemas expuestos aparecen porque dos hilos están accediendo a un mismo objeto de datos sin ningún sincronismo. Por lo tanto, la forma de evitar los problemas planteados es que cuando un hilo esté accediendo a ese objeto de datos, no pueda hacerlo el otro y viceversa. Esta sección de código que en un instante determinado tiene que acceder exclusivamente a un objeto de datos compartido, recibe el nombre de *sección crítica*.

Crear una sección crítica

En Java, cada “objeto” tiene un *monitor* (también llamado cerrojo -- *lock*). En un instante determinado, ese monitor es controlado, como mucho, por un solo hilo. El monitor controla el acceso al código sincronizado del objeto; en otras palabras, a la *sección crítica*.

Y ¿cómo se crea una *sección crítica*? La forma más sencilla de crear una *sección crítica* es agrupando el código definido como crítico en un método declarado **synchronized** (sincronizado).

En el ejemplo anterior, la sección de código crítica es el método *cálculos*. Esto quiere decir que un hilo no debe acceder a *cálculos* cuando otro hilo lo está ejecutando, para lo cual, el método *cálculos* de la clase *CDatos* debe ser declarado **synchronized**:

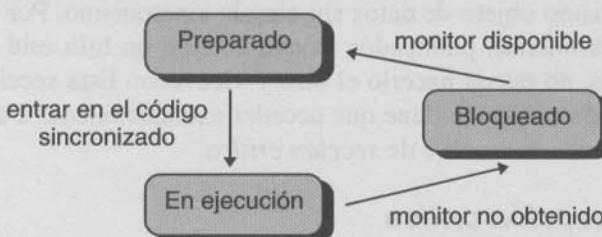
```

public class CDatos
{
    // ...
    public synchronized int cálculos(String hilo)
    {
        if (ind >= tamaño) return tamaño;
        double x = Math.random();
        System.out.println(hilo + " tomó la muestra " + ind);
        asignar(x, ind);
        ind++;
        return ind;
    }
}

```

Si ahora ejecuta de nuevo la aplicación *Test* comprobará que todo funciona como esperábamos.

Un hilo que quiera ejecutar el código sincronizado de un objeto debe primero intentar adquirir el control del monitor de ese objeto. Si el monitor está disponible, esto es, si no está controlado por otro hilo, entonces lo adquirirá y ejecutará el código sincronizado y cuando finalice liberará el monitor. En cambio, si el monitor está controlado por otro hilo, entonces el hilo que lo intentó se bloqueará y sólo retornará al estado *preparado* cuando el monitor esté disponible.



Las secciones de código sincronizadas, llamadas secciones críticas, denotan que el acceso a ellas es crítico para el éxito de la ejecución de los hilos del programa. Por ello, en ocasiones, nos referimos a las secciones críticas como *operaciones atómicas*, significando que ellas representan para cualquier hilo una operación que debe ejecutarse de una sola vez.

Una sección crítica puede ser también un bloque de código que se ejecuta sobre un determinado objeto. En este caso, la sección crítica se delimita así:

```

synchronized (objeto)
{
    // Código que se ejecuta sobre objeto
}

```

Si aplicamos esta segunda técnica sobre el ejemplo anterior, podemos eliminar el método *cálculos* de la clase *CDatos* y reescribir el método **run** de la clase *CAdquirirDatos* así:

```
public class CAdquirirDatos extends Thread
{
    private CDatos m; // objeto para almacenar los datos

    public CAdquirirDatos(CDatos mdatos) // constructor
    {
        m = mdatos;
    }

    public void run()
    {
        double x;
        do
        {
            synchronized (m)
            {
                if (m.ind >= m.tamaño) return;
                x = Math.random();
                System.out.println(getName() + " tomó la muestra " + m.ind);
                m.asignar(x, m.ind);
                m.ind++;
            }
        } while (m.ind < m.tamaño);
    }
}
```

Observe que el código anterior exige que el atributo *ind* de *CDatos* sea público. En la versión anterior era privado.

Lo que no se debe hacer es lo que se muestra a continuación, ya que si el bucle **while** pertenece a la sección crítica, el planificador no podrá bloquear el hilo hasta que no termine de ejecutarse y por lo tanto, no podrá asignar tiempo de UCP al otro hilo. Cuando el bucle **while** finalice, la matriz ya estará llena, lo que supone que el bucle **while** para el otro hilo nunca se ejecutará.

```
synchronized (m)
{
    do
    {
        if (m.ind >= m.tamaño) return;
        x = Math.random();
        System.out.println(getName() + " tomó la muestra " + m.ind);
        m.asignar(x, m.ind);
```

```

        m.ind++;
    }
    while (m.ind < m.tamaño);
}

```

En general es mejor aplicar la sincronización a nivel del método que a un bloque de código. La primera técnica facilita más el diseño orientado a objetos y proporciona un código más fácil de interpretar y por lo tanto, más fácil de depurar y de mantener.

Monitor reentrant

Supongamos que en alguna ocasión necesitamos que un método sincronizado tenga que llamar a otro método también sincronizado de la misma clase. Por ejemplo, para facilitar la comprensión de lo que se trata de explicar, vamos a suponer que el método *asignar* también está sincronizado:

```

public class CDatos
{
    // ...

    public synchronized void asignar(double x, int i)
    {
        dato[i] = x;
    }

    public synchronized int cálculos(String hilo)
    {
        if (ind >= tamaño) return tamaño;
        double x = Math.random();
        System.out.println(hilo + " tomó la muestra " + ind);
        asignar(x, ind);
        ind++;
        return ind;
    }
}

```

La clase *CDatos* contiene ahora dos métodos sincronizados: *asignar* y *cálculos*. El segundo llama al primero. Cuando un hilo trata de ejecutar el método *cálculos* primero toma el control del monitor del objeto *CDatos*. A continuación ejecuta este método, el cual llama al método *asignar*. Como *asignar* está también sincronizado, el hilo intenta adquirir otra vez el control del monitor del objeto *CDatos*. Parece lógico que el hilo debe bloquearse asimismo puesto que trata de adquirir un monitor que él mismo debe ceder, cosa que no puede hacer hasta que no finalice la ejecución de *cálculos*. En cambio no sucede así ¿por qué?

La máquina Java permite a un hilo volver a tomar el control de un monitor del que ya lo tiene, porque los monitores Java son reentrantes. Esto sólo funcionará en sistemas que soporten monitores reentrantes.

Utilizar wait y notify

Hemos visto que las secciones críticas son muy fáciles de utilizar, pero sólo se pueden emplear para sincronizar hilos involucrados en una única tarea; en el ejemplo anterior la tarea era única: almacenar datos en una matriz. Los métodos **wait** y **notify** proporcionan una alternativa más para compartir un objeto, pero con la diferencia de que permiten sincronizar hilos involucrados en tareas distintas, una dependiente de la otra. Piense, por ejemplo, en un sistema que cada vez que genera un mensaje lo encapsula en un objeto *CMensaje* con el fin de manipularlo. En este caso, las tareas involucradas sobre el objeto *CMensaje* son: una, almacenar el mensaje generado y otra, obtener el mensaje almacenado para mostrarlo. Claramente se ve que una tarea depende de la otra; evidentemente, un mensaje no puede ser mostrado si antes no se ha producido.

Supongamos la clase *CMensaje* según se muestra a continuación:

```
public class CMensaje
{
    private String textoMensaje;
    private int númeroMensaje;

    public synchronized void almacenar(int nmsj)
    {
        númeroMensaje = nmsj;
        // Suponer operaciones para buscar el mensaje en una tabla
        // de mensajes; resultado:
        textoMensaje = "mensaje";
    }

    public synchronized String obtener()
    {
        // Componer el mensaje bajo un determinado formato
        String mensaje;
        mensaje = textoMensaje + " #" + númeroMensaje;
        return mensaje;
    }
}
```

Se ha considerado que los métodos *almacenar* y *obtener* son *operaciones atómicas*, significando que para cualquier hilo deben ejecutarse de una sola vez; dicho de otra forma, se han definido como secciones críticas.

En el ejemplo expuesto, estamos pensando en un sistema que tendrá un productor de mensajes (mensajes de aviso, de error, etc.) para generar y almacenar los mensajes producidos (sólo se recuerda el último mensaje) y un consumidor de mensajes que mostrará al usuario el texto de cada mensaje que se produzca. Tanto el productor como el consumidor serán hilos que se suponen están alertas para desempeñar su función cuando sea requerida.

```
public class Productor extends Thread
{
    private CMensaje mensaje; // último mensaje producido
    // Cuando se produce un mensaje, el productor
    // almacena el texto en su miembro "mensaje"
}

public class Consumidor extends Thread
{
    private CMensaje mensaje; // mensaje a mostrar
    // Cuando se ha producido un mensaje, el consumidor
    // lo obtiene de su miembro "mensaje" y lo muestra
}
```

Completemos el código del productor. La clase *Productor* tiene un constructor que inicia el atributo *mensaje* con el objeto *CMensaje* pasado como argumento; este mismo objeto será el que utilice el consumidor para mostrar el último mensaje producido. Como se puede observar, ésta es una forma sencilla de hacer que dos o más hilos comparten datos. Asimismo, sobreescribe el método **run** para almacenar el mensaje que se produzca en el objeto *CMensaje*; este método simula que cada *msegs* milisegundos, valor generado aleatoriamente para cada mensaje, se produce el mensaje de número *númeroMsj*, valor generado también aleatoriamente. El hilo permanece dormido y despierta cada vez que se produce un mensaje. Según lo expuesto, una aproximación a la implementación de esta clase puede ser la siguiente:

```
public class Productor extends Thread
{
    private CMensaje mensaje; // último mensaje producido

    public Productor(CMensaje c) // constructor
    {
        mensaje = c;
    }

    public void run()
    {
        int númeroMsj; // número de mensaje
        while (true)
        {
```

```

númeroMsj = (int)(Math.random() * 100);
mensaje.almacenar(númeroMsj); // almacena el mensaje
System.out.println("Productor " + getName() +
                     " almacena el mensaje #" + númeroMsj);
try
{
    int mseggs = (int)(Math.random() * 100);
    // Poner a dormir el hilo hasta que se produzca el
    // siguiente mensaje.
    sleep(mseggs);
}
catch (InterruptedException e) { }
}

```

Completemos a continuación el código del consumidor. La clase *Consumidor* tiene un constructor que inicia el atributo *mensaje* con el objeto *CMensaje* que comparte con el productor. Asimismo, sobreescribe el método **run** para obtener el mensaje almacenado en el objeto *mensaje* y mostrarlo. Según esto, la implementación de esta clase puede ser la siguiente:

```
public class Consumidor extends Thread
{
    private CMensaje mensaje;           // mensaje a mostrar

    public Consumidor(CMensaje c) // constructor
    {
        mensaje = c;
    }

    public void run()
    {
        String msj;

        while (true)
        {
            msj = mensaje.obtener(); // obtiene el último mensaje
            System.out.println("Consumidor " + getName() +
                " obtuvo: " + msj);
        }
    }
}
```

Una aplicación que lance los hilos productor y consumidor y muestre los resultados que producen puede ser la siguiente:

```
public class Test
```

```

public static void main(String[] args)
{
    CMensaje mensaje = new CMensaje();
    Productor productor1 = new Productor(mensaje);
    Consumidor consumidor1 = new Consumidor(mensaje);

    productor1.start();
    consumidor1.start();
}
}

```

Cuando ejecute la aplicación anterior, tenga presente que los hilos productor y consumidor trabajarán indefinidamente. Por lo tanto, para detener la ejecución tendrá que pulsar las teclas *Ctrl+C*. En lugar de esto, podríamos haber utilizado la técnica mostrada en el apartado “Finalizar un hilo”. No lo hemos hecho para no complicar el código y centrarnos en el tema de sincronización. Una vez que haya ejecutado la aplicación, observará resultados análogos a los siguientes:

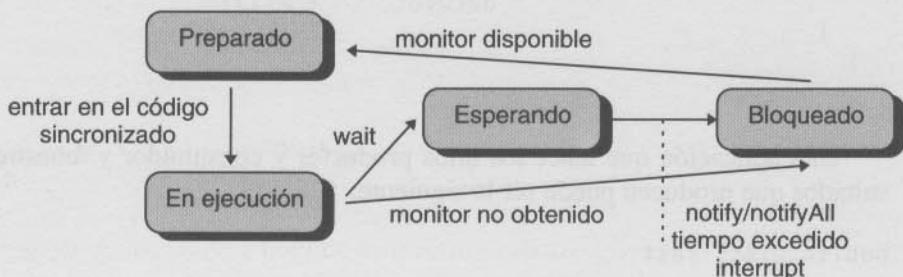
```

Consumidor Thread-1 obtuvo: null #0
Productor Thread-0 almacena: mensaje #15
Consumidor Thread-1 obtuvo: mensaje #15
Productor Thread-0 almacena: mensaje #19
Consumidor Thread-1 obtuvo: mensaje #19
Consumidor Thread-1 obtuvo: mensaje #19

```

Un análisis sencillo nos conduce a la conclusión de que independientemente de que los métodos se hayan definido como secciones críticas, no existe una sincronización entre el productor y el consumidor. El consumidor muestra el último mensaje producido cada vez que el planificador le asigna tiempo de UCP, en lugar de hacerlo única y exclusivamente cada vez que se produzca un mensaje.

Para conseguir la sincronización deseada, el monitor asociado con el objeto *CMensaje* tiene que auxiliarse de los métodos **wait** y **notify**. La siguiente figura muestra las transiciones de estados cuando intervienen estos métodos:



Dijimos que en Java, cada “objeto” tiene un *monitor*. El monitor controla el acceso al código sincronizado del objeto; en otras palabras, a la *sección crítica*. Y según hemos visto, un objeto *CMensaje* presenta dos secciones críticas. Pues bien, el método **notify** despierta sólo un hilo de los que estén esperando por ese monitor. Esto es, si hay varios hilos esperando, se elige uno arbitrariamente. El hilo despertado competirá de la manera habitual con el resto de los hilos que estén en el estado *preparado*, por adquirir la UCP. Según lo expuesto, el método **notify** sólo puede ser llamado por un hilo que haya adquirido el control del monitor.

Un hilo se pone a esperar por el monitor de un determinado objeto invocando al método **wait**. Además, el hilo cede el control del monitor.

```
void wait([milisegundos[, nanosegundos]])
```

El método **wait** envía al hilo actualmente en ejecución al estado de espera, hasta que otro hilo, el que tiene el control del monitor, invoque al método **notify**, **notifyAll** o **interrupt**, o bien hasta que transcurra el tiempo especificado. Cuando el hilo se pone a dormir, cede el control sólo del monitor que controla el acceso al código sincronizado del objeto que lo ha invocado, lo que permitirá a otro hilo que esté esperando por él, adquirirlo; esto es, cualquier otro objeto actualmente controlado por el hilo que se pone a dormir permanecerá bloqueado mientras éste esté dormido.

Precisamente, una de las diferencias entre **sleep** y **wait** es que el primero, cuando es llamado, no cede el control del monitor, mientras que el segundo sí.

El método **notifyAll**, a diferencia de **notify**, despierta todos los hilos que están esperando por el monitor que controla el acceso al código sincronizado de un objeto. Igualmente, los hilos despertados competirán de la manera habitual por adquirir la UCP con el resto de los hilos que estén en el estado *preparado*.

Evidentemente, el método **notify** es más rápido que **notifyAll**, pero su forma de proceder nos puede conducir a situaciones no deseadas cuando hay varios hilos esperando en el mismo objeto. En este caso, rara vez se utiliza, y por seguridad se sugiere utilizar **notifyAll**.

Lo anteriormente expuesto conduce a la conclusión de que los métodos **wait**, **notify**, **notifyAll** e **interrupt**, deben ser invocados desde código sincronizado.

Aplicando la teoría expuesta vamos a continuación a sincronizar los hilos productor y consumidor del ejemplo que estamos desarrollando. Para ello, hay que tener presente que no se puede mostrar un mensaje que aún no se ha generado (por supuesto, los mensajes se muestran una sola vez) y no se puede almacenar un mensaje, si aún no se ha mostrado el último generado.

Entonces, añadiremos a la clase *CMensaje* un atributo *disponible* de tipo **boolean**, que valga **false** cuando no haya ningún mensaje que mostrar, y **true** en caso contrario.

```
public class CMensaje
{
    private String textoMensaje;
    private int númeroMensaje;
    private boolean disponible = false;

    // ...
}
```

Ahora, cuando el hilo productor adquiera el control del monitor del objeto *CMensaje* y ejecute el método sincronizado *almacenar*, lo primero que hará será interrogar el atributo *disponible*. Si su valor es **true**, el hilo se pondrá a dormir hasta que se muestre el último mensaje producido y cede el control del monitor, y si vale **false**, almacena el nuevo mensaje, cambia el atributo *disponible* a **true**, e invoca a **notifyAll** para despertar a todos los hilos que estén esperando por este monitor.

```
public synchronized void almacenar(int nmsj)
{
    while (disponible == true)
    {
        // El último mensaje aún no ha sido mostrado
        try
        {
            wait(); // el hilo se pone a dormir y cede el monitor
        }
        catch (InterruptedException e) { }

        númeroMensaje = nmsj;
        // Suponer operaciones para buscar el mensaje en una tabla
        // de mensajes; resultado:
        textoMensaje = "mensaje";
        disponible = true;
        notifyAll();
    }
}
```

Asimismo, cuando el hilo consumidor adquiera el control del monitor del objeto *CMensaje* y ejecute el método sincronizado *obtener*, lo primero que hará será interrogar el atributo *disponible*. Si su valor es **false**, el hilo esperará hasta que haya un mensaje cediendo el control del monitor, y si su valor es **true**, cambia el atributo *disponible* a **false**, invoca a **notifyAll** para despertar a todos los hilos que estén esperando por este monitor y retorna el mensaje.

```

public synchronized String obtener()
{
    while (disponible == false)
    {
        // No hay mensaje
        try
        {
            wait(); // el hilo se pone a dormir y cede el monitor
        }
        catch (InterruptedException e) { }
    }
    disponible = false;
    notifyAll();
    // Componer el mensaje bajo un determinado formato
    String mensaje;
    mensaje = textoMensaje + " #" + númeroMensaje;
    return mensaje;
}

```

Una vez modificados los métodos *almacenar* y *obtener* de la clase *CMensaje*, ejecute de nuevo la aplicación *Test*. Observará que ahora los resultados sí son los esperados:

```

Productor Thread-0 almacena: mensaje #74
Consumidor Thread-1 obtuvo:   mensaje #74
Productor Thread-0 almacena: mensaje #17
Consumidor Thread-1 obtuvo:   mensaje #17
Productor Thread-0 almacena: mensaje #85
Consumidor Thread-1 obtuvo:   mensaje #85
Productor Thread-0 almacena: mensaje #3
Consumidor Thread-1 obtuvo:   mensaje #3
Productor Thread-0 almacena: mensaje #91
Consumidor Thread-1 obtuvo:   mensaje #91

```

¿Por qué los métodos *almacenar* y *obtener* utilizan un bucle?

Antes de proceder a la explicación, añada más consumidores y observar los resultados. Por ejemplo:

```

public class Test
{
    public static void main(String[] args)
    {
        CMensaje mensaje = new CMensaje();

        Productor productor1 = new Productor(mensaje);
        Consumidor consumidor1 = new Consumidor(mensaje);
        Consumidor consumidor2 = new Consumidor(mensaje);
    }
}

```

```

        productor1.start();
        consumidor1.start();
        consumidor2.start();
    }
}

```

A continuación edite los métodos *almacenar* y *obtener*, y cambie las sentencias **while** por **if**:

```

1. if (disponible == false) // antes: while (disponible == false)
2. {
3.     try
4.     {
5.         wait(); // el hilo se pone a dormir y cede el monitor
6.     }
7.     catch (InterruptedException e) { }
8. }
9. disponible = false;
10.notifyAll();
11.String mensaje;
12.mensaje = textoMensaje + "#" + númeroMensaje;
13.return mensaje;

```

Compile y ejecute de nuevo la aplicación. Compare los resultados con los obtenidos anteriormente ¿Qué ha ocurrido?

```

Productor Thread-0 almacena: mensaje #14
Consumidor Thread-1 obtuvo:   mensaje #14
Consumidor Thread-2 obtuvo:   mensaje #14

```

El peligro de utilizar una sentencia **if** en lugar de **while** es que algunas veces el hilo que adquiere el control del monitor, *Thread-2*, ejecuta la línea 1 y suponiendo que la condición es cierta se pone a esperar, además de ceder el monitor. Más tarde, otro hilo adquiere el monitor, *Thread-1*, suponiendo que la condición es falsa ejecuta la línea 10 y despierta a los hilos que esperan por este monitor. Los hilos en el estado *preparado* compiten por la UCP. Supongamos que el planificador se la adjudica al hilo original, *Thread-2*; éste continuará donde lo dejó, a partir de la línea 5, independientemente del estado del monitor. El resultado es que retorna el mismo mensaje que *Thread-1*. Utilizando una sentencia **while** en lugar de **if**, cuando *Thread-2* continúe donde lo dejó, a partir de la línea 5, volverá a ejecutar la línea 1 y se pondrá de nuevo a esperar por ser la condición cierta.

Interbloqueo

Anteriormente dijimos que la inanición (*starvation*) ocurre cuando un hilo se queda completamente bloqueado y no puede progresar porque no puede acceder a los

recursos que necesita; si esto ocurre entre dos o más hilos porque esperan por una condición recíproca que nunca puede ser satisfecha, estamos en un caso de *interbloqueo* (*deadlock*; algunos autores prefieren denominarlo *abrazo mortal*). Por ejemplo dos hilos necesitan imprimir un documento almacenado en el disco, para lo que necesitan los recursos disco e impresora. Puesto que los hilos se están ejecutando paralelamente, suponga que uno ya ha adquirido el disco y el otro la impresora. Esto significa que ambos hilos quedarán bloqueados, cada uno de ellos esperando por el recurso que tiene el otro.

Para la mayoría de los programadores Java, la mejor de evitar el interbloqueo es prevenirlo, mejor que probar y detectarlo. En cualquier caso, cualquiera de las técnicas existentes para manejar los interbloqueos se sale fuera del objetivo de este capítulo.

GRUPO DE HILOS

Cada hilo Java es un miembro de un *grupo de hilos*. Este grupo puede ser el predefinido por Java o uno especificado explícitamente. Los grupos de hilos proporcionan un mecanismo para agrupar varios hilos en un único objeto con el fin de poder manipularlos todos de una vez; por ejemplo, poder interrumpir un grupo de hilos invocando una sola vez al método **interrupt**. A su vez, un grupo de hilos también puede pertenecer a otro grupo, formando una estructura en árbol. Desde el punto de vista de esta estructura, un hilo sólo tiene acceso a la información acerca de su grupo, no a la de su grupo padre o de cualquier otro grupo.

Java proporciona soporte para trabajar con grupos de hilos a través de la clase **ThreadGroup** del paquete **lang**.

Grupo predefinido

Cuando creamos un hilo sin especificar su grupo en el constructor, Java lo coloca en el mismo grupo (*grupo actual*) del hilo bajo el cual se crea (*hilo actual*).

Por ejemplo, la siguiente aplicación obtiene una referencia al *grupo actual* (grupo predefinido) al cual pertenece el *hilo actual* (en este caso el hilo primario) y la almacena en *consumidores*.

Después, cada hilo consumidor que es creado es añadido al grupo actual que hemos denominado *consumidores*.

Finalmente, más adelante, se envía al grupo actual el mensaje **list** con el objetivo de escribir información acerca del grupo de hilos. Otros métodos puede verlos en la documentación proporcionada con el JDK.

```
public class Test
{
    public static void main(String[] args)
    {
        ThreadGroup consumidores =
            Thread.currentThread().getThreadGroup();
        CMensaje mensaje = new CMensaje();
        Productor productor1 = new Productor(mensaje);
        Consumidor consumidor1 = new Consumidor(mensaje, consumidores,
                                                "consumidor1");
        Consumidor consumidor2 = new Consumidor(mensaje, consumidores,
                                                "consumidor2");
        consumidores.list();

        // ...
    }
}
```

¿Cómo se añaden los hilos a un grupo? Pues utilizando alguno de los constructores que la clase **Thread** proporciona para ello. Por ejemplo:

Thread(ThreadGroup grupo, String nombreHilo)

Siguiendo con el ejemplo anterior, vemos que cuando se invocó al constructor *Consumidor* se pasaron tres argumentos: un objeto *CMensaje*, el grupo de hilos y el nombre del hilo que se desea añadir al grupo. Según esto, el constructor de esta clase será como se indica a continuación:

```
public class Consumidor extends Thread
{
    private CMensaje mensaje;           // mensaje a mostrar

    public Consumidor(CMensaje msj, ThreadGroup grupo, String nombre)
    {
        super(grupo, nombre);
        mensaje = msj;
    }

    public void run()
    {
        // ...
    }
}
```

Se puede observar que el constructor de la clase *Consumidor* invoca al constructor de su clase base (**Thread**) pasándole como argumento el grupo al cual se quiere añadir el hilo, el cual está referenciado por **this**, y el nombre del hilo.

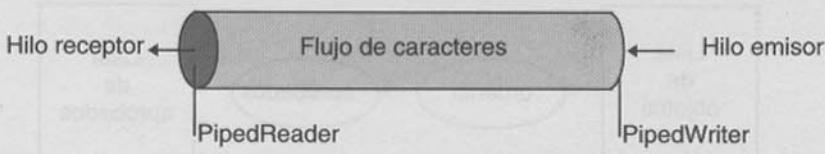
Grupo explícito

Para añadir un hilo a un determinado grupo primero crearemos el grupo y después procederemos de la misma forma explicada en el apartado anterior. Por ejemplo, si en el ejemplo anterior en lugar de utilizar el grupo predefinido por Java quisieramos definir explícitamente un grupo referenciado por la variable *consumidores* y denominado también *consumidores*, la primera línea del método **main** la sustituiríamos por la sombreada en el código mostrado a continuación:

```
public class Test
{
    public static void main(String[] args)
    {
        ThreadGroup consumidores = new ThreadGroup("consumidores");
        // ...
    }
}
```

TUBERÍAS

Básicamente una tubería es utilizada para canalizar la salida de un hilo (puede ser el hilo principal de un programa en ejecución) hacia la entrada de otro. De esta forma los hilos pueden compartir datos sin tener que recurrir a otros elementos como, por ejemplo, ficheros temporales o matrices.



Java proporciona las clases **PipedReader** y **PipedWriter** (y sus homólogas para bytes, **PipedInputStream** y **PipedOutputStream**) para trabajar con tuberías a través de las cuales circularán flujos de caracteres. La primera representa el extremo de la tubería del cual un hilo obtiene los datos y la segunda el extremo de la tubería por el cual un hilo envía los datos al otro. Por lo tanto, estas clases trabajan conjuntamente para proporcionar un flujo de datos a través de una tubería de forma muy similar a como una tubería real proporciona un flujo de agua; en ésta,

si se cerrara un extremo se interrumpiría el flujo. Esto mismo ocurre con los flujos que denominamos tuberías.

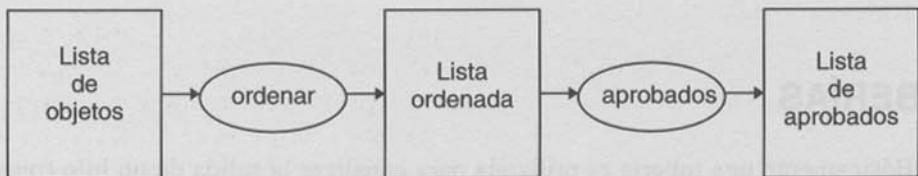
Para crear la estructura de la figura anterior, primero crearíamos un extremo de la tubería (extremo sobre el que trabajará el emisor) y después el otro conectado al anterior para formar la tubería (extremo sobre el que trabajará el receptor). El código necesario para realizar lo expuesto es el siguiente:

```
PipedWriter emisor = new PipedWriter();
PipedReader receptor = new PipedReader(emisor);
```

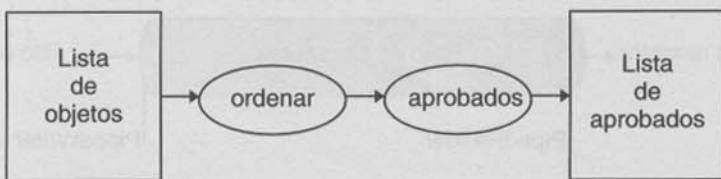
o bien:

```
PipedReader receptor = new PipedReader();
PipedWriter emisor = new PipedWriter(receptor);
```

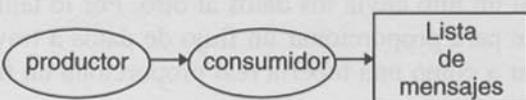
Por ejemplo, pensemos en una lista de objetos, relacionados con alumnos que cursan una determinada asignatura, que deseamos ordenar para después obtener una lista de los aprobados. Sin tuberías, el programa tendría que almacenar los resultados entre cada paso en algún lugar, por ejemplo, en matrices:



Con tuberías, la salida de un proceso se conecta directamente a la entrada del siguiente, según muestra la figura siguiente:



Dejamos este problema para que lo resuelva el lector. Nosotros vamos a resolver uno más breve que muestre simplemente cómo se utilizan las tuberías. Un hilo productor produce mensajes que pasa a través de una tubería a otro hilo consumidor para que los muestre en pantalla. La figura siguiente resume lo expuesto:



En primer lugar vamos a mostrar la aplicación que lanzará los hilos productor y consumidor:

```
import java.io.*;
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            PipedWriter emisor = new PipedWriter();
            PipedReader receptor = new PipedReader(emisor);

            Productor productor1 = new Productor(emisor);
            Consumidor consumidor1 = new Consumidor(receptor);

            productor1.start();
            consumidor1.start();
        }
        catch (IOException ignorada) {}
    }
}
```

Se puede observar que el método **main** crea una tubería *emisor-receptor*. A continuación crea el hilo *productor1* y le pasa como argumento el extremo de la tubería por el cual debe de enviar los mensajes al hilo consumidor. Después crea el hilo *consumidor1* y le pasa como argumento el extremo de la tubería por el cual debe obtener los mensajes enviados por el hilo productor. Finalmente, lanza los dos hilos para su ejecución.

Mostramos a continuación la clase correspondiente al hilo productor. Esta clase tiene un atributo *emisor*, que referenciará el extremo de la tubería por lo que se enviarán los mensajes al hilo consumidor. Este atributo será establecido por el constructor de la clase.

Para enviar mensajes al consumidor, el método **run** del productor crea un flujo (*flujoS*) de la clase **PrintWriter** hacia el extremo *emisor*. Este flujo permitirá utilizar el método **println**, que **PipedWriter** no tiene, para enviar los mensajes por la tubería. La generación de los mensajes se simula igual que hicimos en la versión de la aplicación productor consumidor anterior. En este caso, por tratarse de una tubería, los mensajes producidos son enviados por la misma y puestos en cola mientras el consumidor los va recuperando.

```
import java.io.*;
public class Productor extends Thread
{
    private PipedWriter emisor = null;
```

```

private PrintWriter flujoS = null;

public Productor(PipedWriter em) // constructor
{
    emisor = em;
    flujoS = new PrintWriter(emisor);
}

public void run()
{
    while (true)
    {
        almacenarMensaje();
        try
        {
            int mseg = (int)(Math.random() * 100);
            // Poner a dormir el hilo hasta que se produzca el
            // siguiente mensaje.
            sleep(mseg);
        }
        catch (InterruptedException e) { }
    }
}

public synchronized void almacenarMensaje()
{
    int numeroMsj;      // número de mensaje
    String textoMensaje; // texto mensaje

    numeroMsj = (int)(Math.random() * 100);
    // Suponer operaciones para buscar el mensaje en una tabla
    // de mensajes; resultado:
    textoMensaje = "mensaje #" + numeroMsj;
    flujoS.println(textoMensaje); // enviar mensaje por la tubería
    System.out.println("Productor " + getName() +
                       " almacena: " + textoMensaje);
}
}

protected void finalize() throws IOException
{
    if (flujoS != null) { flujoS.close(); flujoS = null; }
    if (emisor != null) { emisor.close(); emisor = null; }
}
}

```

Finalmente, mostramos la clase correspondiente al hilo consumidor. Esta clase tiene un atributo *receptor*, que referenciará el extremo de la tubería desde el cual el hilo consumidor obtendrá los mensajes. Este atributo será establecido por el constructor de la clase.

Para obtener los mensajes, el método **run** del hilo consumidor crea un flujo (*flujoE*) de la clase **BufferedReader** desde el extremo *receptor*. Este flujo permitirá utilizar el método **readLine**, que **PipedReader** no tiene, para obtener los mensajes enviados por el productor. Cuando no haya ningún mensaje, simplemente el hilo que ejecuta el método **readLine** queda bloqueado.

```

import java.io.*;
public class Consumidor extends Thread
{
    private PipedReader receptor = null;
    private BufferedReader flujoE = null;

    public Consumidor(PipedReader re) // constructor
    {
        receptor = re;
        flujoE = new BufferedReader(receptor);
    }

    public void run()
    {
        while (true)
        {
            obtenerMensaje();
        }
    }

    public synchronized void obtenerMensaje()
    {
        String msj = null;
        try
        {
            msj = flujoE.readLine(); // obtener mensaje de la tubería
            System.out.println("Consumidor " + getName() +
                               " obtuvo: " + msj);
        }
        catch (IOException ignorada) {}
    }

    protected void finalize() throws IOException
    {
        if (flujoE != null) { flujoE.close(); flujoE = null; }
        if (receptor != null) { receptor.close(); receptor = null; }
    }
}

```

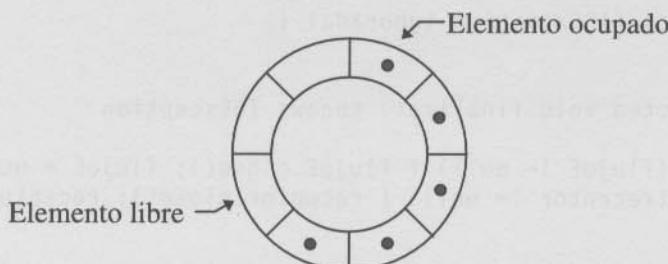
ESPERA ACTIVA Y PASIVA

Cuando se diseña un hilo, lógicamente no sólo pensamos en el trabajo que tiene que desempeñar, sino en cómo su trabajo puede verse afectado por otros hilos. De ahí el estudio de la sincronización de hilos. Pero no es menos importante pensar cómo tiene que comportarse el hilo como unidad individual de ejecución; esto es, poniéndonos en el caso de que durante espacios más o menos cortos, su ejecución no va a ser interferida por otros hilos. Si esto es así, los objetos de sincronización no serán requeridos por otros hilos y puede haber tiempo suficiente para que el hilo finalice su trabajo a la espera de que se den otros eventos que lo requieran de nuevo. En un caso como éste, la espera debe ser pasiva y no activa; es decir, no debe consumir tiempo de UCP.

En general un hilo realiza una espera pasiva poniéndose él mismo a dormir porque cuando está durmiendo, no entra en la planificación del sistema operativo; esto es, el sistema operativo no le asigna tiempo de UCP y, por consiguiente, detiene su ejecución. Tanto **wait** como **sleep** ponen un hilo a dormir; en el primer caso, para despertado hay que invocar a **notify** o **notifyAll**, o bien esperar a que transcurra el tiempo si se especificó, y en el segundo caso despierta cuando transcurra el tiempo especificado.

EJERCICIOS RESUELTOS

- Realizar una aplicación que utilice dos hilos, un productor y un consumidor, trabajando sobre una única matriz de enteros positivos. Esto es, un hilo productor generará enteros que almacenará en una matriz circular y un hilo consumidor obtendrá de esa matriz los enteros generados por el productor. Muchas aplicaciones de la vida ordinaria reproducen este problema. Un ejemplo es el administrador de impresión en un servidor de red; los productores son los usuarios de la red y el consumidor la impresora o impresoras.



La figura anterior muestra un esquema del problema del *productor* y el *consumidor*. Para un correcta sincronización entre los hilos, el productor deberá blo-

quear la matriz sólo mientras se esté insertando un dato y el consumidor lo hará sólo mientras se esté extrayendo. Cuando la matriz esté vacía, el hilo consumidor se pondrá a esperar hasta que haya datos. Asimismo, cuando la matriz esté llena, el hilo productor se pondrá a esperar hasta que haya elementos libres.

La matriz que almacenará los datos será un objeto de la clase *CMatriz*. Esta clase estará formada por los atributos:

<i>m</i>	Matriz de <i>n</i> enteros positivos.
<i>indProd</i>	Índice del elemento donde el productor debe insertar el siguiente elemento. Su valor será: 0, 1, 2, ..., <i>n</i> -1, 0, 1, 2, ...
<i>indCons</i>	Índice del elemento donde el consumidor debe obtener el siguiente elemento. Su valor será: 0, 1, 2, ..., <i>n</i> -1, 0, 1, 2, ...
<i>elementosVacíos</i>	Número de elementos vacíos en un instante determinado.
<i>elementosLlenos</i>	Número de elementos llenos en un instante determinado.

y por los métodos:

<i>almacenar</i>	Almacena un dato en el siguiente elemento vacío.
<i>obtener</i>	Obtiene el siguiente dato aún no extraído.

El código correspondiente a esta clase se muestra a continuación:

```
///////////////////////////////
// Sincronización de hilos: wait y notify.
//
public class CMatriz
{
    private int[] m;
    private int indProd = 0; // índice productor
    private int indCons = 0; // índice consumidor
    private int elementosVacíos, elementosLlenos;

    public CMatriz(int n)
    {
        if (n < 1) n = 10;
        m = new int[n];
        elementosVacíos = m.length;
        elementosLlenos = 0;
    }

    public synchronized void almacenar(int num)
    {
        // Esperar a que haya elementos vacíos
        while (elementosVacíos == 0)
        {
            try
            {
                wait(); // el hilo se pone a dormir y cede el monitor
            }
        }
    }
}
```

```

        }
        catch (InterruptedException e) { }
    }
    elementosVacíos--;
    elementosLlenos++;
    System.out.print("vacíos: " + elementosVacíos + ", llenos: " +
                    elementosLlenos + "\r");
    m[indProd] = num;
    indProd = (indProd + 1) % m.length;
    // Despertar hilos;
    notifyAll();
}

public synchronized int obtener()
{
    // Esperar a que haya elementos llenos
    while (elementosLlenos == 0)
    {
        try
        {
            wait(); // el hilo se pone a dormir y cede el monitor
        }
        catch (InterruptedException e) { }
    }
    elementosVacíos++;
    elementosLlenos--;
    System.out.print("vacíos: " + elementosVacíos + ", llenos: " +
                    elementosLlenos + "\r");
    int num = m[indCons];
    indCons = (indCons + 1) % m.length;
    notifyAll();
    return num;
}
}

```

En el problema del productor y del consumidor los recursos que estos hilos deben adquirir para poder ejecutarse son los elementos vacíos y los elementos llenos de la matriz, respectivamente. Cada uno de estos tipos de recursos los representaremos por sendas variables que actuarán como semáforos: *elementosLlenos* y *elementosVacíos*. Un valor cero equivale a semáforo en *rojo* y un valor distinto de cero a semáforo en *verde*.

elementosLlenos es un semáforo inicialmente en *rojo* para el consumidor (porque no hay ningún elemento lleno, esto es, no se puede *obtener*) que representa los elementos actualmente llenos de la matriz y *elementosVacíos* es un semáforo inicialmente en *verde* para el productor (porque todos los elementos están vacíos, esto es, se puede *almacenar*) que representa los elementos actualmente

vacíos de la matriz. Por lo tanto, el contador de *elementosLlenos* debe valer inicialmente cero y el de *elementosVacíos* debe valer *m.length*.

Cuando un hilo necesita un recurso de un tipo particular, decrementa el contador del semáforo correspondiente, y cuando lo libera lo incrementa. Por ejemplo, cuando el productor quiere *almacenar* un dato necesita el recurso “elementos vacíos”, de tal forma que cada vez que lo adquiere lo decrementa; cuando llegue a cero implica semáforo en rojo indicando que el recurso “elementos vacíos” está ocupado. Lógicamente decrementar *elementosVacíos* implica incrementar *elementosLlenos*.

```
// elementosVacíos es el semáforo para el productor
while !(elementosVacíos == 0)
{
    try
    {
        wait(); // el hilo se pone a dormir y cede el monitor
    }
    catch (InterruptedException e) { }
}
elementosVacíos--;
elementosLlenos++;
m[indProd] = num;
indProd = (indProd + 1) % m.length;
notifyAll();
```

El código anterior, que pertenece al hilo productor, decrementa el contador del semáforo *elementosVacíos*, inserta un dato en el siguiente elemento vacío de la matriz y, lógicamente, incrementa el contador del semáforo *elementosLlenos*. Si el contador de *elementosVacíos* fuera cero, el hilo productor pasaría al estado bloqueado hasta que el hilo consumidor extraiga uno o más datos y, por consiguiente, incremente *elementosVacíos*. Un razonamiento análogo haríamos para el consumidor.

Según los expuesto, el hilo productor básicamente se limitará a llamar al método *almacenar* de la clase *CMatriz*. Esto es:

```
///////////
// Sincronización de hilos. Hilo productor.
//
public class Productor extends Thread
{
    private CMatriz matriz;
    private boolean continuar = true;

    public Productor(CMatriz m) // constructor
    {
```

```

        matriz = m;
    }

    public void run()
    {
        int número; // número producido

        while (continuar)
        {
            número = (int)(Math.random() * 100);
            matriz.almacenar(número); // almacena el número
            //System.out.println("Productor " + getName() +
            //                    " almacena: número " + número);
        }
    }

    public void terminar()
    {
        continuar = false;
    }
}

```

Análogamente, el hilo consumidor básicamente se limitará a llamar al método *obtener* de la clase *CMatriz*. Esto es:

```

// Sincronización de hilos. Hilo consumidor.
//
public class Consumidor extends Thread
{
    private CMatriz matriz;
    private boolean continuar = true;

    public Consumidor(CMatriz m) // constructor
    {
        matriz = m;
    }

    public void run()
    {
        int número;
        while (continuar)
        {
            número = matriz.obtener();
            //System.out.println("Consumidor " + getName() +
            //                    " obtuvo: número " + número);
        }
    }
}

```

```

    public void terminar()
    {
        continuar = false;
    }
}

```

Para probar el comportamiento de ambos hilos puede servir la aplicación siguiente:

```

import java.io.*;
// Sincronización de hilos.
//
public class Test
{
    public static void main(String[] args)
    {
        CMatriz matriz = new CMatriz(10);
        Productor productor1 = new Productor(matriz);
        Consumidor consumidor1 = new Consumidor(matriz);

        System.out.println("Pulse [Entrar] para continuar y");
        System.out.println("vuelva a pulsar [Entrar] para finalizar.");

        InputStreamReader is = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(is);
        try
        {
            br.readLine(); // ejecución detenida hasta pulsar [Entrar]
            // Iniciar la ejecución de los hilos
            productor1.start();
            consumidor1.start();
            br.readLine(); // ejecución detenida hasta pulsar [Entrar]
        }
        catch (IOException e) {}
        // Permitir a los hilos finalizar
        productor1.terminar();
        consumidor1.terminar();
    }
}

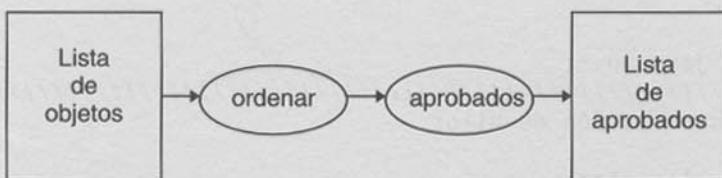
```

EJERCICIOS PROPUESTOS

1. Escribir una aplicación que lance tres hilos que ordenen otras tantas matrices, todas de la misma dimensión, utilizando, el primero el método de ordenación de la burbuja, el segundo el de inserción y el tercero el método *quicksort*. Visualizar

como resultado el nombre de los métodos de ordenación colocados de más rápido a menos rápido.

- Supongamos una lista de objetos, relacionados con alumnos que cursan una determinada asignatura, que deseamos ordenar para después obtener una lista de los aprobados. Utilizando tuberías, podemos plantear la solución del problema según muestra la figura siguiente:



Cada objeto alumno almacenará información relativa al nombre del alumno, al nombre de la asignatura y a la nota.

Para obtener el resultado solicitado, los pasos a seguir básicamente pueden ser los siguientes:

- Crear el fichero con la información de los alumnos ordenada por el *nombre* del alumno.
- Abrir un flujo desde el fichero que permita leer la información del mismo.
- Invocar a un método *ordenar* que reciba como parámetro el flujo abierto en el punto 2 y devuelva una referencia a un objeto **PipedInputStream** (o a su superclase), que se corresponda con el extremo de una tubería en la que un hilo lanzado por este método coloque los alumnos clasificados por la *nota*. Para realizar la ordenación, el hilo cargará la información en una matriz, la ordenará y después la volcará en la tubería.
- Invocar a un método *aprobados* que reciba como argumento el flujo de datos resultante del punto 3 y devuelva una referencia a un objeto **PipedInputStream** (o a su superclase), que se corresponda con el extremo de una tubería en la que otro hilo lanzado por este método coloque los alumnos aprobados.
- Grabar el resultado obtenido en el punto 4 en otro fichero. Después, visualizar el fichero para comprobar el resultado obtenido.

APÉNDICE F

© F.J.Ceballos/RA-MA

ÍNDICE

A

abstracción, 32
abstract, 330
Accesibilidad, 723
acceso aleatorio, 457
acceso secuencial, 429; 479
accesos directos, 699
Ackerman, 586
acos, 115
ActionEvent, 729
ActionListener, 727
actionPerformed, 728
add, 730
addActionListener, 727
addAdjustmentListener, 727
addFocusListener, 728
addItemListener, 728
addKeyListener, 728
addMouseListener, 728
addMouseMotionListener, 728
addWindowListener, 728
AdjustmentListener, 727
administrador de interfaz de usuario, 731
administradores de diseño, 730
 asignar, 731
algoritmo Boyer y Moore, 602
algoritmo de planificación determinista, 649
algoritmos hash, 616
ámbito de una variable, 85
anidar if, 124
anidar while, do, o for, 136
animación, 718
ANSI, 763

añadir un componente a un panel, 730
añadir un elemento a una matriz, 297
aplicación, 11; 63; 264
append, 187
applet, 8; 11; 707, 705
 crear, 705
 parámetros, 711
APPLET, etiqueta, 705
appletviewer, 10
árbol, 542
 binario, 543
 binario de búsqueda, 546
 binario perfectamente equilibrado, 558
 recorrer, 544
archivo, 420
argumentos, 73
 pasar, 219
argumentos en la línea de órdenes, 221
ArithmeticException, 148
arrastrar y colocar, 723
arraycopy, 218
ArrayIndexOutOfBoundsException, 168
Arrays, 233
ASCII, 765
asignación de objetos, 274
asignar un administrador de diseño, 731
asin, 115
ASP, 704
atan, 115
atan2, 115
atómicas, operaciones, 660
atrapar la excepción, 403
atributos, 26; 255
 con el mismo nombre, 341

atributos (continuación)

- de la clase, 78
- iniciar, 255
- static, 289
- AudioClip**, 716
- available**, 114
- awt**, 710; 723

B

barra de direcciones, 695
 barra de estado, mensajes, 718
 biblioteca de clases, 15
 bin, 10
binarySearch, 233
 bit, 4
 bloque, 72
 bloque de finalización, 405
 bloqueado, 637
 boolean, 27; 43; 103
BorderLayout, 731
 borrar los elementos de una lista, 504
 borrar nodo, 553
 borrar un elemento de una lista, 503
BoxLayout, 731
 Boyer y Moore, 602
 break, 131; 146
BufferedInputStream, 98
BufferedReader, 99; 613
BufferedWriter, 613
 burbuja, 592
 buscar nodo, 550
 buscar un elemento en una lista, 504
 buscar un elemento en una matriz, 299
 búsqueda binaria, 233; 601
 búsqueda de cadenas, 602
 búsqueda secuencial, 600
 byte, 4; 41; 103

C

cadenas de caracteres, 175
 concatenar, 46
 leer y escribir, 176
Calendar, 263
 campo, 420
 capacity, 187
 capas de servicios de la red, 689
 carácter siguiente en el flujo de entrada, 427
 caracteres, 38
 \r\n, 112
 disponibles en un flujo, 114

manipular, 173
CardLayout, 731
 cast, 51
 catch, 95; 403
 ceil, 115
CGI, 703
 ciclo de vida, 646
 clase, 25, 253
 abstracta, 92; 330
 anidada, 381
 anónima, 384
 aplicación, 30
Applet, 707
Arrays, 233
 base, 329
BufferedInputStream, 98
BufferedReader, 99; 613
BufferedWriter, 613
Calendar, 263
CArbolBinB, 555
CArbolBinE, 559
Class, 97
Color, 713
 crear, 26
DataInputStream, 442
DataOutputStream, 441
Date, 74; 232
DateFormat, 232
 de un objeto, 97
DecimalFormat, 226
 dentro de un método, 383
 derivada, 329
 derivada de otra ya existente, 429
EOFException, 404
Error, 413
Excepción, 400
FieldPosition, 228
File, 434
FileInputStream, 433
FileOutputStream, 430
FileReader, 439
FileWriter, 438
 final, 268
Font, 713
Format, 226
 genérica, 509
Graphics, 708
GregorianCalendar, 263
HttpServletRequest, 738
InputStream, 93
InputStreamReader, 99
 interna, 382
IOException, 400

- clase (continuación)
 JApplet, 734
 JButton, 727
 JComboBox, 727
 JCheckBox, 727
 JFrame, 724
 JLabel, 727
 JOptionPane, 727
 JRadioButton, 727
 JScrollPane, 727
 JTextArea, 727
 JTextField, 727
 Leer, 106; 415
 LinkedList, 519
 lista lineal simplemente enlazada, 505
 Locale, 228
 Math, 114
 MessageFormat, 233
 Method, 97
 NumberFormat, 226
 Object, 92; 97; 168; 235; 287; 509; 758
 ObjectInputStream, 451
 ObjectOutputStream, 450
 OutOfMemoryError, 413
 OutputStream, 94
 PrintStream, 100
 PrintWriter, 102
 pública, guardar, 67
 PushbackReader, 425
 Random, 241
 Reader, 93
 RuntimeException, 400
 SimpleDateFormat, 231
 String, 27; 181
 StringBuffer, 186
 System, 69
 Thread, 638
 Throwable, 95; 399
 TreeMap, 567
 TreeSet, 566
 Writer, 94
 clases con ficheros, 480
 clases Hash..., 567
 clases para tipos primitivos, 103
 class, 26; 97
 CLASSPATH, 110; 305
 CListaLinealSE, 512; 523; 534
 clone, 168; 218
 close, 425; 432; 458
 códigos de bytes, 7
 cola, 529
 colecciones de objetos, 566
 colector de basura, 83
 Color, 713
 Collection, 566
 comentario, 47
 compareTo, 182
 compareToIgnoreCase, 183
 compilación, 14
 compilador, 6
 compilador JIT, 8
 complejos, 326
 componente *Swing*, 726
 comportamiento, 27
 concat, 182
 concatenar, 46
 constante simbólica, 48
 constructor, 34; 74, 269
 constructor copia, 276
 constructor de la superclase, invocar, 345
 constructor de una subclase, 345
 constructor por omisión, 270; 272
 consulta dinámica, 371
 contador, 174
 contenedores, 729
 de nivel intermedio, 730
 de nivel superior, 730
 continue, 146
 control de acceso a una clase, 265
 control el acceso, 336
 conversión, 51
 de cadenas a números, 191
 explícita en subclases, 359; 511
 implícita en subclases, 357
 copiar una matriz numérica, 217
 correo electrónico, 692
 cos, 115
 CR, 112
 crear un objeto, 73
 crear un paquete, 304
 crear una clase, 26
 crear una nueva excepción, 408
 CRLF, 112
 cuanto, 650
 currentTimeMillis, 629
 char, 42
 Character, 103
 CharArrayReader, 423
 CharArrayWriter, 423
 charAt, 185; 189

D

- DataInput, 442
 DataInputStream, 442
 DataOutput, 441

DataStream, 441
Date, 74; 232
DateFormat, 232
DecimalFormat, 226
declaración, 70
declarar las excepciones, 407
definición de un método, 72
definir una subclase, 338
delete, 188; 436
demonio, 643
depuración, 16
destroy, 709
destructor, 34; 276
destructor en una subclase, 347
dirección de Internet, 691
dirección IP, 691
diseño, administradores, 730
dispositivo de salida, 26
dispositivos estándar, 472
DNS, 690
do ... while, 139
doGet, 739
dominio, 690
doPost, 739
double, 43; 103
doubleValue, 104
drawImage, 715
drawString, 721

E

ejecución, 14; 637
eliminar un elemento de una matriz, 298
else if, 126
e-mail, 692
encapsulación, 77
encapsulamiento, 33
endsWith, 184
enfocado, 728
enlaces, 699
ensamblador, 4
entorno de desarrollo integrado, 11; 16
EOFException, 404
equals, 168; 234; 236; 287
err, 96
Error, 400
escuchadores de eventos, 727
estados de un hilo, 636
estados de un proceso, 634
estructura de una aplicación gráfica, 724
estructura else if, 126
estructura interna, 255
estructuras abstractas de datos, 496

estructuras dinámicas, 495
etiquetas, 147
eventos, 727
excepción
 atrapar, 403
 declarar, 406
 lanzar, 402
 manejar, 401
 utilizar, 413
excepciones, 95; 397
 crear, 408
 explícitas, 400
 implícitas, 400
Excepción, 400
exists, 435
exp, 115
expresiones booleanas, 54
expresión, 51
extcheck, 10
extends, 335; 372
extranet, 689

F

fecha/hora, 263
fichero, 420
 leer líneas de texto, 477
fichero temporal, 471
FieldPosition, 228
File, 434
FileInputStream, 433
FileOutputStream, 430
FileReader, 439
FileWriter, 438
filtros, 441
fill, 234
fin de fichero, 110
final, 48; 268
finalize, 237
finalize en una subclase, 347
finally, 405
float, 42; 103
floatValue, 104
floor, 115
FlowLayout, 731
flujo, 91; 421
 limpiar, 427
flujos
 de bytes, 430
 de caracteres, 437
 de datos, 440
 estándar, 96
flush, 102

foco, 728
 FocusListener, 728
 Font, 713
 for, 142
 Format, 226
 formato, 225
 alineación, 228
 para fechas y horas, 231
 para mensajes, 233
 símbolos, 226
 FTP, 691; 693
 fuentes, 713

G

garbage collector, 279
 gc, 279
 GET, 739
 getBytes, 186
 getClass, 97
 getCodeBase, 715
 getContentPane, 730
 getCrossPlatformLookAndFeelClassName, 731
 getCurrencyInstance, 227
 getDateInstance, 232
 getDocumentBase, 715
 getEndIndex, 228
 getImage, 715
 getMessage, 403
 getMethods, 97
 getName, 97
 getNumberInstance, 227
 getParameter, 712
 getPercentInstance, 227
 getPriority, 651
 getSystemLookAndFeelClassName, 731
 getTime, 242
 getTimeInstance, 232
 Gopher, 691
 gráficos en un documento HTML, 701
 Graphics, 708
 GregorianCalendar, 263
 GridBagLayout, 731
 GridLayout, 731
 grupos de noticias, 693

H

Hanoi, 589
 hash, 616
 Hash..., 567
 herencia, 33; 329; 331

múltiple, 34; 335; 379
 simple, 33; 335
 hilo, 635
 ¿está vivo?, 647
 creado, 641
 crear, 641
 egoísta, 654
 prioridad, 651
 terminar, 644
 hilos
 cooperantes, 655
 espera activa, 678
 espera pasiva, 678
 estados, 636
 independientes, 654
 sincronizar, 655
 hipertexto, 699
 HTML, 695
 etiquetas, 696
 HttpServlet, 738

I

identificadores, 46
 IEEEremainder, 115
 if, 121
 if anidados, 124
 imagen en un applet, 714
 imágenes en un documento HTML, 701
 implementación de una clase, 259
 implements, 374
 import, 69
 in, 96
 indexOf, 185
 iniciador estático, 293
 init, 708; 739
 inorden, 544
 InputStream, 93
 InputStreamReader, 99
 inserción, 595
 insert, 187
 insertar nodo, 551
 insertar un elemento en una lista, 500
 instanceof, 530
 instancia, 25
 int, 41
 Integer, 103
 interfaces, 372
 gráficas, 723
 múltiples, 381
 interfaz, 34; 256; 269
 AudioClip, 716
 DataInput, 442

interfaz (continuación)

 DataOutput, 441
 definición, 371
 para qué sirve, 380
 pública, 77
 Runnable, 639; 641
 Serializable, 449
 SingleThreadModel, 740
 tipo de datos, 378
 utilizar, 374
 vs. clase abstracta, 377

intern, 240

Internet, 688

Internet Explorer, 694

intérprete, 6

interrupt, 667

intranet, 689

intValue, 104

invocar a un método redefinido, 359

IOException, 95; 400

isAlive, 647

ISAPI, 704

isDirectory, 436

isFile, 436

isNaN, 206

ItemListener, 728

J

JApplet, 730; 734

jar, 10

Java, 7; 10

Java 2D, 723

Java Runtime Environment, 649

java.io, 89; 421

java.lang, 69; 89

java.text, 226

java.util, 74; 228

javac, 10

javadoc, 10

javah, 10

javap, 10

javax.servlet, 738

JButton, 727

JComboBox, 727

JCheckBox, 727

jdb, 10

JDialog, 730

JDK, 10; 759

jerarquía de clases, 329; 336; 350

JFC, 723

JFrame, 724

JIT, 8; 11

JLabel, 727

JOptionPane, 727

JRadioButton, 727

jre, 11; 649

JScrollBar, 727

JSP, 737

JSDK, 742

JTextArea, 727

JTextField, 727

K

kernel, 636

KeyListener, 728

L

lang, 69

lanzar una excepción, 402

lastIndexOf, 185

leer, 104

leer líneas de texto, 477

leer una línea de texto, 99; 428

Leer, clase, 415

length, 168; 184; 187

lenguaje máquina, 5

lenguajes de alto nivel, 5

LF, 112

lib, 11

limpiar un flujo, 427

LinkedList, 519

List, 566

lista circular, 522

lista circular doblemente enlazada, 534

lista doblemente enlazada, 533

lista lineal simplemente enlazada, 496

lista lineal simplemente enlazada, 496

lista lineal, recorrer, 504

listas lineales, 496

literal, 43

- de cadena de caracteres, 45
- de un solo carácter, 45

entero, 44

real, 44

Locale, 228

log, 115

long, 41; 103

longitud de una matriz, 168

longValue, 104

loop, 716

LPT1, 472

M

mail, 692
 main, 13; 29; 73; 79
 argumentos, 222
 manejadores de eventos, 727
 Map, 566
 máquina Java, 649
 máquina virtual, 6
 marcos en páginas HTML, 702
 mark, 613
 Math, 114
 matrices, 164
 de objetos, 294
 métodos, 168
 verificar si son iguales, 234
 matriz
 acceder a un elemento, 167
 asignar un valor a todos sus elementos, 234
 asociativa, 172
 buscar un valor, 233
 como valor retornado, 217
 crear, 166
 de cadenas de caracteres, 196
 de longitud 0, 297
 de objetos String, 203
 declarar, 165
 es un objeto, 166
 multidimensional, 191
 numérica multidimensional, 192
 ordenar, 235
 pasar como argumento, 215
 sparse, 213
 max, 115
 MAX_VALUE, 104
 memoria para objetos String, 238
 memoria, asignar y liberar, 74
 mensaje, 24
 mensajes, 75
 mensajes en la barra de estado, 718
 MessageFormat, 233
 Method, 97
 método, 24; 28; 72
 abreviado, 263
 abstracto, 330
 consulta dinámica, 371
 de inserción, 595
 de la burbuja, 592
 de la clase, 78
 de quicksort, 596
 final, 268
 recursivo, 224
 sobrecargado, 262

static, 291
 métodos, 27; 256
 de una subclase, 340
 en línea, 371
 mezcla natural, 606
 miembro de una clase, 30
 miembros del objeto, 77
 miembros heredados, 336
 miembros que son punteros, 279
 milisegundos transcurridos desde el 1 de enero
 de 1970, 242
 min, 115
 MIN_VALUE, 104
 mkdir, 436
 módem, 692
 modificador, 72
 modificadores de acceso, 257
 monitor, 659; 667
 monitores reentrantes, 663
 MouseListener, 728
 MouseMotionListener, 728

N

NaN, 104; 206
 NEGATIVE_INFINITY, 104
 new, 73; 74; 79
 newAudioClip, 717
 newLine, 613
 news, 693
 nivel de protección predeterminado, 77
 nodo de un árbol, 544
 notify, 663; 667
 notifyAll, 667
 null, 43; 111; 203; 498
 NumberFormat, 226
 NumberFormatException, 106
 número racional, 306
 números aleatorios, 240

O

Object, 92; 97; 168; 235; 287; 509; 758
 ObjectInputStream, 451
 ObjectOutputStream, 450
 objeto, 24
 aplicación, 63
 String, crear, 238
 temporal, 309
 objetos, guardar/leer en/de un fichero, 449
 ocultación de datos, 257

operadores, 52
 a nivel de bits, 55
 aritméticos, 52
 condicional, 57
 de asignación, 56
 de relación, 53
 instanceof, 530
 lógicos, 54
 new, 73; 74; 79
 ternario, 57
 unitarios, 55
 ordenación, 591
 ordenar un fichero, 605
 utilizando acceso aleatorio, 614
 out, 13; 26; 96
 OutOfMemoryError, 74; 413
 OutputStream, 94

P

package, 305
 página dinámica, 703
 páginas
 ASP, 704
 JSP, 737
 web, 695
 paint, 708
 palabras clave, 47
 panel de contenido, 730
 panel raíz, 730
 paquete, 67; 303
 awt, 710
 crear, 304
 java.io, 421
 java.text, 226
 java.util, 228
 protección de, 68
 parámetros, 73
 de un applet, 711
 pasados por referencia, 83; 215
 pasados por valor, 83; 215
 parseInt, 104
 pasar argumentos, 82
 PI, 115
 pila, 527
 planificación, 649
 planificador, 637
 planificador de hilos, 641
 plantillas, 509
 play, 716
 polimorfismo, 34; 360
 POO, 23
 POP 2 y 3, 691

POSITIVE_INFINITY, 104
 POST, 740
 postorden, 544
 pow, 115
 predeterminado, 258
 preemptive, 649
 preorden, 544
 preparado, 636
 print, 101
 println, 13; 26; 101
 PrintStream, 26; 100
 PrintWriter, 102
 prioridad de un hilo, 651
 private, 258
 proceso, 633
 proceso ligero, 635
 productor-consumidor, 678
 programa, 4; 634
 programación orientada a objetos, 23
 protección de una clase, 68
 protected, 258
 protocolo, 689
 de transferencia de ficheros, 693
 proyecto, 18
 public, 68; 258
 public, clase, 32
 pública, 67
 PushbackReader, 425

Q

quicksort, 596

R

racional, 306
 raíz de un árbol, 544
 random, 115; 241; 242
 RandomAccessFile, 458
 read, 93; 113; 197; 425
 Reader, 93
 readLine, 99; 111; 203
 readUTF, 463
 ready, 425; 458
 recolector de basura, 75; 279
 recorrer un árbol, 544
 recursión, 585
 recursividad, 224
 recursos, 715
 redefinir miembros de la superclase, 343
 reentrantes, monitores, 663
 referencia, 79

- a un tipo primitivo, 219
final, 268
referencias a subclases, 356
referencias y objetos String, 238
reflexión, 98
registro, 420
repaint, 708; 719
replace, 185; 188
representación interna, 269
reset, 613
resultado, 72
return, 73
reverse, 188
rint, 115
r\n, 112
round, 115
round-robin, 650
run, 640
Runnable, 639; 641
RuntimeException, 400
- S**
- saltar n caracteres en un flujo, 114
sección crítica, 659
secciones críticas, 655
secuencia de escape, 39
seguridad en los applets, 722
sentencia
 break, 146
 compuesta, 72
 continue, 146
 de asignación, 90
 do ... while, 139
 for, 142
 if, 121
 import, 69
 return, 73
 simple, 71
 switch, 129
 while, 133
seriación, 449
Serializable, 449
servidor de nombres, 691
servidor de servlets, 742
servidor Web de Java, 742
servlet, 737
 ejecutar, 743
 estructura, 738
Set, 566
setColor, 713
setCharAt, 189
setFont, 713
- setLayout, 731
setLength, 187
setLookAndFeel, 731
setPriority, 651
short, 41; 103
showStatus, 718
SimpleDateFormat, 231
sin, 115
sincronización de hilos, 655
 exclusión mutua, 663
 secciones críticas, 655
SingleThreadModel, 740
sistema de nombres de dominio, 690
skip, 114
sleep, 667
SMTP, 692
sobrecarga de métodos, 262
sobrecarga del operador +, 309
sonido en un applet, 716
sonido en una aplicación, 717
sort, 235
sqrt, 115
start, 641; 708
startsWith, 184
starvation, 650; 654
static, 48; 78; 289; 290
static iniciador, 293
stop, 708; 716
stream, 421
String, 27; 90; 181
String, constructor, 181
StringBuffer, 186
subclase, 329; 335
subdominio, 690
subproceso, 635
substring, 185; 189
super, 339; 342; 344; 345
superclase, 329
 directa, 354
 indirecta, 354
Swing, 723
switch, 129
synchronized, 659
System, 13; 26; 69
System.err, 96
System.in, 96
System.out, 96
- T**
- tan, 115
TCP/IP, 689
Telnet, 692

this, 266; 344
Thread, 638
throw, 402
Throwable, 95; 399
throws, 407
tiempo de ejecución, 629
time-slice, 650
tipo
 boolean, 43
 byte, 41
 char, 42
 double, 43
 float, 42
 int, 41
 long, 41
 referenciado, 80
 short, 41
 String, 90
tipos primitivos, 40
tipos referenciados, 43
toCharArray, 186
toDegrees, 115
toLowerCase, 184
toRadians, 115
torres de Hanoi, 589
toString, 104; 182; 189; 237
toUpperCase, 184
TreeMap, 567
TreeSet, 566
trim, 184
try, 95; 403
try ... catch, 148

U

UIManager, 731
Unicode, 38; 42
unread, 425
update, 708
URL, 699
URL de la carpeta, 715
USENET, 692; 693
UTF-8, 441
util, 74

V

valueOf, 104; 186
variable, 49
CLASSPATH, 110
iniciar, 50
local, 86
miembro de una clase, 86
void, 28; 72

W

wait, 663; 667
Web, 693
while, 133
while, do, o for anidados, 136
windowClosing, 725
WindowListener, 728
World Wide Web, 692; 693
write, 94; 431; 613
Writer, 94
writeUTF, 463
WWW, 692; 693

Nota del escaneador

Escaneado en abril del 2005 en Toledo (España)

Disculpen la calidad, ya que el libro está muy tocado y es de la biblioteca.

Este libro es muy bueno, es de Javier Ceballos, si te gusta, por favor, compratelo!. Ahora mismo cuesta 42 euros.

Como veis faltan páginas, si, falta la parte de los applets y demás (parte 3 del libro).

Escanear está mal, es ilegal, pero tambien lo es tener windows sin comprarlo
Utiliza linux, y verás lo que verdaderamente se pude hacer con un sistema operativo.

Espero que este libro os sirva para algo.