Introducción a las ciencias de la computación con JAVA

Elisa Viso G.
Canek Peláez V.

TEMAS DE COMPUTACIÓN

las prensas de ciencias

Elisa Viso G. y Canek Peláez V.

Introducción a las ciencias de la computación con JAVA

Facultad de Ciencias, UNAM
2007

Esta obra aparece gracias al apoyo
del proyecto PAPIME PE-100205

Introducción a las ciencias de la computación con JAVA
1ª edición, 2007

Diseño de portada: Laura Uribe

Circuito exterior. Ciudad Universitaria. México 04510
cse@fciencias.unam.mx

ISBN: 978-970-32-4268-9

Impreso y hecho en México

Índice general

Índice de figuras

Índice de algoritmos y listados

Introducción 1

La disciplina de la computación es el estudio sistemático de procesos algorítmicos que describen y transforman información: su teoría, análisis, diseño, eficiencia, implementación y aplicación. La pregunta fundamental subyacente en toda la computación es, "¿Qué puede ser (eficientemente) automatizado?"

Peter Denning, 2005.

Hay mucha confusión respecto a términos que, aparentemente describen a la misma disciplina. Usaremos a lo largo de estas notas los términos computación y ciencias de la computación casi indistintamente. Es necesario recalcar que estamos usando el término computación como abreviatura para ciencias de la computación, con el significado particular que le estamos dando a este último en nuestro contexto. El error más común es el de confundir la programación con la computación. La diferencia que existe entre estos dos términos es tal vez la misma que existe entre saber la fórmula para resolver una ecuación de segundo grado y conocer la teoría de ecuaciones. Si bien la programación es una parte de la computación, la computación contempla muchísimos otros aspectos que no forzosamente tienen que ver con la programación o llevarse a cabo con una computadora. También se utilizan los términos de ingeniería y ciencias de la computación y, excepto por el enfoque que se pudiera dar en uno u otro caso, estaríamos hablando del mismo cuerpo de conocimientos.

Otro término que se utiliza frecuentemente (sobre todo en nuestro medio) es el de informática. Si bien en muchos casos, se utiliza este término para referirse

a todo lo que tiene que ver con computación, nosotros lo entendemos más bien como refiriéndose a aquellos aspectos de la computación que tienen que ver con la administración de la información (sistemas de información, bases de datos, etc.). Al igual que la programación, la informática la podemos considerar contenida propiamente en la computación.

El término cibernética es un término forjado por los soviéticos en los años cincuenta. Sus raíces vienen de combinar aspectos biológicos de los seres vivos con ingeniería mecánica, como es el caso de los robots, la percepción remota, la simulación de funciones del cuerpo, etc. A pesar de que se utiliza muchas veces en un sentido más general, no lo haremos así en estas notas.

1.1 Conceptos generales

El objeto fundamental de estudio de las ciencias de la computación son los *algoritmos* y, en su caso, su *implementación*. Veamos antes que nada la definición de *algoritmo*:

Definición 1.1 Un *algoritmo* es un método de solución para un problema que cumple con:

1. Trabaja a partir de 0 o más datos (*entrada*).
2. Produce al menos un resultado (*salida*).
3. Está especificado mediante un número finito de pasos (*finitud*).
4. Cada paso es susceptible de ser realizado por una persona con papel y lápiz (*definición*).
5. El seguir el algoritmo (la ejecución del algoritmo) lleva un tiempo finito (*terminación*).

Estamos entonces preocupados en ciencias de la computación por resolver problemas; pero no cualquier problema, sino únicamente aquéllos para los que podamos proporcionar un método de solución que sea un algoritmo – más adelante en la carrera demostrarán ustedes que hay más problemas que soluciones, ya no digamos soluciones algorítmicas.

La segunda parte importante de nuestra disciplina es la *implementación* de algoritmos. Con esto queremos decir el poder llevar a cabo un algoritmo dado (o diseñado) de manera automática, en la medida de lo posible.

En la sección que sigue exploraremos la historia de estos dos conceptos y la manera en que se distinguieron para conformar lo que hoy conocemos como ciencias de la computación.

1.2 Breve historia del concepto de algoritmo

La computación es una disciplina mucho muy antigua. Tiene dos raíces fundamentales:

- La búsqueda de una sistematización del pensamiento, que dicho en nuestro terreno se interpreta como la búsqueda de algoritmos para resolver problemas, algunas veces generales y otras concretos.
- La búsqueda para desarrollar implementos o medios que permitan realizar cálculos de manera precisa y eficiente.

En esta segunda raíz podemos reconocer los implementos de cómputo como el ábaco chino y el ábaco japonés, las tablillas de los egipcios para llevar la contabilidad de las parcelas cultivadas y, en general, todos aquellos mecanismos que permitían de manera más expedita, conocer un cierto valor o llevar a cabo un cálculo.

Respecto a la búsqueda de la sistematización del pensamiento, en la Antigua Grecia se hizo una contribución enorme en este dirección con el desarrollo del método axiomático en matemáticas y el desarrollo de la geometría como un sistema lógico deductivo, en el que juegan un papel mucho muy importante el *modus ponens* $((A \implies B \wedge A) \implies B)$ y el *modus tollens* $((A \implies B \wedge \neg B) \implies \neg A)$. Lo que se pretende con estos mecanismos es, a partir de un conjunto de hechos aceptados (axiomas) y mediante un cierto conjunto de reglas del juego (reglas de inferencia o de transformación) lograr determinar la validez de nuevos hechos. En el fondo lo que se buscaba era un algoritmo que describiera o automatizara la manera en que los seres humanos llegamos a conclusiones. Estos patrones de razonamiento se utilizan no sólo en matemáticas, sino en la vida diaria y han dado origen a una disciplina muy extensa y rigurosa que es la lógica matemática. No creemos necesario remarcar el gran avance que ha tenido esta disciplina en la última mitad del siglo XX. Sin embargo, es importante notar que el desarrollo ha estado siempre presente, no únicamente en este último período. Para ello presentamos en la siguiente página una tabla con aquellos eventos históricos que

consideramos más relevantes en el desarrollo de la computación, con una pequeña anotación de cual fue la aportación en cada una de las instancias.

Cuadro 1.1 Resumen de la historia de Ciencias de la computación

2000 AC	Babilonios y egipcios	Tablas para cálculos aritméticos, como raíces cuadradas, interés compuesto, área de un círculo (8/9*1/2=3.1604...)
Siglo IV AC	Aristóteles (384-322 AC)	Lógica formal con Modus Ponens y Modus Tollens
825	Abu Ja'far Mohammed ibn Müsa al-Khowärizmì	Libro sobre "recetas" o métodos para hacer aritmética con los números arábigos
1580	François Viète (1540-1603)	Uso de letras para las incógnitas: surgimiento del álgebra
1614	John Napier (1550-1617)	Huesos de Napier para la multiplicación. El concepto de logaritmo
1620	Edmund Gunter (1581-1626)	Primer antecesor de la regla de cálculo
Siglos XVI y XVII	Galileo (1564-1642)	Formulación matemática de la Física
Siglo XVI y XVII	Descartes (1596-1650)	Descubre la geometría analítica, posibilitando la aplicación del álgebra a problemas geométricos, y por lo tanto a problemas que tenían que ver con movimiento físico
1623	Wilhelm Schickard	Primera calculadora digital: sumaba, restaba automáticamente; multiplicaba y dividía semiautomático
1642-1644	Blaise Pascal (1623-1662)	Calculadora que sobrevivió. Sólo sumaba y restaba automáticamente
Siglo XVII y XVIII	Gottfried Wilhelm Leibniz (1646-1717)	Coinventor del cálculo con Newton. Primer investigador occidental de la aritmética binaria. Inventor de la "rueda de Leibniz", que hacía las 4 operaciones aritméticas. Fundamentos de la lógica simbólica

Cuadro 1.1 (cont.) Resumen de la historia de Ciencias de la computación

Siglo XIX	Charles Babbage (1791-1871)	Máquina diferencial y máquina analítica
1800	Ada Lovelace	Primer programador de la máquina analítica de Babbage
1854	Pehr George Scheutz	Construyó un modelo de la máquina diferencial de Babbage
1854	George Boole (1815-1864)	Estableció los fundamentos para el estudio moderno de la Lógica Formal
1890	Herman Hollerith	Uso de equipo tabulador (de "registro unitario")
1893	Leonardo Torres y Quevedo (1852-1936)	Máquina electromecánica basada en la de Babbage
1900	David Hilbert (1862-1943)	Propone a los matemáticas el encontrar un sistema de axiomas lógico matemático único para todas las áreas de la matemática
1928	Hollerith	Elaboración de tablas de posición de la luna utilizando su máquina de registro unitario: uso científico de herramientas pensadas para procesamiento de datos
1936	Kurt Gödel (1906-1978)	Demuestra que lo que propone Hilbert no es posible, i.e. que hay problemas matemáticos inherentemente insolubles
1936	Alan Turing (1912-1954)	Atacó el problema de cuando se puede decir que se tiene un método de solución, que el problema no tiene solución, etc. Diseña la Máquina de Turing
1939-1945	Wallace J. Eckert con John W. Mauchly	Extensión de la máquina tabuladora de IBM para propósitos científicos. Diseño y construcción de la ENIAC, primera gran computadora digital totalmente electrónica
1937-1942	Claude Shannon	El uso del álgebra booleana para el análisis de circuitos electrónicos. Liga entre la teoría y el diseño

Cuadro 1.1 (cont.) Resumen de la historia de Ciencias de la computación

1940	John V. Atanasoff (1903-)	Solución de ecuaciones lineales simultáneas. Computadora ABC
1937-1944	Howard T. Aiken (1900-1937)	Construcción de la MARK I que tenía: ▪ Posibilidad de manejar números positivos y negativos ▪ Posibilidad de utilizar diversas funciones matemáticas ▪ Totalmente automática ▪ Posibilidad de ejecutar operaciones largas en su orden natural
1943-1945	John W. Mauchly	ENIAC. Primera computadora totalmente electrónica
1944	John von Neumann	Notación para describir la circuitería de la computadora. Conjunto de instrucciones para la EDVAC. El concepto de programa almacenado - la noción de que los datos y los programas pueden compartir el almacenaje. El concepto de operación serial Aritmética binaria como mecanismo en las computadoras

1.3 Sistemas numéricos

A lo largo de esta historia se han transformado notablemente los símbolos que se usaron para denotar a los objetos de los cálculos, a los algoritmos, y a los resultados mismos. Podemos pensar, por ejemplo, en que una sumadora de las de Shickard usaba engranes como símbolos para realizar las cuentas. El hombre primitivo usaba notación "unaria" para contar: ponía tantos símbolos como objetos deseaba contar. Una primera abreviatura a estos métodos se dio con sistemas que agrupaban símbolos. Así por ejemplo, el sistema de números romanos combina símbolos para abreviar, y en lugar de escribir 100 rayas verticales utiliza el símbolo "C".

Un concepto importante es el del número cero. Su origen es mucho muy an-

tiguo. También los mayas tenían un símbolo asociado al cero. Este concepto es muy importante para poder utilizar notación posicional. La notación posicional es lo que usamos hoy en día, y consiste en que cada símbolo tiene dos valores asociados: el peso y la posición. Por ejemplo, el número 327.15 se puede presentar de la siguiente manera:

EJEMPLO 1.3.2

$$\mathbf{327{,}15 \;=\; 3\times 10^{2} \;+\; 2\times 10^{1} \;+\; 7\times 10^{0} \;+\; 1\times 10^{-1} \;+\; 5\times 10^{-2}}$$

Decimos que la notación es posicional, porque dependiendo de la posición que tenga un dígito con respecto al resto de los dígitos en un número, ése es el valor (o peso) que tiene. El sistema que usamos es el *decimal posicional*, porque es un sistema posicional que usa al número 10 como *base*. El peso que se le asigna a cada dígito depende del dígito y de su posición. Cada posición lleva un peso de alguna potencia de 10 (decimal) asignadas, alrededor del punto decimal, de la siguiente forma:

...	10^4	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3} ...
	10000	1000	100	10	1	,1	,01	,001
4513,6 =		4	5	1	3	. 6		
30100 =	3	0	1	0	0	. 0	0	0
,075 =						. 0	7	5

Hay varias reglas que observamos respecto a la notación posicional:

I. En cada *posición* se coloca un solo dígito.

II. Los ceros antes del primer dígito distinto de cero, desde la izquierda, no aportan nada al número.

III. Los ceros a la derecha del punto decimal y antes de un dígito distinto de cero sí cuentan. No es lo mismo .75 que .00075.

IV. Los ceros a la derecha del último dígito distinto de cero después del punto decimal no cuentan.

v. Los dígitos que podemos utilizar son del 0 al 9.

vi. Cada dígito aporta su valor específico multiplicado por el peso de la posición que ocupa.

Sabemos todos trabajar en otras bases para la notación posicional. Por ejemplo, base 8 (mejor conocida como *octal*) sería de la siguiente manera:

EJEMPLO 1.3.3

...	8^4	8^3	8^2	8^1	8^0	8^{-1}	8^{-2}	8^{-3}	...	
	4096	512	64	8	1	,125	,15625	,001953125		
$476{,}1_8$			4	7	6	. 1				$= 318{,}125_{10}$
41370_8		4	1	3	7	.				$= 2143_{10}$

También podemos pensar en base 16 (*hexadecimal*), para lo que requerimos de 16 símbolos distintos. Los dígitos del 0 al 9 nos proporcionan 10 de ellos. Tenemos el problema de que con la restricción de que cada posición debe ser ocupada por un único dígito, tenemos que "inventar" símbolos (o dígitos) para 6 valores que nos faltan, y que serían del 10 al 15 inclusive. La tradición es utilizar las letras A, B, C, D, E y F para los valores consecutivos 10, 11, 12, 13, 14 y 15 respectivamente. Siguiendo la notación posicional, pero en base 16, tenemos los siguientes ejemplos:

...	16^4	16^3	16^2	16^1	16^0	16^{-1}	16^{-2}	...	
	65536	4096	256	16	1	,0625	,00390625		
$476{,}1_{16}$			4	7	6	. 1			$= 1142{,}0625_{10}$
$BA7C_{16}$		11	10	7	12	.			$= 47740_{10}$

Pasamos ahora a la base 2, que es la más importante hoy en día en computación. Los primeros implementos de cómputo que usaban notación posicional (los ábacos, huesos de Napier, calculadoras) usaban base 10 (los ábacos, en realidad, usaban una especie de base 5. ¿Por qué?). Mientras las calculadoras se construían con partes físicas (engranes, cuentas) la base 10 no era muy distinta de cualquier otra base. Pero al intentar construir calculadoras (o computadoras) electrónicas, la base 10 presentaba problemas de implementación: ¿cómo distinguir entre el 4 y el 5, cuando se estaba midiendo en términos de niveles, analógicamente? Por lo tanto se optó, aunque no desde el principio, por usar base 2, que tiene un mapeo a

la electrónica bastante simple. Para base 2 requerimos de dos símbolos, y utilizamos el 0 y el 1. El 0 se puede interpretar como ausencia y el 1 como presencia, por lo que más allá de cierto nivel se asume presencia. Esto es mucho más sencillo que ver si tenemos uno de 10 (u 8) niveles distintos. La tabla que sigue corresponde a la notación posicional en base 2:

...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	...	
	128	64	32	16	8	4	2	1	,5	,25	,125		
$11001101{,}11_2$	1	1	0	0	1	1	0	1	. 1	1	1		$= 205{,}75_{10}$
101000_2			1	0	1	0	0	0	.				$= 40_{10}$

Como se puede ver, tratándose de números enteros, es fácil pasar de una base cualquiera a base 10, simplemente mediante la fórmula

$$num_{10} = \sum_{i=n}^{0} d_i \times b^i$$

donde d_i se refiere al dígito correspondiente en la i-ésima posición, con la posición 0 en el extremo derecho, y b^i se refiere a la base elevada a la potencia de la posición correspondiente.

$$\begin{aligned} 101000_2 = & \sum_{i=5}^{0} d_i \times 2^i \\ =& 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ =& \ 32 + 0 + 8 + 0 + 0 + 0 \\ =& \ 40_{10} \end{aligned}$$

Para pasar de base 10 a cualquier otra base, se utiliza el algoritmo 1.1.

Hay que notar que en la línea 7 se está concatenando un símbolo, por lo que si la base es mayor a 10, se tendrán que utilizar símbolos para los dígitos mayores que 9. Así, para pasar de base 10 a base 16 el número 8575_{10} el algoritmo se ejecutaría de la siguiente manera:

dividendo$_{10}$	**cociente**$_{10}$	**residuo**$_{10}$	**Símbolo residuo**$_{16}$	**Número**$_{16}$
8575	535	15	F	F
535	33	7	7	$7F$
33	2	1	1	$17F$
2	0	2	2	$217F$

Algoritmo 1.1 Algoritmo para pasar de base 10 a base h.

```
1: dividendo = num10;
2: divisor = h;
3: residuo = 0;
4: número = ;
5: repeat
6:    residuo = dividendo % divisor;
7:    número = pegar(residuo,número);
8:    dividendo = dividendo ÷ divisor;
9: until (dividendo = 0);
```

Para que nos convenzamos que, en efecto $8575_{10} = 217F_{16}$, procedemos a descomponer $217F$ en potencias de 16:

$$\begin{aligned} 217F_{16} &= 2 \times 16^3 + 1 \times 16^2 + 7 \times 16^1 + 15 \times 16^0 \\ &= 2 \times 4096 + 1 \times 256 + 7 \times 16 + 15 \\ &= 8192 + 256 + 112 + 15 \\ &= 8575 \end{aligned}$$

En general, para pasar de una base a otra, todo lo que se tiene que hacer es la división en la base en la que se encuentra el número que queremos convertir. Por ejemplo, si deseamos pasar un número base 8 a base 6, lo haríamos de la siguiente manera:

I. Consideremos que para base 8 contamos con los dígitos del 0 al 7, mientras que para base 6 únicamente los dígitos del 0 al 5.

II. Seguimos el algoritmo 1.1, pero haciendo la división y la resta en base 8. El resultado se muestra a continuación:

dividendo$_8$	**cociente**$_8$	**residuo**$_8$	**Símbolo residuo**$_6$	**Número**$_6$
7535	1217	3	3	3
1217	155	1	1	13
155	22	1	1	113
22	3	0	0	0113
3	0	3	3	30113

Trabajemos con los dos números en base 10 para corroborar que el algoritmo trabajó bien:

$$\begin{aligned} 7535_8 &= 7 \times 8^3 + 5 \times 8^2 + 3 \times 8^1 + 5 \times 8^0 \\ &= 7 \times 512 + 5 \times 64 + 3 \times 8 + 5 \times 1 \\ &= 3584 + 320 + 24 + 5 \\ &= 3933_{10} \end{aligned}$$

$$\begin{aligned} 30113_6 &= 3 \times 6^4 + 0 \times 6^3 + 1 \times 6^2 + 1 \times 6^1 + 3 \times 6^0 \\ &= 3 \times 1296 + 0 \times 216 + 1 \times 36 + 1 \times 6 + 3 \times 1 \\ &= 3888 + 0 + 36 + 6 + 3 \\ &= 3933_{10} \end{aligned}$$

En general, para pasar de una base B a otra base b, lo que tenemos que hacer es expresar b en base B, y después llevar a cabo el algoritmo en base B. Cada vez que tengamos un residuo, lo vamos a tener en base B y hay que pasarlo a base b. Esto último es sencillo, pues el residuo es, forzosamente, un número entre 0 y b.

Cuando una de las bases es potencia de la otra, el pasar de una base a la otra es todavía más sencillo. Por ejemplo, si queremos pasar de base 8 a base 2 (binario), observamos que $8 = 2^3$. Esto nos indica que cada posición octal se convertirá a exactamente a tres posiciones binarias. Lo único que tenemos que hacer es, cada dígito octal, pasarlo a su representación binaria:

7535_8	7	5	3	5	
	111	101	011	101	111101011101_2

Algo similar se hace para pasar de base 16 a base 2, aunque tomando para cada dígito base 16 cuatro dígitos base 2.

El proceso inverso, para pasar de base 2, por ejemplo, a base 16, como $16 = 2^4$ deberemos tomar 4 dígitos binarios por cada dígito hexadecimal:

111101011101_2	1111_2	0101_2	1101_2	
	15_{10}	5_{10}	13_{10}	
	F_{16}	5_{16}	D_{16}	$F5D$

Las computadoras actuales son, en su inmensa mayoría, digitales, esto es, que representan su información de manera discreta, con dígitos. Operan en base 2

(binario) ya que la electrónica es más sencilla en estos términos. Sin embargo, hay procesos que no son discretos, como las ondas de luz o sonoras. Pero hoy en día se pueden alcanzar excelentes aproximaciones de procesos continuos mediante dígitos binarios. Para ello se cuenta con componentes analógicos/digitales que transforman señales analógicas (continuas) en señales digitales (discretas). Hubo una época en que se tenía mucha fe en las computadoras analógicas, aquellas que funcionaban con dispositivos continuos, pero prácticamente han desaparecido del mercado, excepto por algunas de propósito muy específico, o las que convierten señales analógicas en señales digitales, o viceversa.

La siguiente pregunta que debemos hacernos es:

¿Cuáles son los distintos elementos que requerimos para poder implementar un algoritmo en una computadora digital?
¿Cómo se representan en binario esos distintos elementos?

Pensemos, por ejemplo, en las máquinas de escribir. La orden para que se escriba una letra determinada se lleva a cabo oprimiendo una cierta tecla. Esto es porque hay una conexión mecánica (física) entre la tecla del teclado y el dado que imprime la tecla. La primera computadora, la ENIAC, funcionaba de manera muy similar. Cada vez que se deseaba que resolviera algún problema, se alambraban los paneles de la computadora para que hubiera conexiones físicas entre lo que se recibía en el teletipo y las operaciones que se ejecutaban. De manera similar, las calculadoras mecánicas, al darle vuelta a una manivela, se conseguía que los engranes seleccionados efectuaran una determinada operación.

En las computadoras modernas, de propósito general, vienen alambradas para reconocer ciertos patrones, de forma similar a como lo hacía el telar de Jackard o la máquina del censo de Hollerith. Cada patrón indica una operación a realizarse. Los patrones son números binarios con un número fijo de posiciones (***bi****nary digi****ts***). A cada conjunto de posiciones de un cierto tamaño se le llama una *palabra*. El tamaño de palabra es, en general, una potencia de 2: 8, 16, 32, 64, 128. A los grupos de 8 bits se les llama *byte*. Al conjunto de patrones distintos es a lo que se conoce como *lenguaje de máquina*, que por supuesto es "personal" de cada modelo o tipo de computadora. Originalmente se distinguía entre micro, mini y computadoras por el tamaño en bits de sus palabras. El tamaño de la palabra va unido al poderío de la máquina: si tengo más bits por palabra tengo más patrones posibles, y por lo tanto un lenguaje de máquina más extenso y posibilidad de representar números más grandes o con más precisión[1] Las primeras microcomputadores tenían palabras de 8 bits, mientras que las "grandes" computadoras tenían palabras de 64 bits. Las

[1]En breve veremos la representación de datos en la computadora.

supercomputadoras, que surgieron alrededor de 1985, tenían palabras de 128 bits y posibilidades de proceso en paralelo.

El tamaño de la palabra también le da velocidad a una computadora, pues indica el número de bits que participan electrónicamente en cada operación.

1.4 La arquitectura de von Neumann

Se le llama *arquitectura de una computadora* a la organización qué tiene en sus componentes electrónicos, y la manera como éstos están integrados para funcionar.

Lo que se conoce como *arquitectura de von Neumann* es una organización muy parecida a la de Babbage: tenemos un *procesador central* – el *molino* de Babbage – en el que se ejecutan las operaciones aritméticas y de comparación (lógicas); una *memoria central* que se utiliza para almacenar datos, resultados intermedios y el programa a ejecutarse; tenemos unidades de entrada y salida (*input/output*) que sirven para darle a la computadora el programa y los datos y recibir los resultados; por último, tenemos *memoria externa o auxiliar*, como discos, diskettes, cintas magnéticas, que nos sirven para almacenar, ya sean datos o programas, de una ejecución a otra, sin tener que volver a realizar el proceso, o sin que tengamos que volverlos a proporcionar. Un esquema de una computadora con arquitectura de von Neumann se muestra en la figura 1.1.

Las flechas que van de un componente a otro pueden tener distintas formas de funcionar y muy diversas capacidades. Una de las formas en que funciona es lo que se conoce como un *bus*. Por ejemplo, si la capacidad de la línea que va de memoria al procesador es de menos de 1 palabra, aunque la computadora tenga palabras muy grandes, la velocidad de la máquina se va a ver afectada.

La memoria está "cuadriculada", o dividida en *celdas*. Cada celda ocupa una posición dentro de la memoria, aunque en principio cada una es igual a cualquier otra: tiene el mismo número de bits y las mismas conexiones. Se puede ver como un vector de celdas, cuyo primer elemento tiene el índice cero. Se habla de posiciones "altas" y posiciones "bajas" refiriéndose a aquéllas que tienen índices grandes o pequeños respectivamente. En cada celda de memoria se puede colocar (escribir, copiar) una instrucción, un número, una cadena de carácteres, etc.

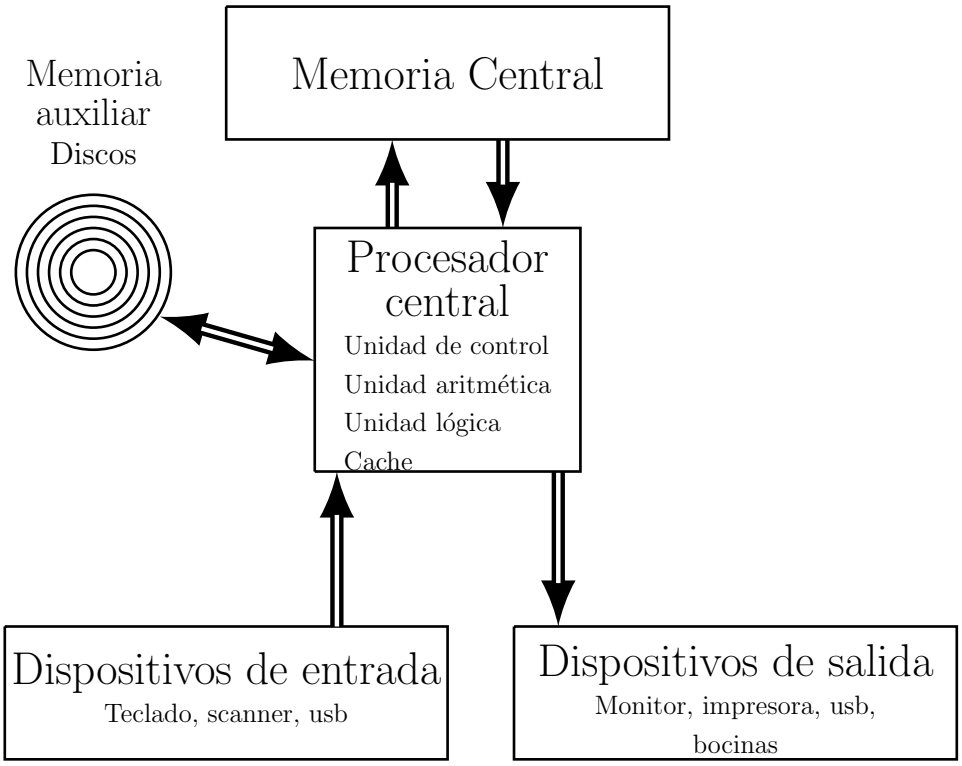
El proceso mediante el que se ejecuta un programa es el siguiente:

1. Se coloca el programa en la memoria de la computadora (se carga, *load*).

II. La unidad de control en el procesador se encarga de ver cuál es la siguiente instrucción. Ésta aparece, simplemente, como un patrón de bits (un *código de máquina*).

III. La Unidad de Control se encarga de ejecutar la instrucción, valiéndose para ello de cualquiera de los componentes de la máquina.

Figura 1.1 Arquitectura de von Neumann



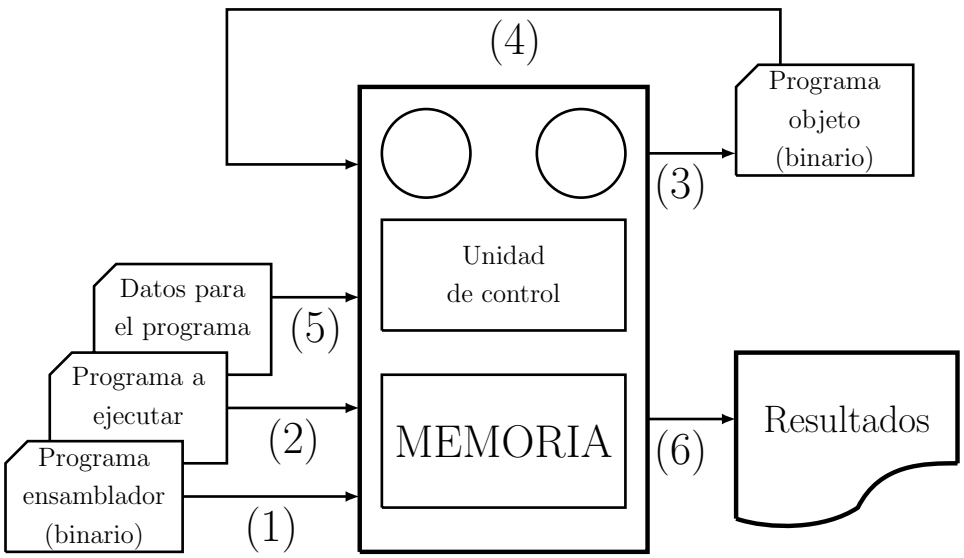
El tipo de instrucciones que tiene una computadora incluyen instrucciones para sumar, restar, multiplicar, dividir, copiar, borrar, recorrer el patrón de bits,

comparar y decidir si un número es mayor que otro, etc. En realidad son instrucciones que hacen muy pocas cosas y relativamente sencillas. Recuérdese que se hace todo en sistema binario.

Lenguajes de programación

Un *lenguaje de programación* es aquél que nos permite expresar un problema de tal manera que podamos instalarlo (cargarlo) en la computadora y se ejecute. Hasta ahora sólo hemos visto el lenguaje de máquina, y éste era el único disponible con las primeras computadoras de propósito general.

Figura 1.2 Proceso para ejecutar un programa escrito en ensamblador



Programar en binario es, en el mejor de los casos, sumamente tedioso y complicado. El programador (que es quien escribe los programas) tiene que tener un conocimiento muy profundo de la computadora para la que está programando. Además de eso, tiene que ejercer un cuidado extremo para no escribir ningún 1 por 0, o para no equivocarse de dirección. Lo primero que se hizo fue escribir en octal, ya que era un poco más claro que binario. El siguiente paso fue asociar *nemónicos* a las instrucciones, asociando a cada patrón de bits un "nombre" o identificador: *add, sub, mul, div,* etc. A esto se le llamó *lenguaje ensamblador*. Se construyó un programa, llamado *ensamblador*, que se encargaba de traducir los nemónicos de este estilo y las direcciones escritas en octal a binario. Este programa no es algo complicado de hacer. Tanto el programa ensamblador como el programa a traducir se alimentaban, cargados en tarjetas perforadas – ver figura 1.2. El primer paquete era el programa ensamblador, escrito en binario; a continuación se presentaba el programa que se deseaba traducir, como datos del primero. La computadora contaba con un tablero, en el que se le indicaba que empezara a cargar el programa ensamblador, y una vez cargado (1), empieza a ejecutarlo. El programa ensamblador indicaba que tenía que traducir las siguientes tarjetas (o cinta perforada), conforme las fuera leyendo (2), y producir tarjetas con el programa en binario (3) o, más adelante, cargar el programa binario a memoria para ser ejecutado (4); al ejecutarse el programa en binario, se leían los datos (5) y se producía el resultado (6).

El siguiente paso en lenguajes de programación fue el de los *macroensambladores*, que asignaban *etiquetas* a las posiciones de memoria que se estaban utilizando para acomodar los datos y resultados intermedios, y las posiciones donde iba a quedar el código. El proceso de traducción era sencillo, ya que se agregaban a las tablas de traducción del código los equivalentes en octal. También se permitía construir secuencias pequeñas de código, a las que nuevamente se les asociaba un nombre o *identificador*, y que podían presentar *parámetros*. A estas secuencias se les llamaba *macros* y de ahí el nombre de *macroensamblador*.

Las computadoras se estaban utilizando tanto con fines científicos como comerciales. En el uso científico era muy común expresar fórmulas matemáticas, que tenían que "despedazarse" en operaciones básicas para poderse llevar a cabo – ver figura 1.3.

El siguiente paso importante fue el de permitirle a un programador que especificara su fórmula como se muestra en la parte izquierda de la figura 1.3. El primer lenguaje de uso generalizado orientado a esto fue *FORTRAN – For*mula *Tran*slator, alrededor de 1956. Pocos años después se desarrolló un lenguaje para usos comerciales, donde lo que se deseaba es poder manejar datos a distintos niveles de agregación. Este lenguaje se llamaba *COBOL – CO*mon *B*ussiness *O*riented *L*anguage. Ambos lenguajes, o versiones modernizadas, sobreviven hasta nuestros

Figura 1.3 Codificación en ensamblador de fórmulas matemáticas.

Fórmula

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Programa simplificado

```
def  x1   100    mul  b2   b    b
def  a    102    mul  ac   a    c
def  b    104    mul  ac   4    ac
def  c    106    mul  a2   2    a
def  ac   108    add  rad  ac   b2
def  a2   110    sqrt rad  rad
def  b2   112    sub  x1   rad  b
def  rad  114    div  x1   x1   a2
```

días.

Estos lenguajes tenían un formato también más o menos estricto, en el sentido de que las columnas de las tarjetas perforadas estaban perfectamente asignadas, y cada elemento del lenguaje tenía una posición, como en lenguaje ensamblador. El proceso por el que tiene que pasar un programa en alguno de estos lenguajes de programación para ser ejecutado, es muy similar al de un programa escrito en lenguaje ensamblador, donde cada enunciado en lenguaje de "alto nivel" se traduce a varios enunciados en lenguaje de máquina (por eso el calificativo de "alto nivel", alto nivel de información por enunciado), que es el único que puede ser ejecutado por la computadora.

Hacia finales de los años 50 se diseñó un lenguaje, *ALGOL* – *ALG*orithmic *O*riented *L*anguage – que resultó ser el modelo de desarrollo de prácticamente todos los lenguajes orientados a algoritmos de hoy en día, como Pascal, C, C++, Java y muchos más. No se pretende ser exhaustivo en la lista de lenguajes, baste mencionar que también en los años 50 surgió el lenguaje *LISP*, orientado a inteligencia artificial; en los años 60 surgió *BASIC*, orientado a hacer más fácil el acercamiento a las computadoras de personas no forzosamente con antecedentes científicos. En los años 60 surgió el primer lenguaje que se puede considerar *orientado a objetos*, *SIMULA*, que era una extensión de *ALGOL*. En los aproximadamente 50 años que tienen en uso las computadoras, se han diseñado y usado más de 1,000 lenguajes de programación, por lo que pretender mencionar siquiera a los más importantes es una tarea titánica, y no el objetivo de estas notas.

Representación de la información

Acabamos de ver que la representación de los programas debe ser, eventualmente en lenguaje de máquina, o sea, en binario. También tenemos restricciones similares para el resto de la información, como son los datos, los resultados intermedios y los resultados finales. Al igual que con los lenguajes de programación, si bien la computadora sólo tiene la posibilidad de representar enteros positivos en binario, debemos encontrar la manera de poder representar letras, números de varios tipos, como enteros negativos, reales, racionales, etc. Para ello se sigue una lógica similar a la del lenguaje de máquina.

Carácteres

Supongamos que tenemos un tamaño de palabra de 16 bits y queremos representar letras o carácteres. Simplemente hacemos lo que hacíamos en la primaria cuando queríamos mandar "mensajes secretos": nos ponemos de acuerdo en algún código.

El primer código que se utilizó fue el *BCD*, que utilizaba 6 bits por carácter. Con esto se podían representar 64 carácteres distintos (en 6 bits hay 64 posibles enteros, del 0 al 63). Con este código alcanzaba para las mayúsculas, los dígitos y algunos carácteres importantes como los signos de operación y de puntuación.

Con el perfeccionamiento de las computadoras se requirieron cada vez más carácteres, por lo que se extendió el código a 7 y 8 bits, con el código *ASCII*, que se usó mucho para transmitir información, y el código *EBCDIC* que se usó como código nativo de muchas computadoras, respectivamente. El lenguaje Java utiliza *Unicode*, que ocupa 16 bits, para representar a cada carácter. Con esto tiene la posibilidad de utilizar casi cualquier conjunto de carácteres de muchísimos de los idiomas en uso actualmente.

Se requiere de programas que transformen a un carácter en su código de máquina y viceversa. Éstos son programas sencillos que simplemente observan "patrones" de bits y los interpretan, o bien, observan carácteres y mediante una tabla, los convierten al patrón de bits en el código que utilice la computadora.

Prácticamente todo manual de programación trae una tabla de los distintos códigos que corresponden a los carácteres. Estas tablas vienen con varias columnas; en la primera de ellas vendrá el carácter, y en columnas subsecuentes su código en octal hexadecimal, binario, y utilizando alguno de estos esquemas para dar

el código que le corresponde en ASCII, EBCDIC o Unicode. Por supuesto que requerimos de 32,768 para mostrar la codificación de Unicode, por lo que no lo haremos, mas que en la medida en que tengamos que conocer el de algunos carácteres específicos.

Números enteros

Ya vimos la manera en que se representan números enteros en la computadora: simplemente tomamos una palabra y usamos notación posicional binaria para ver el valor de un entero.

Hoy en día las computadoras vienen, en su gran mayoría, con palabras de al menos 32 bits. Eso quiere decir que podemos representar enteros positivos que van desde el 0 (32 bits apagados) hasta $2^{32} - 1$ (todos los bits prendidos). Pero, ¿cómo le hacemos para representar enteros negativos? Tenemos dos opciones. La primera de ellas es la más intuitiva: utilizamos un bit, el de la extrema izquierda, como signo. A esta notación se le llama *de signo y magnitud*. Si éste es el caso, tenemos ahora 31 bits para la magnitud y 1 bit para el signo – ver figura 1.4 con palabras de 16 bits.

Figura 1.4 Enteros en signo y magnitud.

s	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	0	0	0	1	0	1	1	0	1	1	1	0	0	1	1	+1467
1	0	0	0	1	0	1	1	0	1	1	1	0	0	1	1	−1467

La representación se signo y magnitud es muy costosa. Por ejemplo, cuando se suman dos cantidades que tienen signos opuestos, hay que ver cuál es la que tiene mayor magnitud, pues la suma se puede convertir en resta de magnitudes. Veamos el algoritmo 1.2 en la siguiente página.

Como se puede ver, los circuitos que se requieren para sumar dos números en notación de signo y magnitud son muy complicados, y por lo tanto muy caros.

El otro modo de codificar enteros positivos y negativos es lo que se conoce como *complemento a 2*. En este método, al igual que con signo y magnitud, se parte al dominio en 2 partes, los que tienen al bit de potencia más alta en 0, y los que lo tienen en 1; los que tienen el bit más alto en cero son los enteros positivos, y los que lo tienen en 1 son los enteros negativos. Para saber la magnitud del número, en el caso de los positivos se calcula igual que con signo y magnitud, pero

Algoritmo 1.2 Sumando dos números representados con signo y magnitud.

```
1: Sean a y b los enteros a sumar.
2: S_a = signo de a; M_a = magnitud de a.
3: S_b = signo de b; M_b = magnitud de b.
4: if S_a = S_b
5:     S_suma = S_a.
6:     M_suma = M_a + M_b.
7: else
8:     if M_a > M_b
9:         S_suma = S_a.
10:        M_suma = M_a − M_b.
11:    else
12:        S_suma = S_b.
13:        M_suma = M_b − M_a.
```

en el caso de los negativos, la magnitud es la que resulta de restar la palabra de una con una posición más, y donde todas las posiciones originales de la palabra tienen 0, con un 1 en la posición extra. Veamos algunos ejemplos en la figura 1.5 en la página opuesta.

Como vemos en la figura 1.5, el bit 15 (el que corresponde a 2^{15}) también nos indica de cierta forma, como en la notación de signo y magnitud, cuando tenemos un entero negativo. En el caso de 16 bits, los enteros positivos son del 0 al $2^{15}-1 = 32,767$ que corresponde a una palabra de 16 bits con todos menos el bit 15 prendidos – ver figura 1.5, línea (1). A partir del número $2^{15} = 32,768$ y hasta $2^{16}-1 = 65,535$, que corresponde a todos los bits en una palabra de 16 bits prendidos, estamos representando a números negativos – ver en la figura 1.5, línea (3). En estos últimos, el bit 15 está siempre prendido, y por eso reconocemos el signo del número. La magnitud (o el valor absoluto) del número que estamos viendo se obtiene sacando el complemento a 2 de la palabra en cuestión. El complemento a 2 se obtiene de dos maneras posibles:

I. Se resta en binario de un número de 17 bits – en la figura 1.5, línea (2) – con ceros en todos los bits, menos el bit 16, que tiene 1.
II. Se complementan cada uno de los bits de la palabra de 16 bits (se cambian los 1's por 0's y los 0's por 1's) y después se le suma 1 a lo que se obtuvo.

Ambas maneras de obtener la magnitud (el complemento a 2) se observa en la línea (4) de la figura 1.5.

La gran ventaja de la notación en complemento a 2 es que es sumamente fácil hacer operaciones aritméticas como la suma y la resta. En complemento a 2, todo lo que tenemos que hacer es sumar (o restar) utilizando los 16 bits. Pudiera suceder,

sin embargo, que el resultado no sea válido. Por ejemplo, si sumamos dos números positivos y el resultado es mayor que la capacidad, tendremos acarreo sobre el bit 15, dando aparentemente un número negativo. Sabemos que el resultado es inválido porque si en los dos sumandos el bit 15 estaba apagado, tiene que estar apagado en el resultado. Algo similar sucede si se suman dos números negativos y el resultado "ya no cabe" en 16 bits – ver figura 1.6.

Figura 1.5 Números en complemento a 2

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
Un entero positivo en complemento a 2:																	
0	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	$3,592$	(1)
La palabra con el complemento a 2:																$2^{16} =$	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$65,536$	(2)
Un entero negativo en complemento a 2:																	
1	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	$36,360$	(3)
La magnitud del entero original:																	
0	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	$29,176$ (4)

Figura 1.6 Suma de dos números con complemento a 2

	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	0	0	0	0	0	0	0	0	1	1	0	0	1	1	1	1	207
+	0	0	0	1	0	0	0	1	1	1	0	0	0	0	1	1	4547
	0	0	0	1	0	0	1	0	1	0	0	1	0	0	1	0	4754
	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	25
+	1	1	1	1	1	1	0	0	0	1	0	1	0	1	0	1	−939
	1	1	1	1	1	1	0	0	0	1	1	0	1	1	1	0	−914

Pueden verificar, sumando las potencias de 2 donde hay un 1, que las sumas en la figura 1.5 se hicieron directamente y que el resultado es el correcto.

La desventaja del complemento a 2 es que se pueden presentar errores sin que nos demos cuenta de ello. Por ejemplo, si le sumamos 1 al máximo entero positivo (una palabra con 0 en el bit 15 y 1's en el resto de los bits) el resultado resulta ser un número negativo, aquel que tiene 1's en todas las posiciones de la palabra – ver figura 1.7.

Figura 1.7 Sumando 1 al máximo entero positivo

	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32767
+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	−32768

En algunos lenguajes de programación este tipo de errores se detectan en ejecución, pero en la mayoría no. Hay que tener presente que la representación interna es con complemento a 2, para manejar de manera adecuada este tipo de posibles errores.

Muchas veces el tamaño de la palabra de una computadora no es suficiente para los números que deseamos representar. Entonces, el lenguaje de programación puede usar más de una palabra para representar enteros, simplemente utilizando notación posicional base 2^{16}, de manera similar a como se maneja la base 2. Se dice entonces que la aritmética se hace *por software*.

Números reales

Para representar a los número reales tenemos realmente dos opciones:

Punto fijo: Como su nombre lo indica, el fabricante (o el lenguaje de programación) elige una posición entre dos bits, y lo que se encuentra a la derecha de esa posición es la parte fraccionaria y lo que se encuentra a la izquierda es la parte entera, ambos vistos como enteros en notación posicional binaria – ver figura 1.8.

Como se puede ver en la figura 1.8, se mantiene la notación de complemento a 2, el bit más alto indicándonos que el número es negativo.

Figura 1.8 Notación de punto fijo.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	·	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	1	0	0	1	1	0	1		0	1	0	0	0	0	1	1	77,67
1	0	0	1	1	0	1	1		0	1	0	1	0	1	1	0	−100,162

Una de las ventajas de este tipo de notación es que es muy sencillo hacer operaciones aritméticas, pues se usa a toda la palabra como si fuera un entero y el proceso de colocar el punto decimal se hace al final. Sin embargo, tiene una gran desventaja que es la poca flexibilidad para representar números que tengan muchos dígitos en la fracción, o muy pocos.

Punto flotante: El punto flotante es otra manera de representar números reales. Básicamente como se logra es dividiendo a la palabra en dos partes, una para la mantisa y la otra para el exponente, utilizando lo que se conoce como notación científica. Veamos los siguientes ejemplos:

$$\begin{aligned} 1{,}32456 \times 10^{6} &= 1324560 \\ 1{,}32456 \times 10^{-6} &= {,}00000132456 \\ -1{,}32456 \times 10^{3} &= -\,1324{,}56 \end{aligned}$$

Como podemos ver del ejemplo anterior, nos ponemos de acuerdo en cuántos dígitos van a estar a la izquierda del punto decimal, y todos los números reales los representamos con ese número de enteros. A continuación, damos una potencia de 10 por la que hay que multiplicar el número, para obtener el número que deseamos.

Una abreviatura de esta notación sería escribiendo los dos números anteriores de la siguiente forma:

$$\begin{aligned} 1{,}32456E6 &= 1324560 \\ 1{,}32456E-6 &= {,}00000132456 \\ -1{,}32456E3 &= -\,1324{,}56 \end{aligned}$$

Esta representación es muchísimo más versátil que la de punto fijo, y de hecho es la que se usa generalmente. Al momento de representar los números, se hace en binario. Si damos mucho espacio para los exponentes tenemos la posibilidad de representar números muy grandes o muy pequeños (con el exponente negativo). Si en cambio le damos mucho espacio a la mantisa, vamos a tener números con mucha precisión (muchas cifras significativas). Es conveniente encontrar un balance entre la magnitud y la precisión. Por ejemplo, la IEEE tiene sus estándares.

Las operaciones con este tipo de números son un poco más complejas que con punto fijo. Por ejemplo, si deseamos sumar dos números, tenemos primero que llevarlos a que tengan el mismo exponente, y una vez hecho esto se puede llevar a cabo la suma. En cambio, multiplicar dos números es sumamente fácil. ¿Por qué?

Al igual que en los números enteros, además de lo que nos proporcione el hardware de la computadora como tamaño de palabra, por software se pueden usar tantas palabras como uno quiera para representar a un entero o a un real. Cada lenguaje de programación proporciona un conjunto de enteros y reales de diversos "tamaños".

La inmensa mayoría de las computadoras utilizan complemento a 2 y mantisa y exponente para representar números.

Limitaciones en la representación interna

Vimos ya que en la computadora no podemos representar a todos y cualquier entero: tenemos un número finito de enteros distintos que podemos representar, no importa que tan grande sea la palabra de una computadora dada, ya que tenemos un número finito de combinaciones de 0's y 1's en cualquier tamaño dado de palabra.

Algo similar ocurre con los números reales. No sólo no tenemos la posibilidad de representar a un número infinito de números reales, sino que además tampoco tenemos la densidad de los números reales. Como estamos usando binario para representar a números que nosotros manejamos – y pensamos – como números en base 10, habrá números que no tengan un representación exacta al convertirlos a base 2 (por supuesto que estamos hablando de números fraccionarios). Adicionalmente, al agregarle 1 a una mantisa, no obtenemos el "siguiente" número real, ya que estamos sumando, posiblemente, en la parte fraccionaria. Por ello, no es posible tener una representación para todos y cada uno de los números reales en

un intervalo dado: nuevamente, éste es un número infinito y lo que tenemos es un número finito de combinaciones.

1.5 Ejecución de programas

Mencionamos arriba que lo único que puede ejecutar una computadora de hardware es un programa escrito en *su* lenguaje de máquina. Por lo que para que nuestros programas escritos en Java puedan ser ejecutados, deberán estar escritos en lenguaje de máquina.

¿Recuerdan lo que hacíamos para obtener un programa escrito en lenguaje de máquina a partir de uno escrito en ensamblador? Se lo dábamos como datos a un programa que traducía de enunciados en ensamblador a enunciados en lenguaje de máquina. Algo similar hacemos cuando estamos trabajando en un lenguaje de alto nivel.

En general, tenemos dos maneras de conseguir ejecutar un programa escrito en un lenguaje de alto nivel. La primera de ellas es mediante un *intérprete*, y la segunda mediante un *compilador*. Veamos qué queremos decir con cada uno de estos términos.

Definición 1.4 Un *intérprete* es un programa que una vez cargado en la memoria de una computadora, y al ejecutarse, procede como sigue:

- Toma un enunciado del programa en lenguaje de alto nivel, llamado el *código fuente*.
- Traduce ese enunciado y lo ejecuta.
- Repite estas dos acciones hasta que alguna instrucción le indique que pare, o bien tenga un error fatal en la ejecución.

Definición 1.5 Un *compilador* es un programa, una vez que reside en memoria y al ejecutarse, toma un programa fuente y lo traduce *completo* a un programa en otro lenguaje de programación, que generalmente es lenguaje de máquina, equivalente.

Mientras que un intérprete va traduciendo y ejecutando, el compilador no se encarga de ejecutar, sino simplemente de producir un programa equivalente, susceptible de ser cargado a la memoria de la máquina y ejecutado.

A los intérpretes se les conoce también como *máquinas virtuales*, porque una vez que están cargados en una máquina, se comportan como si fueran *otra* computadora, aquella cuyo lenguaje de máquina es el que se está traduciendo y ejecutando.

1.5.1. Filosofías de programación

Dependiendo del tipo de problema que queramos resolver – numérico, administrativo, de propósito general, inteligencia artificial, lógico – tenemos distintos lenguajes de programación que permiten representar de mejor manera los formatos de los datos y los recursos que requerimos para resolver el problema. Así, para procesos numéricos se requiere de bibliotecas muy extensas con funciones matemáticas, un manejo sencillo de matrices y, en general, de espacios de varias dimensiones, etc. El lenguaje que fue diseñado para este tipo de problemas fue FORTRAN, y recientemente se usa C. Si lo que queremos es resolver problemas administrativos tenemos a COBOL, que se rehusa a morir, o Visual Basic que provee una fabricación rápida de interfaces con el usuario[2]. Como representantes de lenguajes de propósito general tenemos Pascal, C, Algol. Para problemas que involucran cambios de estado en los datos, o situaciones que suceden no forzosamente una después de la otra se cuenta con lenguajes orientados a objetos como C++, SmallTalk y Java. Para resolver problemas que involucran manejo simbólico de datos, como lo que se requiere para Inteligencia Artificial, se tienen lenguajes como LISP y Scheme. Para problemas de tipo lógico se tiene ProLog. En fin, casi cualquier tipo de aplicación que se nos ocurra, se puede diseñar un lenguaje de programación para el cual el lenguaje que se utilice en el algoritmo sea mucho muy cercano al lenguaje de programación: éste es el objetivo que se persigue cuando se diseñan nuevos lenguajes de programación. Este curso se enfocará a resolver problemas que se expresan fácilmente con orientación a objetos, y el lenguaje que utilizaremos es Java.

Es importante darse cuenta que, finalmente, cualquier problema se puede resolver utilizando cualquier lenguaje: finalmente, todo programa tiene que traducirse a lenguaje de máquina, por lo que no importa en qué lenguaje hayamos programado, terminaremos con un programa equivalente escrito en lenguaje de máquina. El meollo del asunto es, simplemente, qué tanto trabajo nos cuesta pensar en el problema en un lenguaje pensado para resolver otro tipo de problemas. Buscamos que el lenguaje de programación se ajuste de manera sencilla a nuestro modo de

[2]Una *interfaz con el usuario* es aquel programa que permite una comunicación mejor entre el usuario y el programa en la computadora. Se usa para referirse a las intefaces gráficas.

pensar respecto al problema que deseamos resolver.

1.6 Características de Java

Java es un lenguaje *orientado a objetos*, cuyo principal objetivo de diseño es que fuera *portátil*. Una manera de hacer programas escritos en Java es mediante el siguiente truco:

- Se traduce el programa escrito en Java a un lenguaje de bajo nivel, tipo lenguaje de máquina, pero que no sea el de una máquina en específico.
- Se construye (programa) un intérprete de este "lenguaje de máquina", y entonces se ejecuta el programa en lenguaje de máquina en la máquina virtual de Java.

Esto resulta relativamente sencillo. El "lenguaje de máquina" de Java se llama *bytecode*. Es más fácil construir una máquina virtual que entienda el bytecode que construir un compilador para cada posible lenguaje de máquina. Además, una vez que está definida la máquina virtual, se le pueden agregar capacidades al lenguaje, simplemente dando su transformación a bytecode.
Por todo esto, para ejecutar un programa escrito en Java necesitamos:

a. Tener un compilador de Java que traduzca de programas escritos en Java a bytecode (`javac`).

b. Tener un intérprete de bytecode (o máquina virtual de Java) a la que se le da como datos el programa en bytecode y los datos pertinentes al programa.

El proceso del software 2

2.1 ¿Qué es la programación?

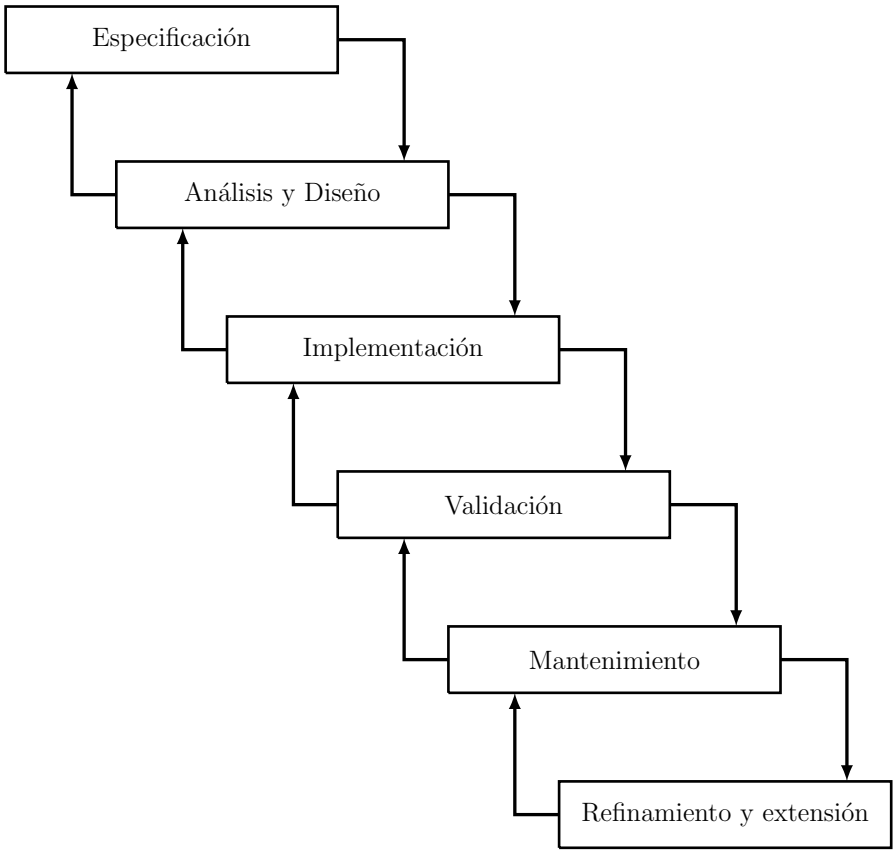
Como ya platicamos al hablar de lenguajes de programación, la programación consiste en *elaborar un algoritmo, escrito en un lenguaje susceptible de ser ejecutado por una computadora, para resolver una clase de problemas.*

Podemos pensar en un algoritmo como la definición de una función. Una vez definida ésta, se procede a aplicar la función a distintos conjuntos de argumentos (datos) para obtener el resultado.

El proceso que nos lleva finalmente a aplicar un programa a datos determinados conlleva varias etapas, algunas de ellas repetitivas. En el estado inicial de este proceso tendremos un enunciado del problema que deseamos resolver, junto con la forma que van a tomar los datos (cuántos datos, de qué tipo). En el estado final deberemos contar con un programa correcto que se puede instalar en una computadora para ser ejecutado con cualquier conjunto de datos que cumpla las especificaciones planteadas. Por ello deberemos observar los siguientes pasos (ver figura 2.1 en la siguiente página para el proceso del Software.[1])

[1]El diagrama que presentamos es lo que corresponde a un proceso de software en espiral, ya que se regresa una y otra vez a etapas anteriores, hasta que se pueda llegar al final del diagrama.

Figura 2.1 Proceso del software.



Especificación del problema: Se nos presenta el enunciado del problema y deberemos determinar de manera precisa las *especificaciones*: de dónde partimos (con qué entradas contamos) y a dónde queremos llegar (cuál es el resultado que deseamos obtener). Como producto de esta etapa deberemos producir tres incisos:

a. Enunciado preciso del problema.

b. Entradas.

c. Salidas.

Análisis y diseño del algoritmo: Planteamos la manera en que vamos a transformar los datos de entrada para obtener el resultado que buscamos, y procedemos a elaborar un modelo de la solución. Muchas veces este modelo involucra varios pasos intermedios (estados de los datos), o más que un resultado concreto, buscamos un cierto comportamiento, como en el caso de un juego o una simulación – como la de un reloj. En estos últimos casos deberemos pensar en procesos sucesivos que nos lleven de un estado de cosas (estado de los datos) al estado inmediato sucesor – fotos instantáneas – y cuál o cuáles son las transformaciones de los datos que nos producen el siguiente estado.

Dependiendo del ambiente (dominio) de nuestro problema, las soluciones que diseñemos deberán tener distintas características. La primera de ellas, que comparten todos los dominios es:

a) La solución debe ser *correcta*, *eficiente* y *efectiva*.

La única excepción posible a esta reglas se presenta si estamos haciendo un programa para ayudarnos a calcular algo, o para sacarnos de un brete momentáneo o coyuntural, el programa se va a utilizar pocas veces en un lapso corto de tiempo; tal vez hasta podemos eliminar la característica de que sea eficiente (haga el trabajo en un tiempo razonable, utilizando una cantidad razonable de recursos como la memoria). En los primeros años de las computadoras, casi todos los programas eran de este tipo: la gente los hacía para sí mismos, o para un grupo reducido que estaba muy al tanto de lo que estaba pasando.

Hoy en día, en que las computadoras están en todo, la mayoría de la gente involucrada haciendo programas los hace para otros. Además, el tamaño de los sistemas ha crecido tanto, que ya casi nadie es el "dueño" de sus programas, sino que se trabaja en el contexto de proyectos grandes, con mucha gente involucrada. En este tipo de situaciones, que hoy en día son más la regla que la excepción, se requiere además que los programas:

b) Sean modulares. Se puedan trazar claramente las fronteras entre pedazos del programa (o sistema), para que la tarea se pueda repartir.

c) Tengan un bajo nivel de acoplamiento. Esta propiedad se refiere a que utilicen lo menos posible del mundo exterior y entreguen lo mínimo posible: que haya poco tráfico entre los módulos, de tal manera que haya la posibilidad de reutilizarlos.

d) Alta cohesión, que se refiere al hecho de que todo lo que esté relacionado (funciones, datos) se encuentren juntos, para que sean fáciles de localizar, entender y modificar.

Implementación o construcción del modelo: En esta etapa deberemos traducir nuestro algoritmo al lenguaje de programación que hayamos elegido. A esta etapa se le conoce también como de *codificación*. Ésta no es una labor muy difícil, si es que tenemos un diseño que siga la filosofía[2] del lenguaje de programación.

Asegurar la documentación del programa: Esto no es – o no debería ser – una etapa separada del proceso, ya que lo ideal es que, conforme se va progresando se vaya documentando el programa. Hoy en día existen muchos paquetes que ayudan a llevar a cabo el diseño y que ayudan con la documentación, por lo que Actualmente es imperdonable que falte documentación en los programas. En el presente, en que los programas se hacen en un 90 % de los casos para otros, es muy importante que el programa lo pueda entender cualquier lector humano; esto se logra, la mayoría de las veces, mediante la documentación.

Prueba y validación: Debemos tener la seguridad de que nuestro programa hace lo que se supone debe de hacer. Para esto hay pruebas informales, que consisten en presentarle al programa distintos conjuntos de datos y verificar que el programa hace lo que tiene que hacer. Estas pruebas deben incluir conjuntos de datos erróneos, para verificar que el programa sabe "defenderse" en situaciones anómalas.

La validación de los programas conlleva demostraciones matemáticas de que los enunciados cambian el estado de los datos de la manera que se busca.

Mantenimiento: La actividad que mayor costo representa hoy en día es la del mantenimiento de los sistemas. Tenemos dos tipos de mantenimiento: correctivo y extensivo. El correctivo tiene que ver con situaciones que el programa o sistema no está resolviendo de manera adecuada. El mantenimiento extensivo tiene que ver con extender las capacidades del programa o sistema para que enfrente conjuntos nuevos de datos o entregue resultados adicionales. Sin una documentación adecuada, estas labores no se pueden llevar a cabo. Sin un diseño correcto, es prácticamente imposible extender un sistema.

[2]Con esto nos referimos a la manera como el lenguaje de programación interpreta el mundo: por procedimientos, orientado a objetos, funcional, lógico. A esto le llamamos *paradigmas* de programación.

Refinamiento y Extensión: Esta etapa generalmente la llevan a cabo personas distintas a las que diseñaron el sistema. Se busca que cuando se extiende un sistema, no se tape un hoyo haciendo otro. La modularidad ayuda, pues se entiende mejor el funcionamiento del programa si se ataca por módulos. Pero esta propiedad debe tener, además, la propiedad de encapsulamiento, que consiste en que cada módulo tiene perfectamente delimitado su campo de acción, la comunicación entre módulos es mucho muy controlada y una implementación interna (modo de funcionar) que pueda ser cambiada sin que altere su comportamiento y sin que haya que cambiar nada más.

Estas etapas, como ya mencionamos en algunas de ellas, no se presentan forzosamente en ese orden. Más aún, muchas veces se da por terminada una de ellas, pero al proceder a la siguiente surgen problemas que obligan a regresar a etapas anteriores a modificar el producto o, de plano, rehacerlo. Queremos insistir en la importancia de tener un buen análisis y un buen diseño, no importa cuánto tiempo nos lleve: ésta es la llave para que en etapas posteriores no tengamos que regresar a rehacer, o nos veamos en la necesidad de tirar trabajo ya hecho.
Pasemos a detallar un poco más algunas de las etapas de las que hablamos.

Especificación

Una buena especificación, sea formal o no, hace hincapié, antes que nada, en cuál es el resultado que se desea obtener. Este resultado puede tomar muy distintas formas. Digamos que el cómputo corresponde a un modelo (la instrumentación o implementación del modelo).

Podemos entonces hablar de los estados por los que pasa ese modelo, donde un estado corresponde a los valores posibles que toman las variables. Por ejemplo, si tenemos las variables x, y, z, un posible estado sería:

$$\{ \quad x = 5 \qquad y = 7 \qquad z = 9 \quad \}$$

Si tenemos la especificación de un programa (rutina) que intercambie los valores de dos variables x, y, podemos pensarlo así:

$$\{ \qquad x = K_1 \qquad y = K_2 \qquad \}$$

es el *estado inicial* (con los valores que empieza el proceso), mientras que

$$\{ \qquad x = K_2 \qquad y = K_1 \qquad \}$$

corresponde al *estado final* deseado. Podemos adelantar que una manera de lograr que nuestro modelo pase del estado inicial al estado final es si a x le damos el

valor de K_2 (el valor que tiene y al empezar) y a y le damos el valor que tenía x. Podemos representar este proceso de la siguiente manera:

$\{ \quad x = K_1 \qquad y = K_2 \quad \}$ // estado inicial
A x ponle K_2
A y ponle K_1
$\{ \quad x = K_2 \qquad y = K_1 \quad \}$ // estado final

y podemos garantizar que nuestras operaciones cumplen con llevar al modelo al estado final. Sin embargo, las operaciones que estamos llevando a cabo están considerando a K_1 y K_2 valores constantes. Un proceso más general sería el siguiente:

$\{ \quad x = K_1 \qquad y = K_2 \quad \}$ // estado inicial
En t copia el valor de x
En x copia el valor de y
// ¡En estos momentos x y y valen lo mismo!
En y copia el valor de t
$\{ \quad x = K_2 \qquad y = K_1 \quad \}$ // estado final

y este proceso funciona no importando qué valores dimos para x y y.

Resumiendo, un estado de un proceso, cómputo o modelo es una lista de variables, cada una de ellas con un cierto valor.

Una especificación de un problema es la descripción del problema, que como mínimo debe tener el estado final del cómputo. El estado inicial puede ser fijado a partir del estado final (determinando qué se requiere para poder alcanzar ese estado), o bien puede darse también como parte de la especificación.

Análisis y diseño

Podemos decir, sin temor a equivocarnos, que la etapa de diseño es la más importante del proceso. Si ésta se lleva a cabo adecuadamente, las otras etapas se simplifican notoriamente. La parte difícil en la elaboración de un programa de computadora es el análisis del problema (definir exactamente qué se desea) y el diseño de la solución (plantear cómo vamos a obtener lo que deseamos). Para esta actividad se requiere de creatividad, inteligencia y paciencia. La experiencia juega un papel muy importante en el análisis y diseño. Dado que la experiencia se debe adquirir, es conveniente contar con una metodología que nos permita ir construyendo esa experiencia.

Así como hay diversidad en los seres humanos, así hay maneras distintas de analizar y resolver un problema. En esa búsqueda por "automatizar" o matematizar el proceso de razonamiento, se buscan métodos o metodologías que nos lleven desde la especificación de un problema hasta su mantenimiento, de la mejor manera posible. El principio fundamental que se sigue para analizar y diseñar una solución es el de *divide y vencerás*, reconociendo que si un problema resulta demasiado complejo para que lo ataquemos, debemos partirlo en varios problemas de menor magnitud. Podemos reconocer dos vertientes importantes en cuanto a las maneras de dividir:

a) Programación o análisis estructurado, que impulsa la descomposición del problema en términos de acciones, convergiendo todas ellas en un conjunto de datos, presentes todo el tiempo. La división se hace en términos del proceso, reconociendo distintas etapas en el mismo. Dado que el énfasis es en el proceso, cada módulo del sistema corresponde a un paso o etapa del proceso. Cuando usamos este enfoque perdemos la posibilidad de "encapsular", pues los datos se encuentran disponibles para todo mundo, y cada quien pasa y les mete mano. Además, hay problemas que son difíciles de analizar en términos de "etapas". Por ejemplo, los juegos de computadora estilo aventura, las simulaciones de sistemas biológicos, el sistema de control de un dispositivo químico, un sistema operativo. Sin embargo, esta metodología es muy adecuada para diseñar, como lo acabamos de mencionar, un cierto proceso que se lleva a cabo de manera secuencial.

b) Análisis y diseño orientado a objetos. El análisis orientado a objetos pretende otro enfoque, partiendo al problema de acuerdo a los objetos presentes. Determina cuáles son las responsabilidades de cada objeto y qué le toca hacer a cada quién.

Implementación del modelo

Para la programación, como ya mencionamos, utilizaremos Java y aprovecharemos la herramienta JavaDoc para que la documentación se vaya haciendo, como es deseable, durante la codificación.

Mantenimiento

Porque se trata de un curso, no nos veremos expuestos a darle mantenimiento a nuestros programas. En las sesiones de laboratorio, sin embargo, tendrán que trabajar con programas ya hechos y extenderlos, lo que tiene que ver con el mantenimiento.

2.1.1. Conceptos en la orientación a objetos

Al hacer el análisis de nuestro problema, como ya mencionamos, trataremos de dividir a la solución en tantas partes como sea posible, de tal manera que cada parte sea fácil de entender y diseñar. La manera de dividir al problema será en términos de *actores* y sus respectivas responsabilidades (o facultades): qué puede y debe hacer cada actor para contribuir a la solución. Cada uno de estos actores corresponde a un *objeto*. Agrupamos y abstraemos a los objetos presentes en *clases*. Cada clase cumple con:

- Tiene ciertas características.
- Funciona de determinada manera.

Quedan agrupados en una misma clase aquellos objetos que presentan las mismas características y funcionan de la misma manera.

En el diseño orientado a objetos, entonces, lo primero que tenemos que hacer es *clasificar* nuestro problema: encontrar las distintas clases involucradas en el mismo.

Las clases nos proporcionan un *patrón* de comportamiento: nos dicen qué y cómo se vale hacer con los datos de la clase. Es como el guión de una obra de teatro, ya que la obra no se nos presenta hasta en tanto no haya actores. Los actores son los *objetos*, que son *ejemplares* o *instancias* de las clases (representantes de las clases).

Al analizar un problema deberemos tratar de identificar a los *objetos* involucrados. Una vez que tengamos una lista de los objetos (agrupando datos que tienen propósitos similares, por ejemplo), deberemos abstraer, encontrando características comunes, para definir las clases que requerimos.

Distinguimos a un objeto de otro de la misma clase por su nombre – *identidad* o *identificador*. Nos interesa de un objeto dado:

i. Su estado: cuál es el valor de sus atributos.

ii. Su conducta:

   - Qué cosas sabe hacer.
   - Cómo va a reaccionar cuando se le hagan solicitudes.

Lo correspondiente a *i.* está determinado por cada objeto, ya que cada objeto es capaz de almacenar su propia información. Lo correspondiente a *ii.* está dado por la definición de la clase, que nos da un *patrón de conducta*.

Tratando de aclarar un poco, pensemos en lo que se conoce como *sistema cliente/servidor*. *Cliente* es aquél que pide, compra, solicita algo: un servicio, un valor, un trabajo. *Servidor* es aquél que provee lo que se le está pidiendo. Esta relación de cliente/servidor, sin embargo, no es estática. El servidor puede tomar el papel de cliente y viceversa.

Lo que le interesa al cliente es que el servidor le proporcione aquello que el cliente está pidiendo. No le importa cómo se las arregla el servidor para hacerlo. Si el cliente le pide al servidor algo que el servidor no sabe hacer, que no reconoce, simplemente lo ignora, o le contesta que eso no lo sabe hacer.

El análisis orientado a objetos pretende reconocer a los posibles clientes y servidores del modelo, y las responsabilidades de cada quien. Divide la responsabilidad global del proceso entre distintos objetos.

Un concepto muy importante en la orientación a objetos es el *encapsulamiento* de la información. Esto quiere decir que cada objeto es dueño de sus datos y sus funciones, y puede o no permitir que objetos de otras clases ajenas vean o utilicen sus recursos.

Un objeto entonces tiene la propiedad de que encapsula tanto a los procesos (funciones) como a los datos. Esto es, conoce cierta información y sabe cómo llevar a cabo determinadas operaciones. La ventaja del encapsulamiento es que en el momento de diseñar nos va a permitir trazar una línea alrededor de operaciones y datos relacionados y tratarlos como una cápsula, sin preocuparnos en ese momento de cómo funciona, sino únicamente de qué es capaz de hacer.

En el caso de los objetos, la cápsula alrededor del mismo oculta al exterior la manera en que el objeto trabaja, el cómo. Cada objeto tiene una interfase pública y una representación privada. Esto nos permite poder hacer abstracciones más fácil y modelos más sencillos, pues lo que tomamos en cuenta del objeto es exclusivamente su interfase; posponemos la preocupación por su representación privada.

Públicamente un objeto "anuncia" sus habilidades: "puedo hacer estas cosas", "puedo decir estas cosas"; pero no dice cómo es que las puede hacer o cómo es que sabe las cosas. Los objetos actúan como un buen jefe cuando le solicitan a otro objeto que haga algo: simplemente le hacen la solicitud y lo dejan en paz para que haga lo que tiene que hacer; no se queda ahí mirando sobre su hombro mientras lo hace.

El encapsulamiento y el ocultamiento de la información colaboran para aislar a una parte del sistema de otras, permitiendo de esta manera la modificación y extensión del mismo sin el riesgo de introducir efectos colaterales no deseados. Para determinar cuáles son los objetos presentes en un sistema, se procede de la siguiente manera:

1. Se determina que funcionalidades e información están relacionadas y deben permanecer juntas, y se encapsulan en un objeto.

2. Entonces se decide qué funcionalidades e información se le van a solicitar a este objeto (qué servicios va a prestar). Éstos se mantienen públicos, mientras que el resto se esconde en el interior del objeto.

Esto se logra mediante *reglas de acceso*, que pueden ser de alguno de los tipos que siguen:

Público: Se permite el acceso a objetos de cualquier otra clase.

Privado: Sólo se permite el acceso a objetos de la misma clase.

En algunos lenguajes de programación se permiten otros tipos de acceso. Por ejemplo, en Java también tenemos los siguientes, pero que no pretendemos dejar claros por el momento:

Paquete: Se permite el acceso a objetos que están agrupados en el mismo paquete (generalmente un sistema).

Protegido: Se permite el acceso a objetos de clases que hereden de esta clase.

Veamos la terminología involucrada en el diseño orientado a objetos:

Mensajes (*messages*)

Un objeto le pide un servicio a otro mandándole un mensaje. A esta acción le llamamos el envío de un mensaje y es la única manera en que un objeto se puede comunicar con otro.

Un mensaje consiste del nombre de una operación y los argumentos que la operación requiere. La solicitud no especifica cómo debe ser satisfecha.

Comportamiento o conducta (*behaviour*)

El conjunto de mensajes a los que un objeto puede responder es a lo que se conoce como la conducta o el comportamiento del objeto.

Al nombre de la operación en el mensaje le llamamos el nombre del mensaje.

Métodos (*methods*)

Cuando un objeto recibe un mensaje, ejecuta la operación que se le solicita, mediante la ejecución de un método. Un método es un algoritmo paso a paso que se ejecuta como respuesta a un mensaje cuyo nombre es el mismo que el del método. Para que un método pueda ser invocado desde un objeto de otra clase, debe ser público. En el caso de algunos lenguajes de programación, a los métodos se les llama *funciones miembro* (***member functions***), porque son miembros de la clase u objeto.

Clases (*classes*)

Si dos objetos presentan el mismo comportamiento, decimos que pertenecen a la misma clase. Una clase es una especificación genérica para un número arbitrario de objetos similares. Las clases permiten construir una taxonomía de los objetos en un nivel abstracto, conceptual. Nos permiten describir a un grupo de objetos.

Por ejemplo, cuando describimos a los seres humanos damos las características que les son comunes. Cada ser humano es un objeto que pertenece a esa clase. Hay que insistir en que las clases corresponden únicamente a descripciones de objetos, no tienen existencia en sí mismas.

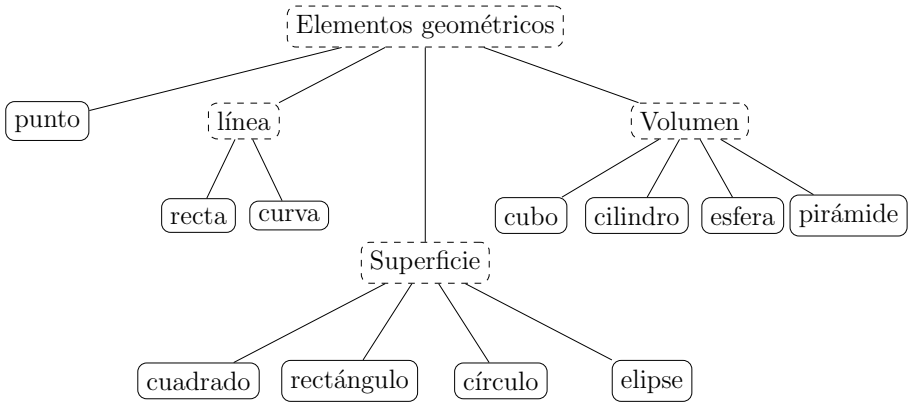
Ejemplares (*instances*)

Las instancias corresponden, de alguna manera, a los "ejemplares" que podemos exhibir de una clase dada. Por ejemplo, describimos a los seres humanos y decimos que Fulanito es un ejemplar de la clase de seres humanos: existe, tiene vida propia, le podemos pedir a Fulanito que haga cosas. Fulanito es un objeto de la clase de seres humanos. Fulanito es el nombre (identificador, identidad) del ejemplar u objeto. Su comportamiento es el descrito por los métodos de su clase. Está en un cierto estado, donde el estado es el conjunto de datos (características) junto con valores particulares.

Un mismo objeto puede pasar por distintos estados. Por ejemplo, podemos definir que las características de un ser humano incluyen: estado de conciencia, estado de ánimo, posición. Fulanito puede estar dormido o despierto, contento o triste, parado o sentado, correspondiendo esto a cada una de las características (o

variables). Sin embargo, hay variables entre las que corresponden a un objeto, que si bien cambian de un objeto a otro (de una instancia a otra), una vez definidas en el objeto particular ya no cambian, son invariantes. Por ejemplo, el color de los ojos, el sexo, la forma de la nariz.

Figura 2.2 Árbol de herencia en clases.



Herencia (*inheritance*)

Es frecuente encontrar una familia de objetos (o clases) que tienen un núcleo en común, pero que difieren cada uno de ellos por algún atributo, una o más funciones. Cuando esto sucede, deberemos reconocer al núcleo original, identificando a una *clase abstracta* a partir de la cual cada uno de las clases de la familia son una *extensión*. A esta característica es a lo que se le conoce como *herencia*: las clases en la familia de la que hablamos, *heredan* de la clase abstracta o *clase base*, remedando un árbol como el que se muestra en la figura 2.2. En esta figura las clases abstractas o base se presentan encuadradas con líneas intermitentes, mientras que las subclases se presentan encuadradas con línea continua. A este tipo de esquema le llamamos de *jerarquía de clases*.

Polimorfismo (*polymorphism*)

Dado que tenemos la posibilidad de agrupar a las clases por "familias", queremos la posibilidad de que, dependiendo de cuál de los herederos se trate, una función determinada se lleve a cabo de manera "personal" a la clase. Por ejemplo, si tuviéramos una familia, la función de arreglarse se debería llevar a cabo de distinta manera para la abuela que para la nieta. Pero la función se llama igual: arreglarse. De la misma manera, en orientación a objetos, conforme definimos herencia podemos modificar el comportamiento de un cierto método, para que tome en consideración los atributos adicionales de la clase que hereda. A esto, el que el mismo nombre de método pueda tener un significado distinto dependiendo de la clase a la que pertenece el objeto particular que lo invoca, es a lo que se llama *polimorfismo* – tomar *varias formas*.

En resumen, presentados ante un problema, estos son los pasos que deberemos realizar:

1. Escribir de manera clara los requisitos y las especificaciones del problema.
2. Identificar las distintas clases que colaboran en la solución del problema y la relación entre ellas; asignar a cada clase las responsabilidades correspondientes en cuanto a información y proceso (atributos y métodos respectivamente); identificar las interacciones necesarias entre los objetos de las distintas clases (diseño).
3. Codificar el diseño en un lenguaje de programación, en este caso Java.
4. Compilar y depurar el programa.
5. Probarlo con distintos juegos de datos.

De lo que hemos visto, la parte más importante del proceso va a ser el análisis y el diseño, así que vamos a hablar de él con más detalle.

2.2 Diseño orientado a objetos

El diseño orientado a objetos es el proceso mediante el cual se transforman las especificaciones (o requerimientos) de un sistema en una especificación detallada

de objetos. Esta última especificación debe incluir una descripción completa de los papeles y responsabilidades de cada objeto y la manera en que los objetos se comunican entre sí.

Al principio, el proceso de diseño orientado a objetos es exploratorio. El diseñador busca clases, agrupando de distintas maneras para encontrar la manera más natural y razonable de abstraer (modelar) el sistema. Inicialmente consiste de los siguientes pasos:

1. Determina (encuentra) las clases en tu sistema.
2. Determina qué operaciones son responsabilidad de cada clase, y que conocimientos la clase debe mantener o tener presentes para poder cumplir con sus responsabilidades.
3. Determina las formas en las que los objetos colaboran con otros objetos para delegar sus responsabilidades.

Estos pasos producen:

- una lista de clases dentro de tu aplicación,
- una descripción del conocimiento y operaciones que son responsabilidad de cada clase, y
- una descripción de la colaboración entre clases.

Una vez que se tiene este esquema, conviene tratar de establecer una jerarquía entre las clases que definimos. Esta jerarquía establece las relaciones de herencia entre las clases. Dependiendo de la complejidad del diseño, podemos tener anidados varios niveles de encapsulamiento. Si nos encontramos con varias clases a las que conceptualizamos muy relacionadas, podemos hablar entonces de subsistemas. Un subsistema, desde el exterior, es visto de la misma manera que una clase. Desde adentro, es un programa en miniatura, que presenta su propia clasificación y estructura. Las clases proporcionan mecanismos para estructurar la aplicación de tal manera que sea reutilizable.

Si bien suena sencillo eso de "determina las clases en tu aplicación", en la vida real éste es un proceso no tan directo como pudiera parecer. Veamos con un poco de más detalle estos subprocesos:

1. Determina las clases en tu sistema (encuentra los objetos).

   Para poder determinar cuáles son los objetos en tu sistema, debes tener muy claro cuál es el objetivo del mismo. ¿Qué debe lograr el sistema? ¿Cuál

es la conducta que está claramente fuera del sistema? La primera pregunta se responde dando una especificación completa del problema. En un principio daremos descripciones narrativas del problema y nuestro primer paso deberá consistir en dividir el problema en subproblemas, identificando las clases.

En esta descripción narrativa, existe una relación entre los sustantivos o nombres y los objetos (o clases). Una primera aproximación puede ser, entonces, hacer una lista de todos los sustantivos que aparezcan en la especificación narrativa. Una vez que se tiene esta lista, debemos intentar reconocer similitudes, herencia, interrelaciones. Debemos clasificar a nuestros objetos para determinar cuáles son las clases que vamos a necesitar.

Las probabilidades de éxito en el diseño del sistema son directamente proporcionales a la exactitud y precisión con que hagamos esta parte del diseño. Si este primer paso no está bien dado, el modelo que obtengamos a partir de él no va a ser útil y vamos a tener que regresar posteriormente a "parcharlo".

2. Determina sus responsabilidades (métodos y estructuras de datos).

   Determinar las responsabilidades de un objeto involucra dos preguntas:

   - ¿Qué es lo que el objeto tiene que saber de tal manera que pueda realizar las tareas que tiene encomendadas?
   - ¿Cuáles son los pasos, en la dirección del objetivo final del sistema, en los que participa el objeto?

   Respondemos a esta pregunta en términos de responsabilidades. Posponemos lo más posible la definición de cómo cumple un objeto con sus responsabilidades. En esta etapa del diseño nos interesa qué acciones se tienen que llevar a cabo y quién es el responsable de hacerlo.

   De la misma manera que existe una cierta relación entre los sustantivos de la especificación y las clases, podemos asociar los verbos de la especificación a los métodos o responsabilidades. Si hacemos una lista de las responsabilidades y las asociamos a los objetos, tenemos ya un buen punto de partida para nuestro modelo.

   Un objeto tiene cinco tipos de funciones:

   - funciones constructoras, que son las encargadas de la creación de los objetos, así como de su inicialización (establecer el estado inicial);
   - funciones de implementación, que son aquellas que representan a los servicios que puede dar un objeto de esa clase;

- funciones de acceso que proporcionan información respecto al estado del objeto;
- funciones auxiliares que requiere el objeto para poder dar sus servicios, pero que no interactúan con otros objetos o clases; y
- funciones de actualización y manipuladoras, que modifican el estado del objeto.

3. Determina su colaboración (mensajes). En esta subdivisión nos interesa responder las siguientes preguntas respecto a cada una de las clases definidas:

   - ¿Con qué otros objetos tiene que colaborar para poder cumplir con sus responsabilidades (a quién le puede delegar)?
   - ¿Qué objetos en el sistema poseen información que necesita o sabe como llevar a cabo alguna operación que requiere?
   - ¿En qué consiste exactamente la colaboración entre objetos?

   Es importante tener varios objetos que colaboran entre sí. De otra manera, el programa (o sistema) va a consistir de un objeto enorme que hace todo.

   En este paso, aunque no lo hemos mencionado, tenemos que involucrarnos ya con el cómo cumple cada objeto con su responsabilidad, aunque no todavía a mucho nivel de detalle. Un aspecto muy importante es el determinar dónde es que se inician las acciones. En el esquema de cliente/servidor del que hemos estado hablando, en este punto se toman las decisiones de si el objeto subcontrata parte de su proceso, si es subcontratado por otro objeto, etc. Es importante recalcar que mientras en la vida real algunos de los objetos tienen habilidad para iniciar por sí mismos su trabajo, en el mundo de la programación orientada a objetos esto no es así: se requiere de un agente que inicie la acción, que ponga en movimiento al sistema.

   Es muy importante en esta etapa describir con mucha precisión cuáles son las entradas (input) y salidas (output) de cada objeto y la manera que cada objeto va a tener de reaccionar frente a una solicitud. En teoría, un objeto siempre da una respuesta cuando se le solicita un servicio. Esta respuesta puede ser

   - No sé hacerlo, no lo reconozco.
   - Un valor o dato que posee.
   - La realización de un proceso

La manera como estas respuestas se manifiestan van a cambiar de sistema a sistema.

4. Determina la accesibilidad de las funciones y datos. Una vez que se tiene clasificado al sistema, es importante perseguir el principio de ocultamiento de la información. Esto consiste en decidir, para cada clase, cuáles de sus funciones y sus datos van a estar disponibles, públicos, y cuáles van a estar ocultos dentro de los objetos de la clase. Es claro que los métodos o funciones forman parte de la sección pública, pues van a ser solicitados por otros objetos. También es claro que los datos deben permanecer ocultos, pues queremos que el objeto mismo sea el único que puede manipular su propio estado. No queremos que otra clase u objeto tenga acceso a los valores del objeto.

   Sin embargo, a veces requerimos de funciones que sólo el objeto necesita o usa. En estos casos, estas funciones las vamos a colocar también en el espacio privado de la clase, buscando que el acoplamiento entre clases sea mínimo: si nadie requiere de ese servicio, más que el mismo objeto, ¿para qué ponerlo disponible?

Si bien tratamos de dar una metodología para el diseño orientado a objetos, es imposible hacerlo de manera completa en unas cuantas páginas (hay cursos que se dedican únicamente a este tema). Lo que se menciona arriba son únicamente indicaciones generales de cómo abordar el problema, aprovechando la intuición natural que todos poseemos. Conforme avancemos en el material, iremos extendiendo también la metodología.

2.2.1. Tarjetas de responsabilidades

Como resultado de este análisis elaboraremos, para cada clase definida, lo que se conoce como una *tarjeta de responsabilidades*. Esta tarjeta registrará los atributos, responsabilidades y colaboración que lleva a cabo esa clase, y una breve descripción del objetivo de cada atributo y de cada método.

2.3 Diseño estructurado

Como ya mencionamos antes, para diseñar cada uno de los métodos o funciones propias de un sistema debemos recurrir a otro tipo de análisis que el orientado a objetos. Esto se debe fundamentalmente a que dentro de un método debemos llevar a cabo un algoritmo que nos lleve desde un estado inicial a otro final, pero donde no existe colaboración o responsabilidades, sino simplemente una serie de tareas a ejecutar en un cierto orden.

Tenemos cuatro maneras de organizar a los enunciados o líneas de un algoritmo:

Secuencial, donde la ejecución prosigue, en orden, con cada línea, una después de la otra y siguiendo la organización física. Por ejemplo:

1: pone 1 en x
2: suma 2 a x
3: copia x a y

se ejecutaría exactamente en el orden en que están listadas.

Iteración, que marca a un cierto conjunto de enunciados secuenciales e indica la manera en que se van a repetir. Por ejemplo:

1: pone 1 en x
2: Repite 10 veces desde el enunciado 3 hasta el 5:
3: suma 2 a x
4: copia x a y
5: Escribe el valor de x

En este caso, podemos decir que el estado inicial de las variables al llegar a la iteración es con x valiendo 1 y con y con un valor indeterminado. ¿Cuál es el estado final, al salir de la iteración?

La manera como indicamos el grupo de enunciados a repetirse es dando una sangría mayor a este grupo; siguiendo esta convención, el enunciado de la línea 2 podría simplemente ser

2: Repite 10 veces:

y el solo hecho de que los enunciados 3, 4 y 5 aparecen con mayor sangría da la pauta para que ésos sean los enunciados a repetirse.

Ejecución condicional, que se refiere a elegir una de dos o más opciones de grupos de enunciados. Por ejemplo:

1: poner un valor arbitrario a y, entre 0 y 9999
2: **Si** x es mayor que 1:
3: Divide a y entre x y coloca el resultado en z
4: Multiplica a z por $\frac{1}{3}$
5: escribe el valor de z
6: **Si** x **no** es mayor que 1:
7: multiplica a y por $\frac{1}{6}$
8: escribe el valor de y

En este caso planteamos dos opciones, una para cuando el estado inicial antes de entrar a la ejecución condicional sea con x teniendo un valor mayor que 1, y la otra para cuando x tenga un valor menor que 1 (que pudiera ser 0).

Recursividad, que es cuando un enunciado está escrito en términos de sí mismo, como es el caso de la definición del factorial de un número:

$$n! = \begin{cases} n \times (n-1)! & \text{para } n > 1 \\ \\ 1 & \text{para } n = 1 \end{cases}$$

Toda solución algorítmica que demos, sobre todo si seguimos un diseño estructurado, deberá estar dado en términos de estas estructuras de control. El problema central en diseño consiste en decidir cuáles de estas estructuras utilizar, cómo agrupar enunciados y como organizar, en general los enunciados del programa.

Una parte importante de todo tipo de diseño es la notación que se utiliza para expresar los resultados o modelos. Al describir las estructuras de control utilizamos lo que se conoce como pseudocódigo, pues escribimos en un lenguaje parecido al español las acciones a realizar. Esta notación, si bien es clara, resulta fácil una vez que tenemos definidas ya nuestras estructuras de control, pero no nos ayuda realmente a diseñar. Para diseñar utilizaremos lo que se conoce como la metodología de Warnier-Orr, cuya característica principal es que es un diseño controlado por los datos, i.e. que las estructuras de control están dadas por las estructuras que guardan los datos. Además, el diseño parte siempre desde el estado final del problema (qué es lo que queremos obtener) y va definiendo pequeños pasos que van transformando a los datos hacia el estado inicial del problema (que es lo que sabemos antes de empezar a ejecutar el proceso).

Empecemos por ver la notación que utiliza el método de Warnier-Orr, y dado que es un método dirigido por los datos, veamos la notación para representar

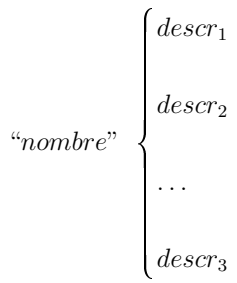
grupos de datos, que al igual que los enunciados, tienen las mismas 4 formas de organizarse: secuencial, iterativa, condicional o recursiva . Por supuesto que también debemos denotar la jerarquía de la información, donde este concepto se refiere a la relación que guarda la información entre sí. Representa a los datos con una notación muy parecida a la de teoría de conjuntos, utilizando llaves para denotar a los conjuntos de datos (o enunciados). Sin embargo, cada conjunto puede ser "refinado" con una "ampliación" de su descripción, que se encuentra siempre a la derecha de la llave. Otro aspecto muy importante es que en el caso de los "conjuntos" de Warnier-Orr el orden es muy importante. La manera en que el método de Warnier-Orr representa estos conceptos se explica a continuación:

Jerarquía

Abre llaves. El "nombre" de la llave es el objeto de mayor jerarquía e identifica al subconjunto de datos que se encuentran a la derecha de la llave. Decimos entonces que lo que se encuentra a la derecha de la llave "refina" (explica con mayor detalle) al "nombre" de la llave. Veamos la figura 2.3.

- Jerarquía: en la figura figura 2.3, "*nombre*" agrupa a $descr_1$, $descr_2$,. . . ,$descr_n$; decimos entonces que las descripciones son un refinamiento de "*nombre*", y que "*nombre*" es de mayor jerarquía que las descripciones.

Figura 2.3 Uso de llaves para denotar composición.



- Secuencia: el orden (la secuencia) se denota verticalmente: $descr_1$ se ejecuta antes que $descr_2$, y así sucesivamente, en el orden vertical en el que aparecen.

- Iteración: la repetición se representa con un paréntesis abajo del "nombre" donde se indican las reglas de repetición:

Figura 2.4 Iteración en diagramas de Warnier-Orr.

$$\begin{array}{c} \text{“}nombre\text{”} \\ \text{(Mientras te digan)} \end{array} \left\{ \begin{array}{l} descr_1 \\ descr_2 \\ \ldots \\ descr_n \end{array} \right.$$

En el la figura 2.4, la condición de repetición es “mientras te digan”. Eso querría decir que se ejecutan en orden, $descr_1$ hasta $descr_n$. Al terminar, se checa si “todavía me dicen”. Si es así, se regresa a ejecutar completo desde $descr_1$ hasta $descr_n$, y así sucesivamente hasta que “ya no me digan”, en cuyo caso sigo adelante.

- Condicional: Por último, para denotar selección se utiliza el símbolo del o exclusivo $\bigoplus$, que aparece entre una pareja de opciones – ver figura 2.5.

Figura 2.5 Selección en diagramas de Warnier-Orr.

$$\text{“}nombre\text{”} \left\{ \begin{array}{l} digito = \text{“}1'' \left\{ \begin{array}{l} \ldots \\ \ldots \end{array} \right. \\ \qquad \bigoplus \\ digito ='' 2'' \left\{ \begin{array}{l} \ldots \\ \ldots \end{array} \right. \\ \qquad \bigoplus \\ \ldots \\ \qquad \bigoplus \\ digito ='' 9'' \left\{ \begin{array}{l} \ldots \\ \ldots \end{array} \right. \end{array} \right.$$

Veamos cómo quedarían representados los pequeños procesos que dimos arriba en términos de la notación de Warnier-Orr en las figuras 2.6 y 2.7, donde el símbolo “$\leftarrow$” significa “obtén el valor de lo que está a la derecha y coloca ese valor en la variable que está a la izquierda”.

Figura 2.6 Diagramas de Warnier-Orr para secuencia.

$$secuencial \begin{cases} x \leftarrow 1 \\ x \leftarrow x + 2 \\ y \leftarrow x \end{cases}$$

Figura 2.7 Diagramas de Warnier-Orr para iteración.

$$iteración \begin{cases} Repite \\ (10) \end{cases} \begin{cases} x \leftarrow x + 2 \\ y \leftarrow x \\ escribe\ el\ valor\ de\ x \end{cases}$$

Por último, el bloque de pseudocódigo que tenemos para la ejecución condicional podría quedar como se ve en la figura 2.8.

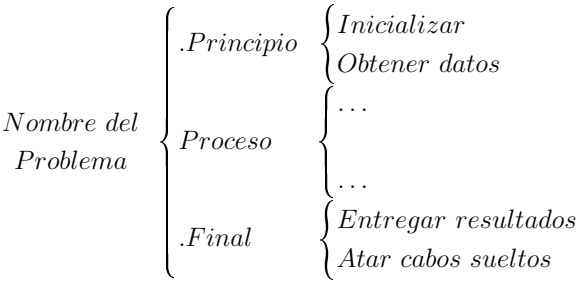
Figura 2.8 Diagrama de Warnier-Orr para selección.

$$selección \begin{cases} y \leftarrow random(10000) \\ x > 1 \begin{cases} z \leftarrow y/x \\ z \leftarrow z/3 \\ escribe\ el\ valor\ de\ z \end{cases} \\ \bigoplus \\ \overline{x > 1} \begin{cases} y \leftarrow y/6 \\ escribe\ el\ valor\ de\ y \end{cases} \end{cases}$$

Por supuesto que el método de Warnier-Orr nos proporciona herramientas para llegar a estos esquemas, que se basan, como dijimos antes, en dejar que los datos nos indiquen las estructuras a través de la jerarquía que guardan entre ellos. Además, utilizando el principio de "divide y vencerás", decimos que cualquier problema puede ser dividido en al menos tres partes: prepararse, ejecutarlo y

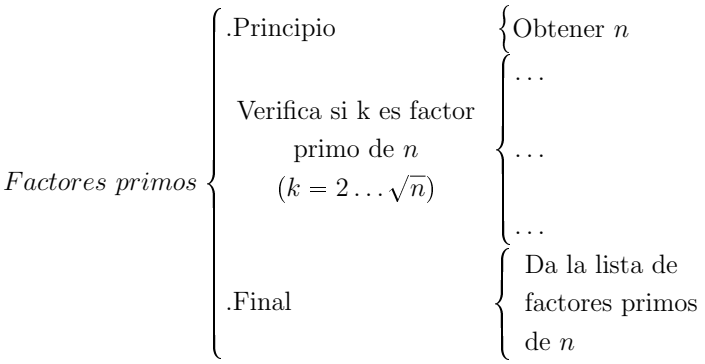
completar o cerrar. Veamos en un ejemplo, el de obtener los factores primos de un entero n, cuáles son los pasos a seguir. Por lo que acabamos de mencionar, todo diagrama de Warnier-Orr empieza con el formato de la figura 2.9.

Figura 2.9 Estado inicial de todo diagrama de Warnier-Orr.



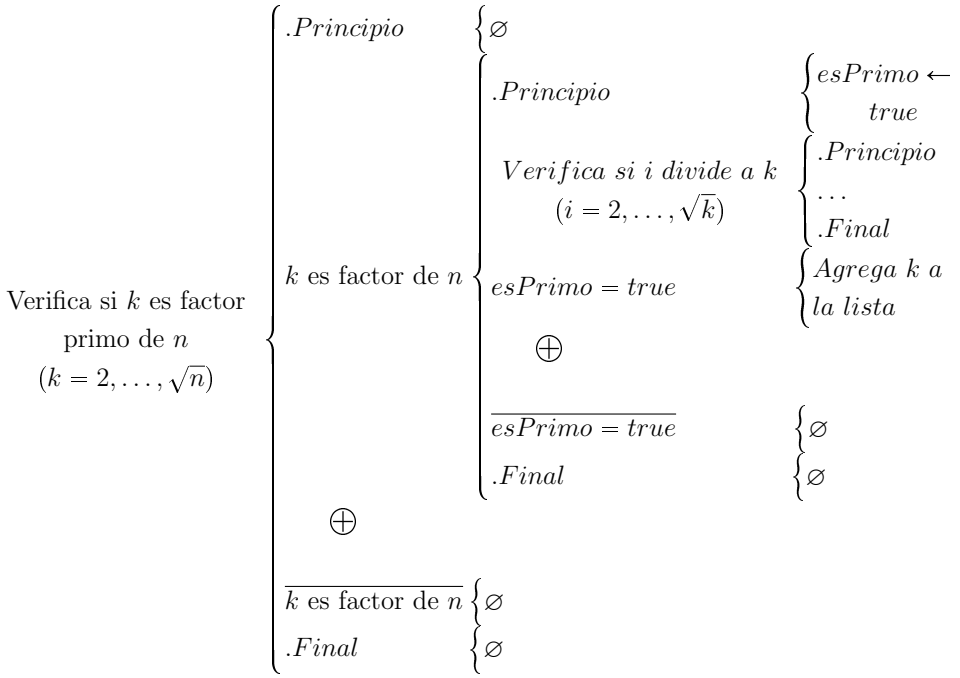
Siguiendo el principio de "ir de lo general a lo particular" indica que intentamos ver el problema desde un punto de vista general, particularizando únicamente cuando ya no es posible avanzar sin hacerlo. En este momento, en el problema que estamos atacando, debemos empezar ya a refinar el proceso. Por último, cuando decimos que vamos desde los resultados hacia los datos, decimos que el problema lo vamos resolviendo preocupándonos por cuál es el resultado que debemos producir y cuál es el paso inmediato anterior para que lo podamos producir.

Figura 2.10 Diagrama inicial para encontrar factores primos.



Para el caso que nos ocupa, determinar los factores primos de un entero, el diagrama con el que empezamos lo podemos ver en la figura 2.10 en la página anterior. En el primer momento, todo lo que sabemos es qué es lo que tenemos de datos (n) y qué esperamos de resultado (una lista con todos los valores de k para los cuales k es primo y k divide a n). Para el principio y final de nuestros procesos podemos observar que lo último que vamos a hacer es proporcionar o escribir la lista de factores primos del número n. Esto corresponde al final de nuestro proceso. Sabemos, porque algo de matemáticas manejamos, que deberemos verificar todos los enteros k entre 2 y $\sqrt{n}$, y que durante este proceso es cuando se debe construir la lista de primos divisores de n. También sabemos, porque corresponde a nuestros datos, que al principio de nuestro proceso deberemos obtener n. Con esta información, podemos producir el diagrama inicial que, como ya mencionamos, se encuentra en la figura 2.10 en la página anterior.

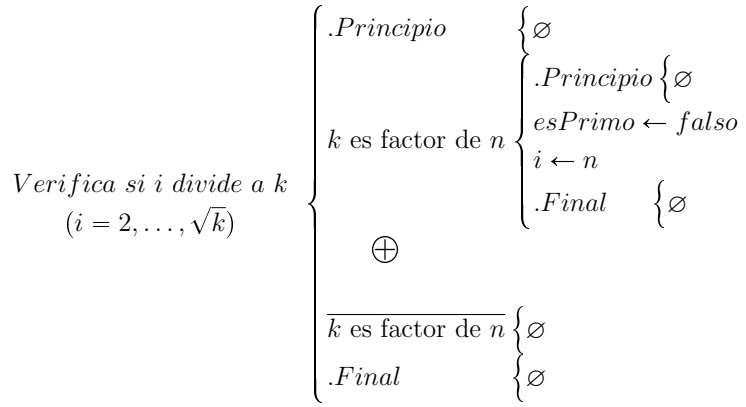
Figura 2.11 Diagrama de Warnier-Orr para procesar cada k.



Progresamos de atrás hacia adelante. Para poder escribir la lista de factores primos, debemos construirla. Esto lo hacemos en el bloque inmediato anterior al del final, que es el que procesa a cada k y decide, primero, si k es un divisor de n, y después, si lo es, si k es primo o no. Si k es primo, se le agrega a la lista. Esta etapa la podemos observar en el diagrama de la figura 2.11 en la página opuesta, que muestra únicamente lo relativo al bloque que maneja a cada k.

Nos falta desarrollar el bloque que corresponde a determinar si k es primo. Esto se hace de manera sencilla, dividiendo a k entre cada i. Si para alguna de estas i's, el residuo es 0, sabemos ya que k no es primo y no vale la pena seguir verificando. Si, en cambio, para ninguna de las i con las que se prueba, la i divide a k, entonces sabemos que k es primo. El diagrama correspondiente a este bloque lo mostramos en la figura 2.12.

Figura 2.12 Diagrama para determinar si k es primo.



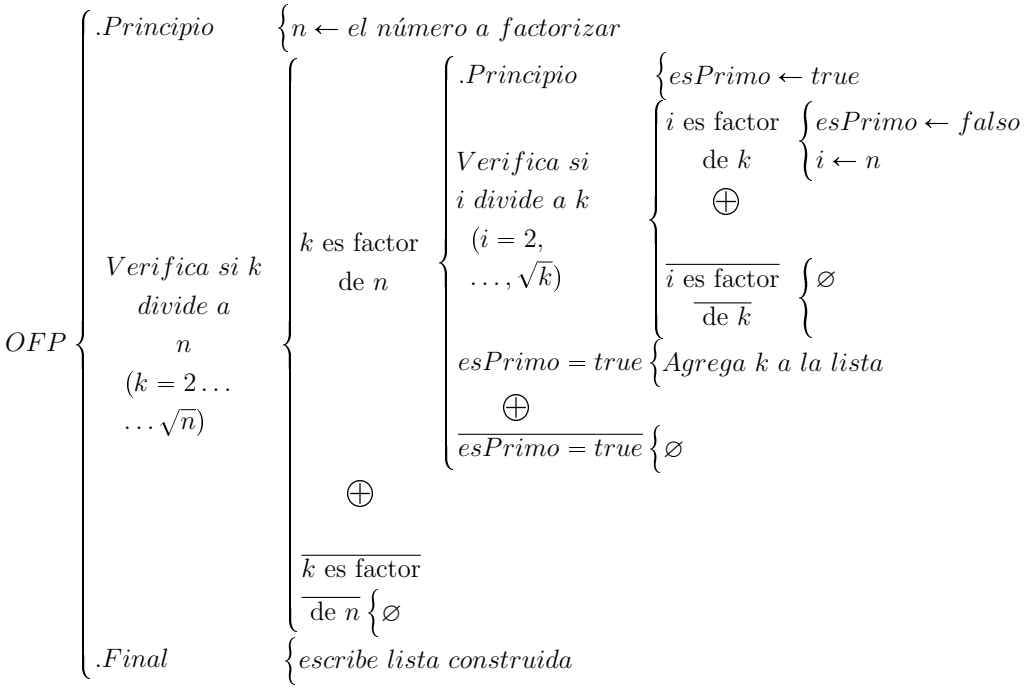
El diagrama completo se puede observar en la figura 2.13 en la siguiente página (para ahorrar espacio, y dado que no contribuyen, eliminamos todos los principios y finales que no hacen nada).

Cuando alguno de los procesos no tiene que hacer nada, simplemente le ponemos un signo de conjunto vacío ($\varnothing$). Varios de los procesos de *Principio* y *Final* no tienen encomendada ninguna tarea, por lo que aparecen vacíos.

Una vez completo el diagrama de Warnier de un problema, el programa está dado realmente por los enunciados en la jerarquía menor (éstos son realmente los enunciados ejecutables), respetando las estructuras de control de repetición y condicionales. Conforme vayamos avanzando en el curso, trataremos de adquirir expe-

riencia en el diseño estructurado. No hay que ignorar el hecho de que la intuición y la experiencia juegan un papel importante en esta etapa. Haciendo el símil con arquitectura, una clase corresponde a los planos de una casa mientras que cada casa que se construye de acuerdo a esos planos corresponde a una instancia u objeto de esa clase. Para poder determinar cada una de estas funciones (no siempre tenemos presentes a todos los tipos) deberemos recurrir al diseño estructurado. Hablaremos más de este tema más adelante en el curso El método de Warnier-Orr no proporciona mecanismos para analizar problemas de naturaleza recursiva. Este tipo de soluciones estará dado por las fórmulas mismas, o la descripción de la solución.

Figura 2.13 Diagrama de Warnier-Orr para obtener factores primos de un entero.



Con esto queda terminado el algoritmo (y los diagramas correspondientes) para obtener factores primos de un entero. Procedemos ahora a incursionar en otros problemas, donde utilizaremos la metodología que acabamos de describir.

Clases y objetos 3

3.1 Tarjetas de responsabilidades

Para ilustrar de mejor manera el proceso de análisis y diseño procederemos a trabajar con un ejemplo concreto: el diseño de un reloj analógico. Este ejemplo, aunque sencillo, nos va a permitir seguirlo hasta su implementación. Los pasos para ello son, como lo mencionamos en secciones anteriores, hacer diagramas o esquemas para cada una de las etapas de definición, que volvemos a listar:

	Acción	**A partir de:**	**Produce:**
1.	Encontrar las clases y los objetos	Una descripción o especificación del problema	Una lista de objetos y un esquema para cada clase
2.	Determinar las responsabilidades	La lista de objetos. La especificación del programa	Esquema de clases con lista de funciones miembro o métodos, clasificados en públicos y privados, y con un breve descripción de qué se supone que hace cada una
3.	Determinar la colaboración entre objetos	La lista de responsabilidades. La lista de objetos.	Adiciona al esquema de responsabilidades, para cada función o método, quién la puede llamar y a quién le va a responder

	Acción	A partir de:	Produce:
4.	Determinar la accesibilidad de las funciones y datos	El esquema de colaboración y responsabilidades	El esquema revisado para que coincida con las reglas que se dieron para accesibilidad.

Pasemos a analizar nuestro problema.

Paso 1: Descripción del problema

Descripción del problema:

Un reloj analógico consiste de una carátula de despliegue, con dos manecillas, una para horas y una para minutos. Cada manecilla tendrá un valor entre 0 y un límite superior prefijado (en el caso de los minutos es 60, mientras que para las horas es 12). El usuario del programa debe ser capaz de construir el reloj inicializando el valor de cada manecilla a las 0 horas con 0 minutos, o bien, a un valor arbitrario (que podría ser el reloj de la máquina). El usuario debe ser capaz también de incrementar el reloj incrementando la manecilla de los minutos y algunas veces también la manecilla de las horas. El usuario deberá ser capaz de establecer el reloj en un cierto valor, estableciendo el valor de cada uno de las manecillas. Finalmente, el usuario puede pedirle al reloj que muestre su valor mostrando la posición de cada una de las manecillas.

Del párrafo anterior, podemos distinguir los siguientes objetos:

reloj

manecilla de horas

manecilla de minutos

valor de horas

valor de minutos

valor del reloj

límites

Podemos ver que el objeto reloj, "posee" dos objetos que corresponde cada uno de ellos a una manecilla. Cada manecilla posee un objeto valor y un objeto límite. El valor concreto de cada manecilla es suficientemente simple como para que se lleve en un entero, lo mismo que el límite; excepto que este último debe ser constante porque una vez que se fije ya no deberá cambiar. Las horas y los minutos

con su valor y límite correspondientes, pertenecen a la misma clase. Podemos entonces mostrar nuestra estructura de la siguiente manera:

Manecilla			
	Datos:	valor	
		límite	
	Funciones:	constructores:	Poner el límite
		incrementa	
		pon valor	
		muestra	
Reloj			
	Datos:	horas	Una manecilla con límite 12
		minutos	Una manecilla con límite 60
	Funciones:	constructores	
		incrementa	
		pon valor	
		muestra	

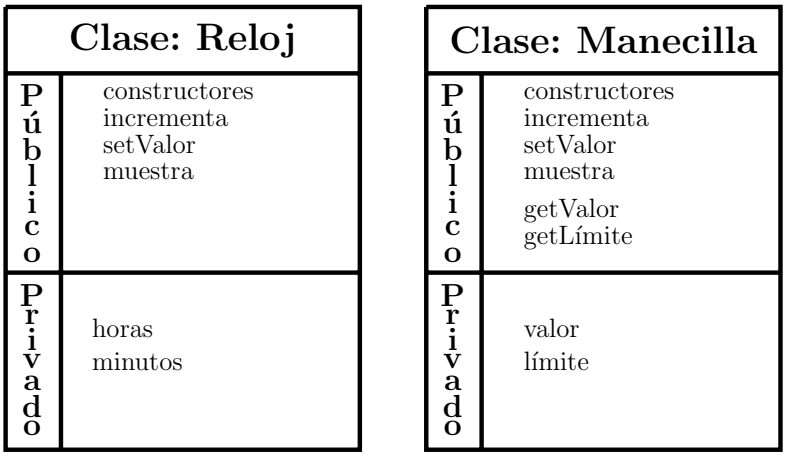
En la clase Manecilla tenemos dos variables, valor y límite, que no queremos puedan ser manipuladas directamente, sin controlar que se mantenga en los límites establecidos, por lo que van a tener acceso privado. Por ello es conveniente agregar dos responsabilidades (métodos) a esta clase, getValor y getLimite, para que se le pueda pedir a los objetos de la clase, en caso necesario, que diga cuánto valen[1].

Paso 2: Elaboración de tarjetas de responsabilidades

En este punto podemos hacer una distribución de los objetos en clases y un esquema de las clases determinadas que presente qué es lo que tiene que contener y tener cada clase, anotándolo en una tarjeta, como se ve en la figura 3.1 en la siguiente página.

[1] Usaremos la palabra get en estos casos, en lugar del término en español da o dame, ya que en Java existe la convención de que los métodos de acceso a los atributos de una clase sean de la forma get seguidos del identificador del atributo empezado con mayúscula. Similarmente en los métodos de modificación o actualización de los valores de un atributo la convención es usar set seguido del nombre del atributo escrito con mayúscula.

Figura 3.1 Tarjetas de clasificación y acceso.



Si1 completamos estos esquemas con las responsabilidades de cada quien, van a quedar como se muestra en la figura 3.2 para la clase Reloj y en la figura 3.3 en la página opuesta para la clase Manecilla.

Figura 3.2 Tarjeta de responsabilidades de la clase **Reloj**

	Clase: Reloj	**(responsabilidades)**
Público	constructor	Inicializa el reloj a una hora dada. Para ello, debe construir las manecillas.
	incrementa	Incrementa el reloj en un minuto
	setValor	Pone un valor arbitrario en el reloj
	muestra	Muestra el reloj
Privado	horas	Registra el valor en horas
	minutos	Registra el valor en minutos

Figura 3.3 Tarjeta de responsabilidades para la clase **Manecilla**.

Clase: Manecilla		(responsabilidades)
Público	constructor	Establece el límite de la manecilla y da valor inicial a la manecilla
	incrementa	Incrementa el valor
	setValor	Pone valor a la manecilla
	muestra	Muestra el valor
	getValor	Dice el valor que tiene
	getLimite	Regresa el valor del límite
Privado	valor	Tiene la información de la manecilla
	limite	Tiene la información respecto al límite

Paso 3: Determinar colaboración

El tercer paso nos dice que determinemos la colaboración entre los objetos. En una primera instancia podemos ver que un objeto de la clase Reloj puede pedirle a cada uno de los objetos de la clase Manecilla que haga su parte: construirse, mostrar su valor, incrementarse. Podemos afinar nuestra descripción de cada una de las clases, describiendo de manera breve en qué consiste cada método o función propia y definiendo la colaboración (quién inicia las acciones, o quién le solicita a quién):

Clase:		Cliente:	Descripción:
Manecilla			
Datos	valor		el valor actual de la manecilla, en el rango $0..límite - 1$.
	límite		el valor en el que el reloj "da la vuelta" o se vuelve a poner en ceros
Métodos	Constructor	Reloj	Pone el valor de la manecilla en ceros y establece el límite
	incrementa	Reloj	Suma 1 al valor y lo regresa a cero si es necesario

Clase:		Cliente:	Descripción:
	setValor	Reloj	Pone el valor
	muestra	Reloj	Muestra el valor que tiene la manecilla
Reloj			
Datos	horas		Una manecilla con límite 12
	minutos		Una manecilla con límite 60
Métodos	Constructor	usuario	Manda un mensaje a ambas manecillas instalando sus límites respectivos
	incrementa	usuario	Incrementa la manecilla de minutos, y si es necesario la de horas
	setValor	usuario	Establece el tiempo en el reloj y para ello lo establece en las manecillas horas y minutos
	muestra	usuario	Pide a las manecillas que se "acomoden"

En forma esquemática las tarjetas quedan como se muestran en las figuras 3.4 y 3.5 en la página opuesta.

Figura 3.4 Tarjeta de colaboraciones de la clase **Manecilla**.

Clase: Manecilla		(colaboración)
Público	constructor	El Reloj a la manecilla
	incrementa	El Reloj a la manecilla
	setValor	El Reloj a la manecilla
	muestra	El Reloj a la manecilla
	getValor	El Reloj a la manecilla
	getLimite	El Reloj a la manecilla
Privado	valor	
	limite	

Figura 3.5 Tarjeta de colaboraciones de la clase **Reloj**.

Clase: Reloj		**(colaboración)**
Público	constructor	El usuario al Reloj
	incrementa	El usuario al Reloj
	setValor	El usuario al Reloj o Reloj a sí mismo
	muestra	El usuario al Reloj
Privado	horas	
	minutos	

Tenemos ya completo el paso de análisis y diseño, ya que tenemos las tarjetas de responsabilidades completas. Pasemos ahora al siguiente paso en la elaboración de un programa, que consiste en la instrumentación del diseño para ser ejecutado en una computadora.

Si bien el diseño orientado a objetos no es un concepto reciente (aparece alrededor de 1972), lo que si es más reciente es la popularidad de las herramientas que facilitan la transición de un modelo orientado a objetos a un programa orientado a objetos. El primer lenguaje que maneja este concepto es Simula (hermanito de Algol 60), aunque su popularidad nunca se generalizó.

Al mismo tiempo que Wirth diseñaba el lenguaje Pascal (una herramienta de programación para el diseño estructurado), se diseñó Smalltalk, un lenguaje orientado a objetos, de uso cada vez más generalizado hoy en día. También se han hecho muchas extensiones a lenguajes "estructurados" para proveerlos de la capacidad de manejar objetos. Entre ellos tenemos Objective Pascal, C++, Objective C, Modula 3, Ada. Muchos de los abogados de la programación orientada a objetos consideran a este tipo de extensiones como "sucias", pues en muchas ocasiones mezclan conceptos, o cargan con problemas que se derivan de tratar de mantener la relación con sus lenguajes originales. Hemos seleccionado Java como herramienta de instrumentación pues contamos con amplia bibliografía al respecto, aceptación generalizada fuera de los ambientes académicos, acceso a muy diversas versiones de la herramienta. Estamos conscientes, sin embargo, de que Java es un lenguaje sumamente extenso, por lo que no pretendemos agotarlo en este curso.

3.2 Programación en Java

En todo lenguaje de programación hay involucrados tres aspectos, relativos a los enunciados escritos en ese lenguaje:

Sintaxis: Se refiere a la forma que tiene que tomar el enunciado. Cada lenguaje tiene sus propias reglas y correspondería a la gramática para un lenguaje natural. Utilizamos para describir la sintaxis lo que se conoce como *BNF extendido*.

Semántica: Se refiere de alguna manera al *significado* del enunciado. Generalmente el significado corresponde a la manera cómo se ejecuta el enunciado, una vez traducido a lenguaje de máquina (en el caso de Java a bytecode). Usaremos lenguaje natural y predicados para describir este aspecto.

Pragmática: Se refiere a restricciones o características dadas por la computadora o la implementación del lenguaje. Por ejemplo, un entero en Java tiene un límite superior e inferior, que no corresponde a lo que entendemos como entero. Este límite es impuesto por la implementación del lenguaje o de la computadora en la que se van a ejecutar los programas. Usaremos lenguaje natural para hablar de la pragmática de un enunciado.

Hablemos un poco de BNF extendido, donde cada enunciado se muestra como si fuera una fórmula:

$$\langle \textit{término a definir} \rangle ::= \langle \textit{expresión regular} \rangle$$

En esta notación del lado izquierdo de "::=" aparece lo que sería un tipo de elementos, lo que vamos a definir, como por ejemplo *acceso*, encerrado entre $\langle$ y $\rangle$ para distinguir al conjunto de alguno de sus representantes. El "::=" se lee "se define como"; del lado derecho se encuentra una $\langle$*expresión regular*$\rangle$, que puede contener a su vez conjuntos o elementos del lenguaje. Una expresión regular es una sucesión de símbolos terminales y no terminales (como en cualquier gramática), pero donde *extendemos* la gramática de la siguiente manera: usamos paréntesis para agrupar – cuando queramos que aparezca un paréntesis tal cual lo marcaremos con negritas –; el símbolo "$|$" para denotar opciones; el símbolo "$*$" para denotar que el grupo anterior se puede repetir cero o más veces; y "+" para denotar que el grupo anterior se puede repetir una o más veces. A los elementos del lenguaje (representantes de los conjuntos, símbolos terminales) los escribimos con negritas, tal como deben aparecer en el archivo fuente. Conforme vayamos avanzando quedará más claro el uso de BNF.

Cuando describamos un recurso del lenguaje, sea éste un enunciado o la manera de organizar a éstos, hablaremos al menos de los dos primeros aspectos; el tercero

lo trataremos en aquellos casos en que tenga sentido.

Como Java es un lenguaje orientado a objetos, la modularidad de los programas en Java se da a través de clases. Una clase es, como ya dijimos, una plantilla para la construcción de objetos, una lista de servicios que los objetos de la clase van a poder realizar.

Otro elemento que utiliza Java para construir sus aplicaciones es la *interfaz*. Una interfaz en Java describe a un grupo de servicios, en términos de lo que los objetos de las clases que la implementen saben hacer, esto es, lista únicamente los servicios que la clase en cuestión va a dar, utilizando qué datos de entrada y proporcionando qué resultados. Una interfaz corresponde a un contrato. Posteriormente podemos construir una o más clases *capaces* de cumplir con ese contrato. A esto último le llamamos *implementar a la interfaz*. Trataremos de trabajar siempre a través de interfaces, pues nos dan un nivel de abstracción más alto que el que nos dan las clases.

Decimos que *declaramos* una interfaz o una clase cuando escribimos la plantilla en un archivo, al que denominamos *archivo fuente*. Se acostumbra, aunque no es obligatorio, que se coloque una clase o interfaz por archivo[2] para tener fácilmente identificable el archivo fuente de la misma. El nombre que se dé al archivo, en este caso, debe coincidir con el nombre de la clase o interfaz. Por ejemplo, la clase Reloj deberá estar en un archivo que se llame Reloj.java; de manera similar, la interfaz ServiciosReloj deberá estar en un archivo que se llame ServiciosReloj.java.

3.2.1. Declaraciones en Java

Lo primero que haremos en Java es, entonces, la definición (declaración) de una interfaz. La sintaxis para ello se puede ver en la figura 3.6 en la siguiente página.

Las palabras que aparecen en negritas tienen que aparecer tal cual. Ése es el caso de **interface** y el **;** que aparece al final de cada ⟨*encabezado de método*⟩. Los que aparecen entre ⟨ y ⟩ deben ser proporcionados por el programador, siguiendo ciertas reglas para ello. En Java el punto y coma (**;**) se usa para *terminar* enunciados, como las declaraciones (los encabezados de un método juegan el papel de una declaración). Por ejemplo, la sintaxis para el ⟨*acceso*⟩ es como se ve en la figura 3.7 en la siguiente página, mientras que un ⟨*identificador*⟩ es cualquier sucesión de letras, dígitos y carácter de subrayado, que empiece con letra o subrayado.

[2]La única restricción real para que haya más de una clase en un archivo es en términos de identificarla, pues no habrá un archivo fuente con el nombre de la clase. Pero sí habrá el archivo correspondiente al bytecode de la clase (*nombre.class*).

Figura 3.6 Encabezado de una interfaz.

SINTAXIS:

⟨ *declaración de interfaz*⟩ ::=
⟨*acceso*⟩ **interface** ⟨*identificador*⟩ **{**
⟨*encabezado de método*⟩**;**
(⟨*encabezado de método*⟩**;**)*
}

SEMÁNTICA:

Se declara una interfaz en un archivo. El nombre del archivo debe tener como extensión `.java` y coincide con el nombre que se le dé a la interfaz. Una interfaz, en general, no tiene declaraciones de atributos, sino únicamente de métodos, de los cuáles únicamente se da el encabezado. Los encabezados de los distintos métodos se separan entre sí por un ; (punto y coma). El que únicamente contenga encabezados se debe a que una interfaz no dice *cómo* se hacen las cosas, sino únicamente *cuáles* cosas sabe hacer.

Figura 3.7 Sintaxis para el ⟨*acceso*⟩.

SINTAXIS:

⟨*acceso* ⟩ ::= **public** | **private** | **protected** | **package** | ∅

SEMÁNTICA:

El acceso a una clase determina quién la puede usar:

public La puede usar todo mundo.

private No tiene sentido para una clase ya que delimita a usar la clase a la misma clase: no se conoce fuera de la clase.

protected Sólo la pueden ver las clases que extienden a ésta. No tiene sentido para clases.

package Sólo la pueden ver clases que se encuentren declaradas en el mismo subdirectorio (paquete). Es el valor por omisión.

Puede no haber regla de acceso, en cuyo caso el valor por omisión es package.

En el caso de las interfaces, el acceso sólo puede ser de paquete o público, ya que el concepto de interfaz tiene que ver con *anunciar* servicios disponibles.

Siguiendo la notación de BNF extendido, el enunciado de Java para denotar a un elemento del conjunto ⟨*identificador*⟩ quedaría como se ve en la figura 3.8.

Figura 3.8 Reglas para la formación de un ⟨*identificador*⟩.

SINTAXIS:

⟨*identificador* ⟩ ::= (⟨*letra* ⟩ | _)(⟨*letra*⟩ | ⟨*dígito*⟩ | _)*

SEMÁNTICA:

Los identificadores deben ser *nemónicos*, esto es, que su nombre ayude a la memoria para recordar qué es lo que representan. No pueden tener blancos insertados. Algunas reglas no obligatorias (aunque exigidas en este curso) son:

Clases: Empiezan con mayúscula y consiste de una palabra descriptiva, como Reloj, Manecilla.

Métodos: Empiezan con minúsculas y se componen de un verbo – da, calcula, mueve, copia – seguido de uno o más sustantivos. Cada uno de los sustantivos empieza con mayúscula.

Variables: Nombres sugerentes con minúsculas.

Constantes: Nombres sugerentes con mayúsculas.

Hay que notar que en Java los identificadores pueden tener tantos carácteres como se desee. El lenguaje, además, distingue entre mayúsculas y minúsculas – no es lo mismo carta que Carta.

Una interfaz puede servir de contrato para más de una clase (que se llamen distinto). Es la clase la que tiene que indicar si es que va a cumplir con algún contrato, indicando que va a implementar a cierta interfaz.

El acceso a los métodos de una interfaz es siempre público o de paquete. Esto se debe a que una interfaz **anuncia** los servicios que da, por lo que no tendría sentido que los anunciara sin que estuvieran disponibles.

Siempre es conveniente poder escribir comentarios en los programas, para que nos recuerden en qué estábamos pensando al escribir el código. Tenemos tres tipos de comentarios:

- Empiezan con `//` y terminan al final de la línea.
- Todo lo que se escriba entre `/*` y `*/`. Puede empezar en cualquier lado y terminar en cualquier otro. Funcionan como separador.
- Todo lo que se escriba entre `/**` y `*/`. Estos comentarios son para JavaDoc, de tal manera que nuestros comentarios contribuyan a la documentación del programa.

Utilizaremos de manera preponderante los comentarios hechos para JavaDoc, en particular para documentar interfaces, clases y métodos. Los comentarios deben tener en el primer renglón únicamente /**, y cada uno de los renglones subsecuentes, menos el último, deberán empezar con un asterisco. En el último renglón aparecerá únicamente */. A partir del segundo renglón deberá aparecer una descripción breve del objetivo de la clase o interfaz.

En el caso de los comentarios de las clases e interfaces, tenemos entre otros un campo, @author, que nos indica quién es el autor de esa clase o interfaz.

La interfaz para nuestro reloj debería anunciar a los servicios que listamos para el reloj en la figura 3.4 en la página 60 – excepto por el constructor –, y la interfaz para la clase Manecilla debe listar los servicios que listamos en la figura 3.5 en la página 61 – también excluyendo al constructor –. Pospondremos por el momento la codificación de los encabezados, hasta que veamos este tema con más detalle. La codificación del encabezado de las interfaces para Reloj y Manecilla se encuentran en los listados 3.1 y 3.2.

Código 3.1 Encabezado de la interfaz para **Reloj** (**ServiciosReloj**)

```
/**
 * Paquete: Reloj.
 * Define los servicios que van a dar los objetos de la clase
 * Reloj.
 */
public interface ServiciosReloj {
  /* Lista de métodos a implementar */
}
```

Código 3.2 Encabezado de la interfaz para **Manecilla** (**ServiciosManecilla**)

```
/**
 * Paquete: Reloj.
 * Define los servicios que van a dar los objetos de la clase
 * Reloj.
 */
public interface ServiciosManecilla {
  /* Lista de métodos a implementar */
}
```

Veamos en la figura 3.9 en la página opuesta la sintaxis y semántica del encabezado de una clase. Ésta es una descripción parcial, ya que por el momento no tiene sentido ver la definición completa.

Figura 3.9 Encabezado de una clase.

SINTAXIS:

⟨*declaración de clase*⟩ ::= ⟨*acceso*⟩ **class** ⟨*identificador*⟩
(∅ |(**implements** (⟨*identificador*⟩)+)
(∅ | **extends** ⟨*identificador*⟩) **{**
⟨*declaraciones*⟩
(∅ | ⟨*método main*⟩)
}

SEMÁNTICA:

Se declara una clase en un archivo. El nombre del archivo debe tener como extensión `.java` y, en general, coincide con el nombre que se le dé a la clase. Una clase debe tener ⟨*declaraciones*⟩ y puede o no tener ⟨*método main*⟩. La clase puede o no adoptar una interfaz para implementar. Si lo hace, lo indica mediante la frase **implements** e indicando a cuál o cuáles interfaces implementa. Las ⟨*declaraciones*⟩ corresponden a los ingredientes o variables y a los métodos que vamos a utilizar. Una ⟨*variable*⟩ corresponde a una localidad (cajita, celda) de memoria donde se va a almacenar un valor. El ⟨*método main*⟩ se usa para poder invocar a la clase desde el sistema operativo. Si la clase va a ser invocada desde otras clases, no tiene sentido que tenga este método. Sin embargo, muchas veces para probar que la clase funciona se le escribe un método main. En Java todo identificador tiene que estar declarado para poder ser usado.

Cuando un archivo que contiene interfaces o clases se compila bien aparecerán en el subdirectorio correspondiente un archivo con el nombre de cada una de las clases o interfaces, pero con el sufijo .class, que es la clase o interfaz pero en bytecode. Éste es el archivo que va a ser interpretado por la Máquina Virtual de Java y el que puede ser ejecutado o invocado.

Vamos codificando lo que ya sabemos cómo. Tenemos dos interfaces, ServiciosReloj y ServiciosManecilla, para los que tenemos que definir los servicios que cada una de ellas va a "contratar". Regresamos a las tarjetas de responsabilidades donde los servicios corresponden a los verbos, y van a ser implementados a través de métodos. Sabemos que hay cinco tipos posibles de métodos:

(a) **Constructores.** Son los que hacen que los objetos de esa clase existan.

(b) **De acceso.** Son los que permiten conocer el estado del objeto.

(c) **De actualización o modificación.** Son los que permiten modificar el estado del objeto.

(d) **De implementación.** Son los que dan los servicios que se requieren del objeto.

(e) **Auxiliares.** Los que requiere el objeto para dar sus servicios de manera adecuada.

Como los métodos involucrados en la interfaz deben ser públicos o de paquete, sólo los de tipo b, c y d van a aparecer en la definición de la interfaz correspondiente. Asimismo, tampoco se ponen en la interfaz a los métodos constructores, pues la interfaz no define ni es capaz de construir objetos. Pospondremos la descripción de los métodos de tipo a y e para cuando revisemos con detalle la definición de clases.

Lo que aparece en la interfaz es únicamente el encabezado de los métodos que van a ser de acceso público o de paquete. Los métodos de actualización o de implementación pueden recibir como entrada datos a los que llamamos parámetros. Los parámetros también se pueden usar para manipulación o para dejar allí información. Un parámetro es, simplemente, una marca de lugar para que ahí se coloquen datos que el método pueda usar y que pueda reconocer usando el nombre dado en la lista. Si regresamos al símil de una obra de teatro, podemos pensar que los parámetros corresponden a la lista de los personajes que viene, adicionalmente, con una descripción de si el personaje es alto, viejo, mujer, etc. (porque el puro nombre no me indica a qué clase de actor contratar para ese papel). El guión viene después en términos de estos personajes: "Hamlet dice o hace". El guión nunca dice quién va a hacer el papel de Hamlet; eso se hace cuando se "monta" la obra. De manera similar con los parámetros, no es sino hasta que se invoca al método que hay que pasar valores concretos. A la lista de parámetros se les llama también *parámetros formales*. Cuando se invoque el método deberán aparecer los "actores" que van a actuar en lugar de cada parámetro. A estos les llamamos los *argumentos* o *parámetros reales*. Daremos la sintaxis de los parámetros cuando aparezcan en alguna definición sintáctica.

En el encabezado de un método cualquiera se localiza lo que se conoce como la *firma* del método, que consiste de los tipos de los parámetros y el nombre del método. Además de la firma, en el método se marca de alguna manera el tipo de método de que se trata. Esto lo revisaremos conforme veamos los distintos tipos de métodos.

Para documentar los distintos métodos de nuestra aplicación utilizaremos también JavaDoc, donde cada comentario empieza y termina como ya mencionamos. En el caso de los métodos, en el segundo renglón deberá aparecer una descripción corta del objetivo del método (que puede ocupar más de un renglón) que debe

terminar con un punto. Después del punto se puede dar una explicación más amplia. A continuación deberá aparecer la descripción de los parámetros, cada uno en al menos un renglón precedido por @param y el nombre del parámetro, con una breve explicación del papel que juega en el método. Finalmente se procederá a informar del valor que regresa el método, precedido de @returns y que consiste de una breve explicación de qué es lo que calcula o modifica el método.

Métodos de acceso

Los métodos de acceso los tenemos para que nos informen del estado de un objeto, esto es, del valor de alguno de los atributos del objeto. Por ello, la firma del método debe tener información respecto al tipo del atributo que queremos observar. La sintaxis se puede ver en la figura 3.10, donde las definiciones de ⟨*tipo* ⟩, ⟨*acceso*⟩ e ⟨*identificador*⟩ son como se dieron antes.

Figura 3.10 Encabezado para los métodos de acceso.

SINTAXIS:

⟨ *encabezado demétodo de acceso* ⟩ ::=
⟨*acceso*⟩ ⟨*tipo*⟩ ⟨*identificador*⟩ **(** ⟨*Parámetros* ⟩**)**

SEMÁNTICA:

La declaración de un método de acceso consiste del tipo de valor que deseamos ver, ya que nos va a "regresar" un valor de ese tipo, seguido de la firma del método, que incluye a los ⟨*Parámetros*⟩. El identificador del método es arbitrario, pero se recomienda algo del estilo "getAtributo", que consista de un verbo que indica lo que se va a hacer, y un sustantivo que indique el atributo que se busca o el valor que se desea calcular.

Los tipos que se manejan en Java pueden ser primitivos o de clase. Un tipo primitivo es aquél cuyas variables no son objetos y son atómicos, esto es, no se subdividen en otros campos o atributos. En la tabla 3.3 en la siguiente página se encuentra una lista, con los tipos primitivos y sus rangos.

Otro tipo de dato que vamos a usar mucho, pero que corresponde a una clase y no un dato primitivo como en otros lenguajes, son las cadenas, sucesiones de carácteres. La clase se llama String. Las cadenas (String) son cualquier sucesión de carácteres, menos el de fin de línea, entre comillas. Los siguientes son objetos tipo String:

```
"Esta␣es␣una␣cadena␣1␣2␣3␣"
""
```

Cuadro 3.3 Tipos primitivos y sus rangos.

Identificador	Capacidad
boolean	true o false
char	16 bits, Unicode 2.0
byte	8 bits con signo en complemento a 2
short	16 bits con signo en complemento a 2
int	32 bits con signo en complemento a 2
long	64 bits con signo en complemento a 2
float	32 bits de acuerdo al estándar IEEE 754-1985
double	64 bits de acuerdo al estándar IEEE 754-1985

La primera es una cadena común y corriente y la segunda es una cadena *vacía*, que no tiene ningún carácter.

La operación fundamental con cadenas es la concatenación, que se representa con el operador +. Podemos construir una cadena concatenando (sumando) dos o más cadenas:

```
"a"+"b"+"c"                          "abc"
"Esta␣cadena␣es"+"muy␣bonita"       "Esta␣cadena␣esmuy␣bonita"
```

Una de las ventajas del operador de concatenación de cadenas es que fuerza a enteros a convertirse en cadenas cuando aparecen en una expresión de concatenación de cadenas. Por ejemplo, si tenemos una variable LIM que vale 12, tenemos lo siguiente:

```
"El␣límite␣es:␣"+LIM+"."      "El␣límite␣es:␣12."
```

Hablaremos más, mucho más, de la clase String más adelante.

En la figura 3.11 damos la sintaxis y semántica para la declaración de los ⟨*Parámetros*⟩.

Cuando invocamos a un método le pasamos el número y tipo de argumentos que define su firma, en el orden definido por la firma. A esto se le conoce como *paso de parámetros*. En general se definen tres objetivos en el paso de parámetros: parámetros de entrada, parámetros de salida y parámetros de entrada y salida. A estos tres tipos se asocian mecanismos de paso de parámetros, entre los que están el paso por valor, donde se evalúa al argumento y se pasa nada más unas copia de ese valor; el paso por referencia, donde se toma la dirección en memoria y eso

es lo que se pasa como argumento; la evaluación perezosa, donde no se evalúa el argumento hasta que se vaya usar dentro de la implementación del método; dependiendo de qué tipo de paso de parámetros contemos, es el efecto que vamos a tener sobre los argumentos. En el caso de Java todos los argumentos se pasan por valor, incluyendo a las referencias a los objetos. Aclararemos más este punto cuando lo enfrentemos.

Figura 3.11 Especificación de parámetros.

SINTAXIS:

⟨*Parámetros*⟩::= ∅ |
⟨*parámetro*⟩(, ⟨*parámetro*⟩)*

⟨*parámetro*⟩ ::= ⟨*tipo*⟩ ⟨*identificador*⟩

SEMÁNTICA:

Los parámetros pueden estar ausentes o bien consistir de una lista de parámetros separados entre sí por coma (,). Un parámetro marca lugar y tipo para la información que se le dé al método. Lo que le interesa al compilador es la lista de tipos (sin identificadores) para identificar a un método dado, ya que se permite más de un método con el mismo nombre, pero con distinta firma.

Por ejemplo, los métodos de una Manecilla que dan los valores de los atributos privados tienen firmas como se muestra en el listado 3.3. En general, podemos pedirle a cualquier método que regrese un valor, y tendría entonces la sintaxis de los métodos de acceso. Como lo que queremos del Reloj es que se muestre, no que nos diga qué valor tiene, no tenemos ningún método de acceso para esta clase.

Código 3.3 Métodos de acceso para los atributos privados de **Manecilla** (**ServiciosManecilla**)

```
interface ServiciosManecilla {
    ...
    public int getValor ();
    public int getLimite ();
    ...
} // ServiciosManecilla
```

Métodos de implementación

Estos métodos son los que dan los servicios. Por ello, el método muestra cuya firma aparece en el listado 3.4 es de este tipo. Es común que este tipo de métodos regresen un valor que indiquen algún resultado de lo que hicieron, o bien que simplemente avisen si pudieron o no hacer lo que se les pidió, regresando un valor booleano. En el caso de que sea seguro que el método va a poder hacer lo que se le pide, sin contratiempos ni cortapisas, se indica que no regresa ningún valor, poniendo en lugar de ⟨tipo ⟩la palabra void. Por ejemplo, el encabezado del método que muestra la Manecilla queda como se muestra en el listado 3.4. También en el listado 3.5 mostramos el método de implementación muestra para la interfaz ServiciosReloj.

Código 3.4 Métodos de implementación de la interfaz **ServiciosManecilla**

```
interface ServiciosManecilla {
  ...
  public void muestra ();
  ...
} // ServiciosManecilla
```

Código 3.5 Métodos de implementación de la interfaz **ServiciosReloj**

```
interface ServiciosReloj {
  public void muestra ();
} // Reloj
```

Como pueden ver, ninguno de estos dos métodos regresa un valor, ya que simplemente hace lo que tiene que hacer y ya. Tampoco tienen ningún parámetro, ya que toda la información que requerirá es el estado del objeto, al que tienen acceso por ser métodos de la clase.

Métodos de manipulación

Los métodos de manipulación son, como ya mencionamos, aquellos que cambian el estado de un objeto. Generalmente tienen parámetros, pues requieren información de cómo modificar el estado del objeto. Los métodos que incrementan y que asignan un valor son de este tipo, aunque el método que incrementa no requiere de parámetro ya que el valor que va a usar es el 1. Muchas veces queremos que también nos proporcionen alguna información respecto al cambio de estado,

como pudiera ser un valor anterior o el mismo resultado; también podríamos querer saber si el cambio de estado procedió sin problemas. En estos casos el método tendrá valor de regreso, mientras que si no nos proporciona información será un método de tipo void. Por ejemplo, el método que incrementa a la Manecilla nos interesa saber si al incrementar llegó a su límite. Por ello conviene que regrese un valor de 0 si no llegó al límite, y de 1 si es que llegó (dio toda una vuelta). La firma de este método se muestra en los listados 3.6 y 3.7.

Código 3.6 Métodos de manipulación para la interfaz **ServiciosManecilla**

```
interface ServiciosManecilla {
  /** Incrementa a la manecilla en una unidad.
   *  @return 0 si no llegó al límite y 1 si llegó */
  public int incrementa ();
  /** Establece un valor arbitrario en esta manecilla.
   *  @param val El valor que se desea establecer. */
  public void setValor (int val);
} // ServiciosManecilla
```

Código 3.7 Métodos de Manipulación para la interfaz **ServiciosReloj**

```
interface ServiciosReloj {
  /**
   * Incrementa este reloj en un minuto.
   */
  public void incrementa ();
  /**
   * Establece el valor del horario y el minutero.
   * @param hrs  el valor a establecer en el horario.
   * @param mins el valor a establecer en el minutero. */
  public void setValor (int hrs, int mins);
} // ServiciosReloj
```

Noten que el método setValor de ServiciosManecilla tiene un parámetro, que es el nuevo valor que va a tomar la manecilla, mientras que el método con el mismo nombre de la clase ServiciosReloj tiene dos parámetros, ya que requiere los valores para las horas y para los minutos.

Procedemos a construir las clases que implementen a cada una de estas interfaces. Identificamos dos clases en nuestro sistema, Manecilla y Reloj. Como la clase Manecilla no se usará más que dentro de Reloj, la ponemos en el mismo archivo que a Reloj, pero dejando a ésta como la primera. El archivo se llamará Reloj.java

Código 3.8 Encabezados para la implementación de **Reloj** y **Manecilla**

```
/**
 * Clase para representar a un Reloj analógico
 * El Reloj debe ser capaz de mantener la hora,
 * incrementarse minuto por minuto, y ponerse
 * a una cierta hora
 */
public class Reloj implements ServiciosReloj    {
    /*  Acá va lo correspondiente a declaraciones de Reloj */
  public void incrementa ()   {
    /* Acá va la implementación de incrementa */
  } // incrementa
  public void setValor (int hrs, int mins)  {
    /* Acá va la implementación. */
  } // setValor
  public void muestra ()   {
    /**  Acá va la implementación. */
  } // muestra
} // Reloj
//Encabezados para la implementación de Manecilla
  /**
   * Una manecilla debe ser capaz de incrementarse,
   * y poder servir tanto para horas, como minutos
   * y segundos
   */
class Manecilla implements ServiciosManecilla   {
  /** Acá va lo correspondiente a las declaraciones
   * de la clase Manecilla
   */
  public int getValor()   {
    /** Acá va la implementación
     */
  } // getValor
  public int getLimite()  {
    /** Acá va la implementación
     */
  } // getLimite
  public void muestra ()   {
    /** Acá va la implementación.
     */
  } // muestra
} // Manecilla
```

porque éste es el objetivo principal del programa. Veamos cómo queda lo que llevamos del programa en el listado 3.8 (omitimos los comentarios de JavaDoc para

ahorrar algo de espacio). Como declaramos que nuestras clases Reloj y Manecilla implementan, respectivamente, a las interfaces ServiciosReloj y ServiciosManecilla, estas clases tendrán que proporcionar las implementaciones para los métodos que listamos en las interfaces correspondientes. El esqueleto construido hasta ahora se puede ver en el listado 3.8. Como a la clase Manecilla únicamente la vamos a utilizar desde la clase Reloj no le damos un archivo fuente independiente.

De las cinco variedades de métodos que listamos, nos falta revisar a los métodos constructores y a los métodos auxiliares, que tienen sentido sólo en el contexto de la definición de clases.

Métodos auxiliares

Estos métodos son aquellos que auxilian a los objetos para llenar las solicitudes que se les hacen. Pueden o no regresar un valor, y pueden o no tener parámetros: depende de para qué se vayan a usar. Dado que el problema que estamos atacando por el momento es relativamente simple, no se requieren métodos auxiliares para las clases.

En cuanto a las interfaces no se presentan otra categoría de métodos, ya que las interfaces no describen objetos, sino únicamente los contratos a los que se obligan las clases que los implementen.

Métodos constructores

Una clase es un patrón (descripción, modelo, plano) para la construcción de objetos que sean ejemplares (*instances*) de esa clase. Por ello, las clases sí tienen constructores que determinan el estado inicial de los objetos construidos de acuerdo a esa clase.

En Java los métodos constructores tienen una sintaxis un poco distinta a la de otros tipos de métodos. Ésta se puede ver en la figura 3.12 en la siguiente página.

Un constructor es el que permite el instanciamiento (la construcción) de un objeto de una clase dada, para que sea asociado a (referido por) una variable de ese tipo. Su objetivo principal es el de establecer el estado inicial del objeto (inicializar los atributos). Puede recibir para la construcción datos en la forma de parámetros.

Figura 3.12 Encabezado de un constructor.

SINTAXIS:

⟨*constructor*⟩ ::=⟨*acceso*⟩ ⟨*identificador de Clase*⟩ **(** ⟨*Parámetros*⟩ **) {**
⟨*implementación*⟩
}

⟨*Parámetros*⟩ ::=⟨*parámetro*⟩(, ⟨*parámetro*⟩)* | ∅

⟨*parámetro*⟩ ::=⟨*tipo*⟩ ⟨*identificador*⟩

SEMÁNTICA:

Los constructores de una clase son métodos que consisten en un acceso – que puede ser cualquiera de los dados anteriormente – seguido del nombre de la clase y entre paréntesis los ⟨*Parámetros*⟩ del método. Un parámetro corresponde a un dato que el método tiene que conocer (o va a modificar). Cada parámetro deberá tener especificado su tipo. Los nombres dados a cada parámetro pueden ser arbitrarios, aunque se recomienda, como siempre, que sean nemónicos y no se pueden repetir.

Código 3.9 Constructores para la clase Reloj

```
  /**   ... */
public class Reloj implements ServiciosReloj  {
  /** ...                                  */
  /**
   * Constructor.
   * @param limH  límite para las horas.
   * @param limM  límite para los minutos.
   */
    Reloj (int limH, int limM) {
        /* Constructor: establece los límites */
        // ⟨Implementación⟩
    } // Firma: Reloj (int, int)
  /**
   * Constructor.
   * @param limH  Límite para las horas.
   * @param limM  Límite para los minutos
   * @param hrs   Horas actuales.
   * @param mins  Minutos actuales.
   */
    Reloj (int limH, int limM, int hrs, int mins) {
        //  ⟨Implementación⟩
    } // Firma: Reloj (int, int, int, int)
} // Reloj
```

Podemos tener tantos constructores como queramos, siempre y cuando se distingan por sus firmas. Por ejemplo, podemos tener un constructor que no tenga parámetros, o uno que tenga otra organización con sus parámetros. Un segundo constructor para Manecilla pudiera ser uno que establece un valor prefijado para la manecilla. Algo similar podemos hacer con la clase Reloj y lo podemos ver en el listado 3.10. Los constructores *siempre* tienen el mismo nombre que la clase de la que son constructores. No es necesario decir qué tipo de valor regresan, porque "regresan" (construyen) a un objeto de su clase.

Código 3.10 Constructores para la clase Manecilla

```
/** ... */
public class Manecilla implements ServiciosManecilla {
  /** ... */
   /** Constructor.
    * @param lim Cota superior para el valor que puede tomar la
    *            manecilla.
    */
   Manecilla(int lim) {
     // ⟨Implementación⟩
   } // Firma: Manecilla (int)

   /** Constructor que establece la hora actual.
    * @param lim  Cota superior para el valor de la manecilla.
    */
   Manecilla(int lim, int val)\ {
     /* Constructor: pone valor máximo y  valor inicial */
     // ⟨Implementación⟩
   } // Firma: Manecilla (int, int)

   ...
} // Manecilla
```

Es importante notar que la firma de un método consiste únicamente del nombre del método junto con los tipos de los parámetros. Por lo tanto, los dos encabezados que se encuentran en el listado 3.11 en la siguiente página tienen la misma firma, y el compilador daría un mensaje de método duplicado, aunque el nombre de los parámetros sea distinto.

Código 3.11 Métodos con la misma firma

```
public Reloj (int hrs, int mins, int limH, int limM)
   // Firma: Reloj(int, int, int, int)

public Reloj(int lim1, int lim2, int val1, int val2)
   // firma: Reloj(int, int, int, int)
```

Toda clase tiene un constructor por omisión, sin parámetros, que puede ser invocado, siempre y cuando no se haya declarado ningún constructor para la clase. Esto es, si se declaró, por ejemplo, un constructor con un parámetro, el constructor sin parámetros ya no está accesible. Por supuesto que el programador puede declarar un constructor sin parámetros que sustituya al que proporciona Java por omisión.

El estado inicial que da el constructor por omisión al objeto es, básicamente, cero en los atributos numéricos, falso en los atributos lógicos y referencia nula (null) en los atributos que son objetos.

Atributos

Antes de definir la implementación de los métodos trabajemos con los atributos que dimos en las tarjetas. Los sustantivos deberán ser atributos (variables o constantes) – datos – mientras que los verbos fueron métodos – procesos o cálculos. Lo primero que tenemos que hacer es lugar para los objetos o datos primitivos que se encuentran en cada clase. Esto lo hacemos mediante una declaración. En la declaración especificamos el nombre que le queremos dar al atributo – ya sea objeto o primitivo – y el tipo que va a tener – entero, tipo Manecilla, etc. Veamos la sintaxis y semántica de una declaración de atributo (dato, campo) en la figura 3.13 en la página opuesta.

Declaremos los atributos que se presentan en el programa que estamos armando en el listado 3.12 en la página 80 – omitimos los comentarios ya presentados para ahorrar un poco de espacio –.

Figura 3.13 Declaración de un atributo

SINTAXIS:

⟨*declaración de atributo*⟩ ::=⟨*acceso*⟩ ⟨*modificador*⟩ ⟨*tipo* ⟩
⟨*identificador* ⟩(,⟨*identificador*⟩)*;

⟨*modificador* ⟩ ::=**final** | **static** | ∅

⟨*tipo* ⟩ ::=⟨*tipo primitivo*⟩ | ⟨*identificador de clase* ⟩

SEMÁNTICA:

Todo identificador que se declara, como con el nombre de las clases, se le debe dar el ⟨*acceso*⟩ y si es constante (final) o no. Por el momento no hablaremos de static. También se debe decir su tipo, que es de alguno de los tipos primitivos que tiene Java, o bien, de alguna clase a la que se tenga acceso; lo último que se da es el identificador. Se puede asociar una lista de identificadores separados entre sí por una coma, con una combinación de acceso, modificador y tipo, y todas las variables de la lista tendrán las mismas características. Al declararse un atributo, el sistema de la máquina le asigna una *localidad*, esto es, un espacio en memoria donde guardar valores del tipo especificado. La cantidad de espacio depende del tipo. A los atributos que se refieren a una clase se les reserva espacio para una *referencia*, que es la posición en el *heap* donde quedará el objeto que se asocie a esa variable[3].

Las declaraciones de las líneas 5:, 7:, 13: y 15: son declaraciones de atributos del tipo que precede al identificador. En la línea 5: se están declarando dos atributos de tipo *Manecilla* y acceso privado, mientras que en la línea 13: se está declarando un atributo de tipo entero y acceso privado. En la línea 15: aparece el modificador final, que indica que a este atributo, una vez asignado un valor por primera vez, este valor ya no podrá ser modificado. Siguiendo las reglas de etiqueta de Java, el identificador tiene únicamente mayúsculas. En el caso de los atributos de tipo Manecilla, debemos tener claro que nada más estamos declarando un atributo, no **el objeto**. Esto quiere decir que cuando se construya el objeto de tipo Manecilla, la variable horas se referirá a este objeto, esto es, contendrá una *referencia* a un objeto de tipo Manecilla. Como los objetos pueden tener muy distintos tamaños sería difícil acomodarlos en el espacio de ejecución del programa, por lo que se construyen siempre en un espacio de memoria destinado a objetos, que se llama *heap*[4], y la variable asociada a ese objeto nos dirá la dirección del mismo en el heap.

[4]El valor de una referencia es una dirección del heap. En esa dirección se encuentra el objeto construido.

Código 3.12 Declaración de atributos de las clases

```
1:     /** ...
2:     */
3: public class Reloj implements ServiciosReloj {
4:   /** Recuerda la hora del reloj.                */
5:     private Manecilla horas,
6:   /** Recuerda los minutos del reloj.            */
7:                       minutos;
8: } // Reloj
9:   /** ...
10:   */
11: public class Manecilla implements ServiciosManecilla {
12:   /** Recuerda el valor de la manecilla.         */
13:     private int valor;
14:   /** Da una cota superior para el valor de la manecilla. */
15:     private final int LIM;
16:                 // Se establece al construirla
17: } // Manecilla
```

El método `main`

El método main corresponde a la colaboración que queremos se dé entre clases. En él se define la lógica de ejecución. No toda clase tiene un método main, ya que no toda clase va a definir una ejecución. A veces pudiera ser nada más un recurso (como es el caso de la clase Manecilla). El sistema operativo (la máquina virtual de Java) reconoce al método main y si se "invoca" a una clase procede a ejecutar ese método. El encabezado para este método se encuentra en el listado 3.13.

Código 3.13 Encabezado para el método **main**

```
    public static void main(String[] args) {
        // ⟨implementación⟩
    } // main
```

Se ve bastante complicado, aunque no lo es tanto. El significado de void y public ya lo sabemos. Lleva el modificador static porque sólo debe haber un método de éstos para la clase, esto es, para todos los objetos de la clase. Finalmente, el parámetro que tiene es un arreglo (denotado por [ ]) de cadenas (String) que son las cadenas que aparecen en la línea de comandos cuando se invoca desde el sistema operativo (un arreglo es simplemente una sucesión de datos). La implementación de este método es como la de cualquier otro. Cuando veamos implementación en general daremos las restricciones que presenta.

3.2.2. Alcance de los identificadores

Para estos momentos ya tenemos bastantes nombres en el programa; algunos se repiten, como el nombre de la clase en el nombre de los constructores o los identificadores de los parámetros en métodos distintos. Es importante saber, cada nombre, qué alcance tiene, esto es, desde donde puede el programa referirse a él.

Figura 3.14 Acceso a atributos o métodos de objetos

SINTAXIS:

⟨*Referencia a atributo o método*⟩::=
(⟨*referencia de objeto o clase*⟩**.**)$^+$(⟨*id de atributo*⟩ |
⟨*invocación a método*⟩)

SEMÁNTICA:

El operador **.** es el de selector, y asocia de izquierda a derecha. Lo usamos para identificar, el identificador que se encuentra a su derecha, de qué objeto forma parte. También podemos usarlo para identificar a alguna clase que pertenezca a un paquete. En el caso de un identificador de método, éste deberá presentarse con los argumentos correspondientes entre paréntesis. la ⟨*referencia de objeto*⟩ puede aparecer en una variable o como resultado de una función que regrese como valor una referencia, que se encuentre en el alcance de este enunciado. Podemos pensar en el **.** como un operador del tipo *referencia*.

La pista más importante para esto son las parejas de llaves que abren y cierran. Para las que corresponden a la clase, todos los nombres que se encuentran en las declaraciones dentro de la clase son accesibles desde cualquier método de la misma clase. Adicionalmente, los nombres que tengan acceso público o de paquete son accesibles también desde fuera de la clase.

Sin embargo, hemos dicho que una clase es nada más una plantilla para construir objetos, y que cada objeto que se construya va a ser construido de acuerdo a esa plantilla. Esto quiere decir que, por ejemplo en el caso de la clase Manecilla, cada objeto que se construya va a tener su atributo valor y su atributo LIM. Si éste es el caso, ¿cómo hacemos desde fuera de la clase para saber de cuál objeto estamos hablando? Muy fácil: anteponiendo el nombre del objeto al del atributo, separados por un punto. Veamos la forma precisa en la figura 3.14.

Si tenemos en la clase Reloj dos objetos que se llaman horas y minutos, podremos

acceder a sus métodos públicos, como por ejemplo incrementa como se muestra en el listado 3.14.

Código 3.14 Acceso a atributos de los objetos

```
horas.incrementa()
minutos.incrementa()
```

Es claro que para que se puedan invocar estos métodos desde la clase Reloj deben tener acceso público o de paquete. También los objetos horas y minutos tienen que ser conocidos dentro de la clase Reloj.

Sin embargo, cuando estamos escribiendo la implementación de algún método, al referirnos, por ejemplo, al atributo valor no podemos saber de cuál objeto, porque el método va a poder ser invocado desde cualquier objeto de esa clase. Pero estamos asumiendo que se invoca, forzosamente, con algún objeto. Entonces, para aclarar que es el atributo valor del objeto con el que se está invocando, identificamos a *este* objeto con this. Cuando no aparece un identificador de objeto para calificar a un atributo, dentro de los métodos de la clase se asume entonces al objeto this. En el código que sigue las dos columnas son equivalentes para referirnos a un atributo *dentro de un método de la clase*.

this.incrementa()	incrementa()
this.valor	valor
this.horas.LIM	horas.LIM

En cuanto a los parámetros de un método, éstos existen sola y exclusivamente dentro de la implementación del método, entre las llaves. Son lo que se conoce como *nombres o variables locales*, locales al método. Si en la clase existe algún atributo cuyo nombre sea el mismo que el del parámetro, el nombre del parámetro *bloquea* al nombre del atributo, y para referirse al atributo dentro del método se tendrá que usar al selector this – véase el listado 3.15 en la página opuesta.

En este listado el atributo horas de la clase Reloj se ve "bloqueado" por el parámetro horas, por lo que para poder ver al atributo hay que rescatar que se trata del atributo del objeto con el que se esté invocando al método.

Código 3.15 Bloqueo de nombres de atributos

```
public class Reloj implements ServiciosReloj  {
    Manecilla horas,
              minutos;
    public void setValor(int horas, int minutos)    {
        this.horas.setValor(horas);
        this.minutos.setValor(minutos);
    } // Reloj.setValor
  ...
} // class Reloj
```

3.2.3. Implementación de métodos en Java

La implementación de cada uno de los métodos nos va a decir el "cómo" y "con quién" va a cubrir el objeto ese servicio.

Figura 3.15 Sintaxis para la implementación de un método

SINTAXIS:

⟨*implementación*⟩ ::=⟨Lista de enunciados⟩
⟨*Lista de enunciados*⟩ ::=⟨*enunciado* ⟩ ⟨*Lista de enunciados*⟩ | ∅
⟨*enunciado*⟩ ::=⟨*enunciado simple* ⟩**;** | ⟨*enunciado compuesto* ⟩
⟨*enunciado simple*⟩ ::=⟨*declaración local*⟩ | ⟨*invocación de método* ⟩
| ⟨*enunciado de asignación*⟩
| **return** | **return** ⟨*expresión*⟩

SEMÁNTICA:

La implementación de un método, no importa de cual categoría sea, consiste de una lista de enunciados entre llaves. Si queremos que el método no haga nada, entonces no ponemos ningún enunciado entre las llaves. Los enunciados pueden ser simples o compuestos[5]. Un enunciado simple puede ser una invocación a un método, puede ser una declaración de variables o puede ser una asignación de valor a una variable. Noten que todos los enunciados simples terminan con punto y coma – **;** – sin importar el contexto en el que aparecen. Es importante mencionar que aquellas variables declaradas en la implementación de un método, así como los parámetros formales, van a ser accesibles (reconocidas) únicamente dentro de la implementación del método en cuestión, a diferencia de las variables de la clase, atributos, que van a ser accesibles (reconocidos) en las implementaciones de cualquiera de los métodos de la clase.

En todo lo que llevamos hasta ahora simplemente hemos descrito los ingredientes de las clases y no hemos todavía manejado nada de cómo hacen los métodos lo que tienen que hacer. En general un método va a consistir de su encabezado y una lista de *enunciados* entre llaves, como se puede ver en la figura 3.15 en la página anterior.

Las declaraciones

Cuando estamos en la implementación de un método es posible que el método requiera de objetos o datos primitivos auxiliares dentro del método. Estas variables auxiliares se tienen que declarar para poder ser usadas. El alcance de estas variables es únicamente entre las llaves que corresponden al método. Ninguna variable se puede llamar igual que alguno de los parámetros del método, ya que si así fuera, como los parámetros son locales se estaría repitiendo un identificador en el mismo alcance. La sintaxis para una declaración se puede ver en la figura 3.16.

Figura 3.16 Declaración de variables locales

SINTAXIS:

⟨*declaración de variable local*⟩ ::= ⟨*tipo*⟩ ⟨*Lista de identificadores*⟩;

SEMÁNTICA:

La declaración de variables locales es muy similar a la de parámetros formales, excepto que en este caso sí podemos declarar el tipo de varios identificadores en un solo enunciado. La ⟨*Lista de identificadores*⟩ es, como su nombre lo indica, una sucesión de identificadores separados entre sí por una coma (“,”).

Hay que notar que localmente únicamente se pueden declarar variables, ya sea de tipo primitivo o referencia, y son conocidas, al igual que los parámetros, únicamente dentro del método en el que se están declarando. No se puede declarar una variable que repita algún identificador usado para los parámetros, ya que los parámetros también se comportan, dentro del método, como variables locales. Al terminar la ejecución del método, estas variables desaparecen.

El único método que requiere de variables auxiliares es el que muestra el reloj, ya que queremos construir unas cadenas de carácteres para que den el mensaje de qué hora es. También requerimos de algún objeto que haga de dispositivo de salida, para poder mostrar *ahí* la hora del reloj; en otras palabras, necesitamos poder hacer entrada y salida.

[5] No entraremos por ahora a lo que es un enunciado compuesto, ya que todavía no los vamos a usar.

La entrada y salida de Java es un poco complicada para ser manejada por principiantes. Por ello, proporcionamos, al menos en una primera parte del curso, una clase Consola que va a permitir hacer entrada y salida de distintos objetos y datos primitivos de Java – consultar la documentación proporcionada por el ayudante. Por el momento, todo lo que tenemos que hacer es declarar un objeto tipo Consola para que podamos ahí mostrar el estado del reloj. Podemos ver estas declaraciones en el listado 3.16.

Código 3.16 Declaraciones locales en el método **muestra** de **Reloj**

```
1: public class Reloj implements ServiciosReloj  {
2:     ...
3:   /**
4:    * Muestra el estado de este reloj.
5:    */
6:     public void muestra ()    {
7:          String mensaje1, mensaje2, mensaje3;
8:          Consola consola;
9:          ...

100: } // muestra
101:    ...
102: } // class Reloj
```

Tenemos un pequeño problema con la clase Consola y es la manera en que nuestro programa la va a identificar. La va a encontrar sin problemas si se encuentra en el mismo subdirectorio que nuestras clases, o bien si se encuentra en alguno de los paquetes de Java. Pero no sucede ni una cosa ni la otra. Por lo tanto, tendremos que avisarle a Java que vamos a usar esta clase y el paquete en donde se encuentra. Esto se hace mediante la directiva import, seguida de toda la ruta que nos puede llevar a esta clase. Por ejemplo, en mi máquina, mis clases están en un subdirectorio progs y la clase Consola con todo lo que necesita se encuentra en un subdirectorio icc1/interfaz, por lo que el enunciado import – que va al inicio del archivo donde está la clase que lo usa – sería, cambiando las diagonales por puntos selectores,

```
import icc1.interfaz.Consola;
```

que es la ubicación relativa de la clase Consola.

Otro detalle con la consola es que, en realidad, no queremos una consola cada vez que le pidamos al reloj que se muestre, sino que vamos a querer que se muestre siempre en la misma consola. Por lo tanto, la declaración no debe ser local al método muestra, sino que tiene que ser para toda la clase. De esa manera se inicializa cuando se construye y desaparece cuando la clase ya no está presente.

Tenemos una tercera opción para la consola, y es declararla en la clase que usa el Reloj, ya que es el usuario el que debe proporcionar la consola donde se vea el Reloj. Si éste es el caso, entonces la declaración estaría en la clase UsoReloj y tendría que pasar como parámetro al método muestra. Podemos ver la versión final del encabezado y las declaraciones de este método en el listado 3.17.

Código 3.17 Versión final del encabezado y declaraciones de **muestra()**

```
/**
 * Muestra el estado de este reloj.
 * @param consola La consola en que muestra este reloj.
 */
  public void muestra (Consola consola) {
      String mensaje1, mensaje2, mensaje3;
      /* Implementación */
   } // muestra
```

Vamos a utilizar, por el momento, únicamente dos métodos de la clase Consola, uno de los constructores y el que escribe en la Consola. Sus firmas se encuentran a continuación:

```
Consola()           // Abre una consola de 400 por 600 pixeles
imprimeln(String)   // Escribe la cadena en la consola y da un
                    // salto de línea.
```

Dado lo anterior, agregamos a la clase UsoReloj la declaración de una Consola para poder mostrar ahí nuestro Reloj. Con lo que llevamos hecho no tenemos todavía una Consola, ya que no hemos construido un ejemplar de la misma.

El enunciado return

Cuando un método está marcado para *regresar un valor*, en cuyo caso el tipo del método es distinto de void, el método debe tener entre sus enunciados a return ⟨*expresión*⟩. En el punto donde este enunciado aparezca, el método suspende su funcionamiento y regresa el valor de la ⟨*expresión*⟩ al punto donde apareció su invocación. Cuando un método tiene tipo void, vamos a utilizar el enunciado return para salir del método justo en el punto donde aparezca este enunciado. Por ejemplo, los métodos de acceso lo único que hacen es regresar el valor del atributo,

por lo que quedan como se muestra en el listado 3.18.

Código 3.18 Implementación de los métodos de acceso de la clase **Manecilla**

```
class Manecilla
        implements ServiciosManecilla  {
    ...
    public int getValor () {
        return valor;
    } // getValor
    public int getLimite () {
        return LIM;
    } // getLimite
    ...
} // Manecilla
```

El enunciado de asignación

Figura 3.17 El enunciado de asignación

SINTAXIS:

⟨enunciado de asignación⟩::= | ⟨*variable*⟩ **=** ⟨*expresión*⟩

⟨*expresión*⟩ ::= ⟨*variable*⟩ | ⟨*constante*⟩
| **new** ⟨*constructor*⟩ | **(** ⟨*expresión*⟩ **)**
| ⟨*operador unario*⟩ ⟨*expresión*⟩
| ⟨*expresión*⟩ ⟨*operador binario* ⟩ ⟨*expresión*⟩
| ⟨*método que regresa valor* ⟩
| ⟨*enunciado de asignación*⟩

SEMÁNTICA:

Podemos hablar de que el ⟨*enunciado de asignación*⟩ consiste de dos partes, lo que se encuentra a la izquierda de la asignación (**=**) y lo que se encuentra a la derecha. A la izquierda tiene que haber una variable, pues es donde vamos a "guardar", copiar, colocar un valor. Este valor puede ser, como en el caso del operador **new**, una referencia a un objeto en el heap, o un valor. El; valor puede ser de alguno de los tipos primitivos o de alguna de las clases accesibles. La expresión de la derecha se evalúa (se ejecuta) y el valor que se obtiene se coloca en la variable de la izquierda. Si la expresión no es del mismo tipo que la variable, se presenta un error de sintaxis. Toda expresión tiene que regresar un valor.

Tal vez el ⟨*enunciado simple*⟩ más importante es el ⟨*enunciado de asignación*⟩, ya que va a ser el que me va a permitir asignarle un estado inicial a un objeto y la posibilidad de cambiarlo. También es el que me permite construir objetos y asociar una variable a cada objeto que construyo. Es conveniente recordar que las clases son únicamente plantillas para la construcción de objetos. Para que, en efecto, se realice algo se requiere construir objetos y asociarlos a variables para que podamos pedirles que hagan algo. El ⟨*enunciado de asignación*⟩ se muestra en la figura 3.17 en la página anterior.

La sintaxis de la expresión para la construcción de objetos se encuentra en la figura 3.18.

Figura 3.18 Construcción de objetos

SINTAXIS:
⟨*construcción de objeto*⟩ ::=**new** ⟨invocación método constructor⟩
SEMÁNTICA:
Para construir un objeto se utiliza el operador new y se escribe a continuación de él (dejando al menos un espacio) el nombre de alguno de los constructores que hayamos declarado para la clase, junto con sus argumentos. El objeto queda construido en el heap y tiene todos los elementos que vienen descritos en la clase.

La invocación de un método constructor o, para el caso de cualquier método del objeto mismo, se puede ver en la figura 3.19.

Figura 3.19 Invocación de método

SINTAXIS:
⟨*invocación de método*⟩ ::=⟨nombre del método⟩(⟨*Argumentos*⟩)
⟨*Argumentos*⟩ ::=⟨*argumento*⟩ (,⟨*argumento*⟩)* | ∅
⟨*argumento*⟩ ::=¡expresión¿
SEMÁNTICA:
Los ⟨*Argumentos*⟩ tienen que coincidir en número, tipo y orden con los ⟨*Parámetros*⟩ que aparecen en la declaración del método. La sintaxis indica que si la declaración no tiene parámetros, la invocación no debe tener argumentos.
Si el método regresa algún valor, entonces la invocación podrá aparecer en una expresión. Si su tipo es void tendrá que aparecer como enunciado simple.

El operador **new** nos regresa una dirección en el heap donde quedó construido el objeto (donde se encuentran las variables del objeto). Tengo que guardar esa referencia en alguna variable del tipo del objeto para que lo pueda usar. Si nos lanzamos a programar los constructores de la clase Reloj, lo hacemos instanciando a las manecillas correspondientes. La implementación de estos constructores se pueden ver en el listado 3.19.

Código 3.19 Constructores de la clase **Reloj**

```
1:      /** Inicia en ceros */
2:    Reloj (int limH, int limM) {
3:        horas = new Manecilla (limH);
4:            /* Usamos el primer parámetro como argumento */
5:        minutos = new Manecilla (limM);
6:            /* Usamos el segundo parámetro como argumento */
7:    } // Reloj (int, int)
8:
9:      /** Inicia en hora preestablecida  */
10:   Reloj (int limH, int limM, int hrs, int mins) {
11:       horas = new Manecilla (limH, hrs);
12:       minutos = new Manecilla (limM, mins);
13:   } // Reloj(int, int, int, int)
```

Para la clase Manecilla, que está compuesta únicamente de valores primitivos, éstos no se tienen que instanciar, por lo que la asignación basta. La implementación se encuentra en el listado 3.20.

Código 3.20 Constructores de la clase **Manecillas**

```
1:      /** Inicia en ceros  */
2:    Manecilla (int lim) {
3:        LIM = lim;
4:    } // Manecilla (int)\
5:
6:      /** Inicia en valor preestablecido  */
7:    Manecilla (int lim, int val) {
8:        LIM = lim;
9:        valor = val;
10:   } // Manecilla (int, int)
```

Podemos seguir con la implementación del resto de los métodos de la clase Manecilla, que son los más sencillos. El nombre de los métodos indica qué es lo que se tiene que hacer, por lo que obviaremos los comentarios, excepto cuando

valga la pena aclarar algo. Para la implementación de estos métodos utilizaremos ampliamente expresiones aritméticas, para poder colocarlas del lado derecho de una asignación. Por ello, conviene primero revisar cómo son las expresiones aritméticas en Java.

3.3 Expresiones en Java

Una expresión en Java es cualquier enunciado que nos regresa un valor. Por ejemplo, `new Manecilla(limH)` es una expresión, puesto que nos regresa un objeto de la clase `Reloj`. Podemos clasificar a las expresiones de acuerdo al tipo del valor que regresen. Si regresan un valor numérico entonces tenemos una expresión aritmética; si regresan falso o verdadero tenemos una expresión booleana; si regresan una cadena de carácteres tenemos una expresión tipo `String`. También podemos hacer que las expresiones se evalúen a un objeto de determinada clase.

Cuando escribimos una expresión aritmética tenemos, en general, dos dimensiones en las cuales movernos: una vertical y otra horizontal. Por ejemplo, en la fórmula que da la solución de la ecuación de segundo grado

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

estamos utilizando tres niveles verticales para indicar quién es el dividendo y quién el divisor. También, para indicar potencia simplemente elevamos un poco el número 2, e indicamos que la raíz se refiere a $b^2 - 4ac$ extendiendo la "casita" a que cubra a la expresión.

Cuando escribimos una expresión para un programa de computadora no contamos con estos niveles, sino que tenemos que "linealizar" la expresión: hacer que todo se encuentre en la misma línea vertical, pero manteniendo la *asociatividad* y la *precedencia*. La asociatividad nos habla de a quién afecta un operador dado, mientras que la precedencia se refiere al orden en que se tienen que evaluar las subexpresiones. Cada operador tiene una precedencia y asociatividad, pero se pueden alterar éstas usando paréntesis. Los paréntesis cumplen dos propósitos:

- Agrupan subexpresiones, de tal manera que se asocien a un operador. Por ejemplo, para indicar que el operando de la raíz es $b^2 - 4ac$ encerraríamos esta subexpresión entre paréntesis.

- Cambian el orden en que se evalúan las subexpresiones, ya que en presencia de paréntesis las expresiones se evalúan de adentro hacia afuera. Por ejemplo:

$\dfrac{x}{x+1}$	$x/(x+1)$
$\dfrac{x}{x}+1$	$x/x+1$

  Como se puede deducir del ejemplo anterior, la división tiene mayor precedencia (se hace antes) que la suma, por lo que en ausencia de paréntesis se evalúa como en el segundo ejemplo. Con los paréntesis estamos obligando a que primero se evalúe la suma, para que pase a formar el segundo operando de la división, como se muestra en el primer ejemplo.

Otra diferencia fuerte entre cuando escribimos fórmulas o expresiones en papel y cuando las escribimos en un programa es que la multiplicación *siempre* debe ser explícita en el programa:

$4ac$	$4 * a * c$
$3(x+2y)$	$3 * (x + 2 * y)$

Finalmente, son pocos los lenguajes de programación que tienen como operador la exponenciación, por lo que expresiones como b^2 se tendrán que expresar en términos de la multiplicación de b por sí misma, o bien usar algún método (como el que usamos para raíz cuadrada) que proporcione el lenguaje o alguna de sus bibliotecas. La "famosa" fórmula para la solución de una ecuación de segundo grado quedaría entonces

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad x1 = (-b + Math.sqrt((b * b) - (4 * a * c)))/(2 * a)$$

Con esta organización de paréntesis, lo primero que se hace es calcular $b * b$ y $4 * a * c$. Una vez que se tiene el resultado, se resta el segundo del primero. Una vez que se tiene el resultado, se le saca raíz cuadrada (se invoca a un método que sabe sacarla). Después se resta este resultado de b, se obtiene el producto de $2 * a$ y lo último que se hace es la división. Si no usáramos paréntesis, la expresión se interpretaría así:

$$-b + \frac{\sqrt{b^2-4ac}}{2}a \qquad -b + Math.sqrt(b * b - 4 * a * c)/2 * a$$

Otro aspecto importante de los operadores es el número de operandos sobre el que trabajan. Estamos acostumbrados a operadores unarios (de un solo operando, como el $-$ o el $\sim$) y binarios (como la suma o la multiplicación). En general, podemos tener operadores que tengan más de dos operandos.

A continuación damos una lista de operadores (no incluye métodos de la clase Math), listados en orden de precedencia, y con su asociatividad y número de operandos indicado. En general los operadores se evalúan de izquierda a derecha, para operadores de la misma precedencia o iguales (cuando la sintaxis lo permite), excepto los operadores de asignación que se evalúan de derecha a izquierda. En el caso de estos operadores únicamente la última expresión a la derecha puede ser algo que no sea una variable.

Cuadro 3.4 Operadores de Java

Operandos	Símbolo	Descripción	Prec
posfijo unario	[ ]	arreglos	1
posfijo unario	.	selector de clase	
prefijo n-ario	(⟨*parámetros*⟩)	lista de parámetros	
posfijo unario	⟨*variable*⟩++	auto post-incremento	
posfijo unario	⟨*variable*⟩−−	auto post-decremento	
unario prefijo	++⟨*variable*⟩	auto pre-incremento	2
unario prefijo	−−⟨*variable*⟩	auto pre-decremento	
unario prefijo	+⟨*expresión*⟩	signo positivo	
unario prefijo	−⟨*expresión*⟩	signo negativo	
unario prefijo	∼⟨*expresión*⟩	complemento en bits	2
unario prefijo	! ⟨expresión⟩	negación booleana	
unario prefijo	**new** ⟨*constructor*⟩	instanciador	3
unario prefijo	(⟨*tipo*⟩) ⟨*expresión*⟩	*casting*	
binario infijo	*	multiplicación	4
binario infijo	/	división	
binario infijo	%	módulo	
binario infijo	+	suma	5
binario infijo	−	rest	
binario infijo	<<	corrimiento de bits a la izquierda llenando con ceros	6
binario infijo	>>	corrimiento de bits a la derecha propangado el sign	
binario infijo	>>>	corrimiento de bits a la derecha llenando con cero	

Cuadro 3.4 Operadores de Java. . . . . (continúa)

Operandos	Símbolo	Descripción	Prec
binario infijo	<	relacional "menor que"	7
binario infijo	<=	relacional "menor o igual que'	
binario infijo	>	relacional "mayor que'	
binario infijo	>=	relacional "mayor o igual que'	
binario infijo	**instanceof**	relacional "ejemplar de'	
binario infijo	==	relacional, igual a	8
binario infijo	! =	relacional, distinto d	
binario infijo	&	AND de bits	9
binario infijo	∧	XOR de bits	10
binario infijo	|	OR de bits	11
binario infijo	&&	AND lógico	12
binario infijo	||	OR lógico	13
ternario infijo	⟨*exp log*⟩**?**⟨*exp*⟩ **:**⟨*exp*⟩	Condicional aritmética	14
binario infijo	=	asignación	15
binario infijo	+ =	autosuma y asignación	
binario infijo	− =	autoresta y asignación	15
binario infijo	* =	autoproducto y asignación	
binario infijo	/ =	autodivisión y asignación	
binario infijo	% =	automódulo y asignación	
binario infijo	>>=	autocorrimiento derecho con propagación y asignación	
binario infijo	<<=	autocorrimiento izquierdo y asignación	
binario infijo	>>>=	autocorrimiento derecho llenando con ceros y asignación	
binario infijo	& =	auto-AND de bits y asignación	
binario infijo	∧ =	auto-XOR de bits y asignación	
binario infijo	| =	auto-OR de bits y asignación	

Sabemos que ésta es una lista extensísima de operadores. Conforme vayamos entrando a cada uno de los temas , y requiramos de los operadores, aclararemos más su uso y su significado.

Estamos ya en condiciones de escribir prácticamente las implementaciones de todos los métodos en nuestro programa. Lo haremos, siguiendo el mismo orden que utilizamos para escribir los encabezados.

Implementación de los constructores

Lo único que deseamos hacer en los constructores de las clases Manecilla y Reloj es la de asignar valores iniciales a los atributos. Estos valores vienen como argumentos de los constructores. La programación se puede ver en los listados 3.22 y 3.21. Omitimos los comentarios de JavaDoc.

Código 3.21 Implementación de constructores de la clase **Manecilla**

```
/* Constructor que inicializa el límite */
public Manecilla(int lim) {
    LIM = lim;          /* Toma el valor del argumento */
} // Manecilla (int)
/* Constructor que inicializa límite y valor*/
public Manecilla(int lim, int val) {
        /* Toma valor de primer argumento */
    LIM = lim;
        /* Toma valor de segundo argumento */
    valor = val;
} // Manecilla (int, int)
```

Código 3.22 Implementación de constructores de la clase **Reloj**

```
/* Establece los límites de las manecillas */
public Reloj (int limH, int limM) {
      /* Construye el objeto horas, invocando a uno de sus
         constructores */
   horas   = new Manecilla(limH);
      /* Ahora construye al objeto minutos */
   minutos = new Manecilla(limM);
} // Reloj (int, int)
/* Establece los límites y el valor inicial de
   las manecillas */
public Reloj (int limH, int limM, int hrs, int mins) {
        /* Construye el objeto horas, estableciendo el límite
         * y el estado inicial */
    horas   = new Manecilla (limH,hrs);
        /* Ahora construye al objeto con estado inicial */
    minutos = new Manecilla(limM,mins);
} // Reloj (int, int, int, int)
```

Hay que recordar que estos métodos, por ser constructores, de hecho regresan

un objeto de la clase de la que son constructores.

Implementación de los métodos de acceso

Los métodos de acceso, como ya mencionamos, regresan un valor del tipo del atributo que deseamos observar. En el listado 3.23 se encuentran las implementaciones de los métodos de acceso de la clase Manecilla – a la clase Reloj no le declaramos ningún método de acceso.

Código 3.23 Métodos de acceso de la clase **Manecilla**

```
/**
 * Informa límite de manecilla
 * @return contenido de LIM
 */
public int getLimite () {
    return LIM;
} // Manecilla.getLimite()
/**
 * Informa valor de manecilla
 * Regresa contenido de valor
 */
public int getValor () {
    return valor;
} // Manecilla.getValor()
```

Implementación de los métodos de manipulación

En estos métodos tenemos todos aquellos que que cambian el estado de los objetos. Los que corresponden a la clase Reloj se encuentran en el listado 3.24 y las de la clase Manecilla en el listado 3.25 en la siguiente página.

Código 3.24 Métodos de manipulación de la clase **Reloj** 1/2

```
/**
 *  Cambia el estado de las dos manecillas
 *  @params hrs  el nuevo valor para horas
 *  @params mins el nuevo valor para minutos
public void setValor(int hrs, int mins) {
    horas.setValor(hrs);       /* pide a horas que cambie */
    minutos.setValor(mins);    /* pide a minutos que cambie */
} // Reloj.setValor (int, int)
```

Código 3.24 Métodos de manipulación de la clase **Reloj** 2/2

```
/**
 *  Incrementa 1 minuto y ve si se requiere
 *  incrementa horas
 */
public void incrementa()  {
    horas.setValor(horas.getValor + minutos.incrementa());
    /* Manecillas.incrementa dará 1 si dio la vuelta
       y 0 si no */
} // Reloj.incrementa ()
```

Código 3.25 Métodos de manipulación de la clase **Manecilla**

```
/** Copia el valor cuidando que esté en rangos */
public void setValor(int val)  {
    valor = val % LIM;                  /* Calcula módulo LIM */
} // Manecilla.setValor (int)
/** Incrementa en 1 el valor, cuidando de que quede en
 *  rangos. Regresa 1 si llegó al LIM y 0 si no */
public int incrementa()     {
    valor++;                    /* incrementa en 1 */
    valor %= LIM;               /* Verifica si alcanzó LIM */
    return (valor == 0) ? 1 : 0;
                                /* Si valor es cero, alcanzó a
                                 * LIM, por lo que debe regresar 1;
                                 * si no es así, regresa 0 */
} // Manecilla.incrementa ()
```

Implementación de métodos de implementación

El único método que tenemos de implementación es, como ya dijimos, el que muestra el reloj, que se encuentra en el listado 3.26.

Código 3.26 Métodos de implementación de la clase **Reloj**

```
public void muestra (Consola consola)  {
    /* El usuario dice dónde mostrar */
    String mensaje1, mensaje2, mensaje3;
    mensaje1 = "Son␣las␣";
    mensaje2 = "␣horas␣con␣";
    mensaje1 = "␣␣minutos.";
    consola.imprime (mensaje1+horas.getValor() +
                     mensaje2+minutos.getValor()+mensaje3);
        /* El operador + con cadenas fuerza a los enteros
           getValor() a convertirse a cadenas */
} // Reloj.muestra(Consola)
```

Tenemos ya las clases terminadas. Ahora tendríamos que tener un usuario que "comprara" uno de nuestros relojes. Hagamos una clase cuya única función sea probar el Reloj. La llamaremos UsoReloj. Se encuentra en el listado 3.27.

Código 3.27 Clase usuaria de la clase **Reloj**

```
import icc1.interfaz.Consola;

public class UsoReloj   {
    public static void main(String[] args)  {
            /* Declaraciones:
            /* declaración de una variable tipo Reloj */
        Reloj relojito;
            /* Dónde mostrar el reloj */
        Consola consolita;
            /* Construcción de los objetos:
               Valores iniciales */
        relojito = new Reloj(12, 60, 11, 58);
            /* El constructor sin parámetros */
        consolita = new Consola();

            /* Manipulación del relojito */
        relojito.incrementa();
        relojito.muestra(consolita);
        relojito.incrementa();
        relojito.muestra(consolita);
        relojito.setValor(10,59);
        relojito.muestra(consolita);
        relojito.incrementa();
        relojito.muestra();
    } // main
} // UsoReloj
```

Se estarán preguntando por qué no se declaró a relojito y consolita como atributos de la clase. La razón es que un método estático de la clase no puede tener acceso a atributos de esa clase. Por ello hay que declararlo en el método. De cualquier forma, como la clase Reloj es pública, cualquiera puede pedir constructores de esa clase.

3.3.1. Declaración y definición simultáneas

No hemos mencionado que en Java se permite asignar valor inicial a los atributos y a las variables locales en el momento en que se declaran. Esto se consigue

simplemente con el operador de asignación y una expresión:

```
public int valor = 0;
Reloj relojito = new Reloj (12, 60);
```

Para el caso de los atributos de un objeto, no se le ve mucho caso asignar estado inicial a los atributos, excepto cuando queramos que todos los objetos de esa clase compartan el estado inicial. Por ejemplo, en el caso de los objetos de la clase Reloj es posible que queramos que todos los objetos empiecen con las 0 horas y 0 minutos; pero en el caso de los objetos de la clase Manecilla, si le diéramos valor inicial al atributo Lim después ya no podríamos volverle a asignar un valor, y todos los objetos tendrían el mismo límite, algo que no queremos que suceda.

Manejo de cadenas y expresiones

4

Uno de los ingredientes que más comúnmente vamos a usar en nuestros programas son las expresiones. Por ello, dedicaremos este capítulo a ellas.

4.1 Manejo de cadenas en Java

Una expresión es cualquier sucesión de operadores y operandos que producen (regresan) un valor al evaluarse. El valor puede ser numérico, de cadenas, una referencia a un objeto, booleano, o de cualquier otra clase accesible al método en la que se encuentra la expresión.

Las cadenas de caracteres van a ser de lo que más vamos a usar en nuestro desarrollo profesional. Prácticamente todo el manejo que hagamos involucrará cadenas, ya sea como títulos, o como objeto de búsquedas, de agrupamiento, etc.

Las cadenas en Java son una clase que nos proporciona el paquete Java.Lang y está accesible sin necesidad de importarlo. Cada vez que declaramos una cadena, mediante el identificador de tipo String, reservamos espacio únicamente para la referencia, ya que se trata de objetos. Sin embargo, por lo común que son las

cadenas la sintaxis de Java es mucho más flexible para la creación de cadenas que de objetos en general y nos permite cualquiera de los siguientes formatos:

I. **En la declaración.** Simplemente inicializamos la variable con una cadena:

```
String cadena = "Esta es una cadenita";
```

II. **En una asignación.** Se asigna una cadena a una variable tipo String:

```
String cadenota;
cadenota = "Una cadena"+ " muy larga";
```

III. **Al vuelo.** Se construye una cadena como una expresión, ya sea directamente o mediante funciones de cadenas:

```
"Cadena Muy Larga".toLowerCase()
```

Es importante mencionar que las cadenas, una vez creadas, no pueden ser modificadas. Si se desea modificar una cadena lo que se debe hacer es construir una nueva con las modificaciones, y, en todo caso, reasignar la nueva. Por ejemplo, si queremos pasar a mayúsculas una cadena, podríamos tener la siguiente sucesión de enunciados:

```
String minusc = "está en minúsculas";
minusc = minusc.toUpperCase();
```

Nótese que en el primer renglón de este código la cadena contiene únicamente minúsculas. En el lado derecho de la asignación en el segundo renglón se construye una cadena nueva que es la cadena minusc pero pasada a mayúsculas; lo último que se hace es reasignar la referencia de minusc a que ahora apunte a esta nueva cadena.

Lo distinto cuando a manejo de cadenas se refiere es que no necesitamos el operador new – aunque lo podemos usar con alguno de los métodos constructores – para construir un objeto tipo String.

La clase String proporciona muchísimos métodos para trabajar con cadenas. No mostramos todos, pues algunos de ellos tienen parámetros o entregan valores que no hemos visto todavía. A continuación se encuentra una tabla con los métodos que podemos querer usar de la clase String.

Cuadro 4.1 Métodos de la clase String

Firma	Descripción
Constructores:	
String() String(String)	Construye una nueva cadena nula en el primer caso, y otra que es copia de la primera en el segundo. En ambos casos regresa un apuntador al heap.
Métodos para crear nuevas cadenas:	
String concat(String)	Crea una nueva cadena que es a la que se le solicita el método, seguida del argumento.
String replace(**char**, **char**)	Crea una nueva cadena en la que reemplaza las apariciones del primer carácter por el segundo.
String replace(String, String)	Crea una nueva cadena en la que reemplaza las apariciones de la primera cadena por la segunda.
String substring(**int**) String substring(**int**, **int**)	Crean una nueva cadena que es una subcadena de la cadena. La subcadena empieza en el primer entero y termina, en el primer caso al final de la cadena y en el segundo en el segundo entero.
String toLowerCase()	Crea una nueva cadena convirtiendo todos los carácteres a minúsculas.
String toUpperCase()	Crea una nueva cadena convirtiendo todos los carácteres a mayúsculas.
String trim()	Crea una nueva cadena quitando los blancos del principio y final
Métodos para crear nuevas cadenas:	
static String valueOf(**boolean**) **static** String valueOf(**char**) **static** String valueOf(**int**)	Crea una cadena con el valor que corresponde al tipo del dato. Como es estática se puede llamar desde la clase: String.valueOf(valor).

Cuadro 4.1 Métodos de la clase String (continúa)

Firma	Descripción
Métodos para crear nuevas cadenas: (continúa)	
static String valueOf(**long**) **static** String valueOf(**float**) **static** String valueOf(**double**)	Crea una cadena con el valor que corresponde al tipo del dato. Como es estática se puede llamar desde la clase: String.valueOf(valor).
Métodos de comparación:	
int compareTo(String)	Compara dos cadenas en el orden del código Unicode. $\begin{cases} \mathbf{0} & \text{si las cadenas son idénticas} \\ > \mathbf{0} & \text{si } \langle cad1 \rangle \text{ va después en el orden que } \langle cad2 \rangle. \\ < \mathbf{0} & \text{si } \langle cad1 \rangle \text{ va antes en el orden que } \langle cad2 \rangle. \end{cases}$
boolean equals(Object)	Dice si la cadena en el parámetro es idéntica a la que invoca.
boolean equalsIgnoreCase(String)	Dice si las cadenas son iguales, ignorando diferencias entre mayúsculas y minúsculas.
Métodos de búsqueda:	
boolean endsWith(String)	Dice si la cadena con la que se invoca termina con la cadena en el parámetro.
int indexOf(**int**) **int** indexOf(**int**, **int**) **int** indexOf(String) **int** indexOf(String, **int**)	El primer entero corresponde al código de un carácter en Unicode (se puede pasar como argumentos también un carácter). La cadena se refiere a una subcadena. En las 4 versiones, regresa la primera posición en la cadena donde se encuentra el primer parámetro. Si se da un segundo parámetro, éste indica que se busque a partir de esa posición. Regresa -1 si no encuentra lo que está buscando.

Cuadro 4.1 Métodos de la clase String (continúa)

Firma		Descripción
boolean	startsWith(String)	Determina si es que la cadena empieza con la cadena que trae como argumento. En la segunda versión, ve a partir del carácter denotado por el argumento entero.
boolean	startsWith(String, **int**)	
int	lastIndexOf(**char**)	El carácter corresponde a un carácter (se puede pasar como argumentos también un código entero de un carácter en Unicode). La cadena se refiere a una subcadena. En las 4 versiones, regresa la última posición en la cadena donde se encuentra el primer parámetro. Si se da un segundo parámetro, éste indica que se busque a partir de esa posición. Regresa -1 si no encuentra lo que está buscando.
int	lastIndexOf(**char**, **int**)	
int	lastIndexOf(String)	
int	lastIndexOf(String, **int**)	
boolean	regionMatches	Determina si una región de la cadena es igual a una región de la cadena en el argumento. La segunda versión, si el argumento booleano es verdadero, compara ignorando diferencias entre mayúsculas y minúsculas. El primer entero es la posición de la región en la cadena que invoca. el segundo entero es loa posición inicial en la cadena del argumento. La tercera posición es el número de carácteres a comparar.
boolean	(**int**, String, **int**, **int**)	
	regionMatches	
	(**boolean**,**int**, String, **int**, **int**)	
Métodos de conversión		
char	charAt(**int**)	Regresa el carácter que se encuentra en la posición dada por el argumento.
String	toString()	Genera la representación en cadena del objeto con el que se le invoca.
Otros Métodos		
int	length()	Regresa el tamaño de la cadena, el número de carácteres.

4.2 Implementación de una base de datos

Supongamos que tenemos un conjunto de cadenas almacenadas de alguna forma (nos preocuparemos de la implementación después). Por ejemplo, tengo los nombres de los estudiantes del grupo con su carrera y quiero poder extraer datos de allí. Tenemos, entonces, lo que se conoce como una *base de datos*. Identifiquemos las operaciones que deseamos poder hacer con esa base de datos:

Problema: Mantener una base de datos con listas de cursos.

Descripción:

Cada objeto del curso consiste del número del grupo (una cadena), la lista de alumnos y el número de alumnos. La lista de alumnos consiste de alumnos, donde para cada alumno tenemos su nombre completo, su número de cuenta, la carrera en la que están inscritos y su clave de acceso a la red.

Las operaciones que queremos se puedan realizar son:

(a) Localizar a un estudiante, proporcionando cualquiera de sus datos, que lo distingan de los otros estudiantes.

(b) Dado un dato particular de un estudiante, recuperar su nombre, clave, cuenta o carrera.

(c) Agregar estudiantes.

(d) Quitar estudiantes.

(e) Poder emitir la lista de todo el grupo.

(f) Emitir la sublista de los que contienen cierto valor en alguno de sus campos.

Entonces, nuestra tarjeta de responsabilidades, en cuanto a la parte pública se refiere, se puede ver en la figura 4.1.

Figura 4.1 Tarjeta de responsabilidades para Curso.

Clase: Curso		Responsabilidades
P	Constructores	A partir de una base de datos inicial y a partir de cero.
ú	daNombre	Regresa el nombre completo de un estudiante
	daCarrera	Regresa la carrera de un estudiante
b	daClave	Regresa la clave de acceso de un estudiante
	daCuenta	Regresa el número de cuenta de un estudiante
l	agregaEstudiante	Agrega a un estudiante, proporcionando los datos correspondientes.
i	quitaEstudiante	Elimina al estudiante identificado para eliminar.
	listaCurso	Lista todos los estudiantes del curso
c	losQueCazanCon	Lista a los estudiantes que cazan con algún criterio específico
o	armaRegistro	Regresa el registro "bonito" para imprimir

De esto, podemos definir ya una interfaz de Java que se encargue de definir estos servicios. La podemos ver en el Listado 4.1.

Código 4.1 Interfaz para el manejo de una base de datos 1/3

```
1: /**
2:  * Da el manejo de una base de datos sobre la que se
3:  * desean hacer consultas.
4:  */
5:
6: public interface Consultas  {
7:   /**
8:     * Produce la lista de todos los registros en la
9:     * base de datos.
10:    * @return Una cadena con la lista editada.
11:    */
12:   public String listaCurso();
```

Código 4.1 Interfaz para el manejo de una base de datos 2/3

```
13:   /**
14:    * Dada la posición del registro, regresa el nombre
15:    * del alumno en esa posición.
16:    * @param   cual La posición del registro.
17:    * @return  Una cadena que contiene el nombre completo
18:    *          del registro en esa posición.
19:    */
20:   public String daNombre(int cual);
21:
22:   /**
23:    * Dada la posición del registro, regresa la carrera
24:    * del alumno en esa posición.
25:    * @param   cual La posición del registro.
26:    * @return  Una cadena que contiene el nombre de la
27:    *          carrera del registro en esa posición.
28:    */
29:   public String daCarrera(int cual);
30:
31:   /**
32:    * Dada la posición del registro, regresa la carrera
33:    * del alumno en esa posición.
34:    * @param   cual La posición del registro.
35:    * @return  Una cadena que contiene la carrera del
36:    *          registro en esa posición.
37:    */
38:   public String daCarrera(int cual);
39:
40:   /**
41:    * Dada la posición del registro, regresa el número
42:    * de cuenta del alumno en esa posición.
43:    * @param   cual La posición del registro.
44:    * @return  Una cadena que contiene el número de cuenta
45:    *          del registro en esa posición.
46:    */
47:   public String daCuenta(int cual);
48:
49:   /**
50:    * Dada la posición del registro, regresa la clave
51:    * de acceso del alumno en esa posición.
52:    * @param   cual La posición del registro.
53:    * @return  Una cadena que contiene la clave de
54:    *          acceso del registro en esa posición.
55:    */
56:   public String daClave(int cual);
```

Código 4.1 Interfaz para el manejo de una base de datos 3/3

```
57:   /**
58:    * Dada la posición del registro, regresa una cadena que
59:    * contiene el registro completo del alumno en esa posición.
60:    *
61:    * @param   cual La posición del registro.
62:    * @return       Una cadena que contiene el número de
63:    *               cuenta del registro en esa posición.
64:    */
65:   public String armaRegistro(int cual);
66:
67:   /**
68:    * Lista a los estudiantes cuyo registro contiene como
69:    * subcadena a la cadena en el parámetro.
70:    *
71:    * @param subcad La subcadena que buscamos.
72:    * @return       Una cadena con un registro por renglón.
73:    */
74:   public String losQueCazanCon(String subcad);
```

En la interfaz que acabamos de dar, casi todos los métodos que hacen la consulta trabajan a partir de saber la posición relativa del registro que queremos. Sin embargo, una forma común de interrogar a una base de datos es proporcionándole información parcial, como pudiera ser alguno de los apellidos, por lo que conviene agregar un método al que le proporcionamos esta información y nos deberá decir la posición relativa del registro que contiene esa información. Este método lo podemos ver en el Listado 4.2.

Código 4.2 Posición de un registro que contenga una subcadena **(Consultas)**

```
75:   /**
76:    * Dada una cadena con el nombre del alumno, regresa la
77:    * posición del registro que contiene ese nombre.
78:    *
79:    * @param   nombre El nombre del alumno que buscamos.
80:    *
81:    * @return  Un entero que corresponde a la posición
82:    *          relativa del registro que contiene al nombre.
83:    */
84:   public int daPosicion(String nombre);
```

Sin embargo, pudiéramos buscar una porción del registro que se repite más de una vez, y quisiéramos que al interrogar a la base de datos, ésta nos diera, uno tras

otro, todos los registros que tienen esa subcadena. Queremos que cada vez que le pidamos no vuelva a empezar desde el principio, porque entonces nunca pasaría del primero. Le agregamos entonces un nuevo parámetro para que la búsqueda sea a partir de un posición. El encabezado de este método se puede ver en el Listado 4.3.

Código 4.3 Posición de un registro a partir de otra posición (**Consultas**)

```
85:   /**
86:    *
87:    * Dados una cadena con el nombre del alumno y a partir
88:    * de cuál posición buscar, regresa la posición del registro
89:    * que contiene ese nombre, sin examinar a los que están antes
90:    * de la posición dada.
91:    *
92:    * @param   nombre El nombre del alumno que buscamos.
93:    *
94:    * @param   desde  A partir de donde se va a hacer
95:    *                 la búsqueda.
96:    *
97:    * @return  Un entero que corresponde a la posición
98:    *          relativa del registro que contiene al nombre,
99:    *          buscado a partir de desde.
100:   *
101:   */
102:  public int daPosicion(String nombre, int desde);
```

Definida ya la interfaz, procedemos a diseñar una implementación para la misma. Siguiendo la metodología que tenemos para definir las clases, una vez definidas las responsabilidades debemos decidir cuál es la información que requiere la clase para poder brindar los servicios anunciados en la interfaz. Lo primero que necesitamos es la información que corresponde a la base de datos y, de alguna manera, la descripción de qué contiene cada registro. La manera como he decidido guardar esta base de datos es en una cadena enorme, pero subdividida en pedazos del mismo tamaño. A cada uno de estos pedazos lo vamos a manejar como un *registro*. Además, cada registro lo vamos a dividir en *campos*, donde cada campo corresponde a una unidad de información; por ejemplo, el nombre del estudiante corresponde a un campo, así como la clave a otro, el número de cuenta a un tercero y así sucesivamente. Esto nos facilita ver a la base de datos como si fuera una tabla, donde cada renglón de la tabla corresponde a un registro y cada columna de la tabla a un campo (o atributo). Supongamos que tenemos una lista de alumnos como la que se muestra en la Tabla 4.2 en la página opuesta:

Cuadro 4.2 Listado del contenido de nuestra base de datos

Nombre:	Carrera:	Cuenta:	Clave:
Aguilar Solís Aries Olaf	Matemático	97541219-1	aguilarS
Cruz Cruz Gil Noé	Computación	99036358-4	cruzCruz
García Villafuerte Israel	Computación	02598658-3	garciaVi
Hubard Escalera Alfredo	Computación	00276238-7	hubardE
Tapia Vázquez Rogelio	Actuaría	02639366-8	tapiaV
...	...	...	...

Como ya mencionamos, vamos a representar a la base de datos con una cadena en la que colocaremos a todos los registros. Si no forzamos a que cada registro ocupe el mismo número de posiciones no podríamos decir de manera sencilla dónde termina un registro y empieza el siguiente. En cada uno de los registros, el número de posiciones que ocupa, por ejemplo, el nombre también debe ser el mismo, también para que podamos calcular, de manera sencilla, la posición en la cadena donde empieza cada campo dentro de cada registro. La declaración con valores iniciales para esta lista se vería, entonces, como sigue:

```
private String
  lista =
   "Aguilar␣Solís␣Aries␣Olaf␣" + "Mate" + "975412191" + "aguilarS" +
   "Cruz␣Cruz␣Gil␣Noé␣␣␣␣␣␣␣␣" + "Comp" + "990363584" + "cruzCruz" +
   "García␣Villafuerte␣Israel" + "Comp" + "025986583" + "garciaV␣" +
   "Hubard␣Escalera␣Alfredo␣␣" + "Comp" + "002762387" + "hubardE␣" +
   "Tapia␣Vázquez␣Rogelio␣␣␣␣" + "Actu" + "026393668" + "tapiaV␣␣";
```

Recuerden que podemos construir una cadena concatenando cadenas, usando el operador +. Además, aparecen los carácteres blancos dentro de las cadenas con ␣ , para que podamos ver fácilmente el número de posiciones que ocupa. Dividimos la cadena – arbitrariamente – en registros, uno por renglón, y cada registro en campos. Noten que cada nombre ocupa 25 posiciones y cada carrera 4; el número de cuenta ocupa 9 posiciones y la clave de acceso 8 posiciones. Cada elemento de mi lista ocupa, entonces, 46 posiciones. Nótese también que elegimos "codificar" la carrera, pues con cuatro carácteres tengo suficiente para reconocer la carrera; para el número de cuenta usamos 9 posiciones, pues la que estaría entre el octavo dígito y el noveno *siempre* contiene un guión; por último, para la

clave de usuario utilizamos 8 posiciones, completando cuando es necesario como lo hicimos en el nombre.

La lista empieza en **0**, y tiene, en total, $\mathbf{5 \times 46 = 230}$ posiciones (de la 0 a la 229). El primer registro empieza en la posición 0; el segundo registro empieza en la posición 46. En general, el $\boldsymbol{i}$-ésimo registro empieza en la posición $(\boldsymbol{i - 1}) * \mathbf{46}$. ¿Por qué $\boldsymbol{i-1}$? Porque hay que "saltar" $\boldsymbol{i-1}$ registros para llegar a donde empieza el $\boldsymbol{i}$-ésimo.

El primer nombre empieza donde el primer registro; el segundo nombre donde el segundo registro y así sucesivamente. La carrera que corresponde al primer registro empieza en la posición 25, una vez saltadas las primeras 25 posiciones (de la 0 a la 24) que corresponden al nombre. La del segundo registro empieza en la posición 71 (46+25), que corresponde a saltar las primeras **46** posiciones (de la 0 a la 45) que corresponden al primer registro, más las primeras **25** (de la 0 a la 24) posiciones que corresponden la nombre del segundo registro. En general, la posición de la carrera del $\boldsymbol{i}$-ésimo registro empieza en la posición $(\boldsymbol{i-1}) * \mathbf{46} + \mathbf{25}$, donde **46** es el número de posiciones que hay que saltar por cada elemento de la tabla que se encuentra *antes* que el que queremos, y **25** es el desplazamiento (*offset*) del campo que deseamos a partir del principio del elemento. En general, si se desea el $\boldsymbol{j}$-ésimo campo del $\boldsymbol{i}$-ésimo registro, se obtiene la posición inicial del $\boldsymbol{i}$-ésimo registro $((\boldsymbol{i - 1}) * \mathbf{46})$ y a eso se le suma el total de las posiciones que ocupan los campos desde el primero hasta el $\boldsymbol{j - 1}$.

Construyamos una clase para manejar listas de cursos. Ya tenemos, de la interfaz, los métodos públicos que vamos a requerir; ahora hay que decidir que atributos requiere la clase para poder dar esos servicios. Si estamos hablando de un grupo en la Facultad de Ciencias es conveniente que se guarde el número del grupo. También es conveniente que cada base de datos me pueda responder, de manera sencilla, el número de registros que tiene en ese momento. Estos tres atributos son privados y en todo caso se tiene acceso a ellos a través de métodos de acceso. La tarjeta de responsabilidades, incluyendo a estos atributos privados se encuentra en la figura 4.2.

Como parte de la información que requiere la clase es conveniente declarar los tamaños del registro y de los campos como constantes para poder dar expresiones aritméticas en términos de estas constantes. De esa manera si decidimos cambiar el tamaño de alguno de los campos únicamente tenemos que localizar la declaración de la constante para hacerlo. El inicio de la codificación la podemos ver en el listado 4.4 en la página opuesta.

Figura 4.2 Tarjeta de responsabilidades para Curso.

Clase: `Curso`		Responsabilidades
Público	Constructores	A partir de una base de datos inicial y a partir de cero.
	daNombre	Regresa el nombre completo de un estudiante
	daCarrera	Regresa la carrera de un estudiante
	daClave	Regresa la clave de acceso de un estudiante
	daCuenta	Regresa el número de cuenta de un estudiante
	agregaEstudiante	Agrega a un estudiante, proporcionando los datos necesarios.
	quitaEstudiante	Elimina a un estudiante después de identificarlo.
	listaCurso	Lista todos los estudiantes del curso
	losQueCazanCon	Lista a los estudiantes que cazan con algún criterio específico
	armaRegistro	Regresa el registro "bonito" para imprimir
Privado	lista	Base de datos con los alumnos inscritos
	grupo	Número que identifica al grupo
	número de registros	En cada momento, el número de registros que contiene el grupo

Código 4.4 Clase que maneja listas de cursos (**Curso**) 1/2

```
1: import icc1.interfaz.Consola
2: /**
3:  * Base de datos, a base de cadenas, que emula la lista de un curso
4:  * de licenciatura. Tiene las opciones normales de una base de
5:  * datos y funciona mediante un Menú
6:  */
7: class Curso implements Consultas     {
8:     private String lista;        /* Base de datos */
9:     private String grupo;        /* Clave del grupo */
10:    private int numRegs;         /* Número total de registros */
```

Código 4.4 Clase que maneja listas de cursos. (**Curso**)2/2

```
11:     /**
12:      * TAM_XXXX: Tamaño del campo
13:      * OFF_XXXX: Distancia del campo XXXX al principio del registro
14:      */
15:     private static final int TAM_REG    = 46;
16:     private static final int TAM_NMBRE = 25;
17:     private static final int TAM_CARRE = 4;
18:     private static final int TAM_CTA    = 9;
19:     private static final int TAM_CLVE   = 8;
20:     private static final int OFF_NMBRE = 0;
21:     private static final int OFF_CARRE = 25;
22:     private static final int OFF_CTA    = 29;
23:     private static final int OFF_CLVE   = 38;
```

En esta pequeña prueba estamos utilizando únicamente una función de cadenas (String), substring, que me entrega la subcadena que empieza en el primer parámetro y termina en el segundo parámetro, ambos enteros. Este método lo invocamos desde la lista de nombres y datos adicionales. Una tabla de métodos relevantes de la clase String en esta etapa se encuentran en la tabla 4.1 en la página 101. Tenemos un método que nos encuentra la primera posición del i-ésimo registro, daPosI(int i), y a partir de ahí nos saltamos los campos que van antes del que queremos. Así, el campo que corresponde al nombre está en la posición 0 de cada registro - OFF_NMBRE = 0 – mientras que para llegar al campo con la clave de usuario hay que saltar el tamaño del nombre, más el tamaño de la cuenta más el tamaño de la carrera – OFF_CLVE = 38 = 25 + 4 + 9. Similarmente localizamos el inicio de cada uno de los otros campos.

Veamos ahora la implementación de los constructores en el diagrama de la figura 4.3. En este diagrama vemos que lo que tenemos que hacer en cada uno de los constructores es darle valor inicial a los datos privados de la clase.

Figura 4.3 Diagrama de Warnier-Orr para los constructores.

$$\textbf{\textit{Constructor}} \begin{cases} \text{Construye lista inicial} \\ \text{Registra número de estudiantes} \\ \text{Registra clave del grupo} \end{cases}$$

Habíamos comentado que queremos dos constructores, uno que trabaje a partir de una lista que dé el usuario, y otro que inicie con una lista vacía y vaya agregando

nombres conforma el usuario los va dando. El código para ambos casos se puede ver en el listado 4.5.

Código 4.5 Constructores para la clase Curso (**Curso**)

```
24:     /**
25:      * Construye una base de datos a partir de los datos que de el
26:      * usuario.
27:      * @param grupo La clave del grupo
28:      * @param lista La lista bien armada del grupo
29:      * @param cuantos El número de registros que se registran
30:      */
31:     public Curso(String grupo, String lista)  {
32:        this.lista = lista == null
33:                     ?""
34:                     : lista;
35:        this.grupo = grupo == null
36:                     ?"????"
37:                     :grupo;
38:        numRegs = lista.length()==0
39:                  ?0
40:                  : lista.length() /TAM_REG + 1;
41:     }
42:     /**
43:      * Construye una base de datos vacía pero con número de grupo
44:      * @param grupo Número de grupo
45:      */
46:     public Curso(String grupo)  {
47:         this.lista = "";
48:         this.grupo = grupo == null
49:                      ? "????"
50:                      : grupo;
51:         numRegs = 0;
52:     }
```

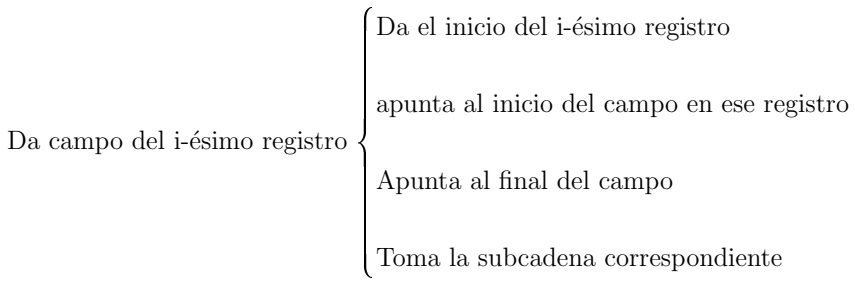
El método que da la lista completa es muy sencillo, ya que únicamente regresa la lista. Lo podemos ver en el listado 4.6.

Código 4.6 Método que regresa toda la lista (**Curso**)

```
53:     /**
54:      * Da acceso a la base de datos completa
55:      * @return toda la lista
56:      */
57:     public String daLista()  {
58:         return lista;
59:     }
```

Para los métodos que regresan un determinado campo de un registro tenemos el algoritmo que se muestra en la figura 4.4.

Figura 4.4 Diagrama de Warnier-Orr para regresar el contenido de un campo.



Como siempre nos vamos a estar moviendo al inicio del i-ésimo registro vamos a elaborar un método, privado, que me dé el carácter en el que empieza el i-ésimo registro, mediante la fórmula

$$\text{posición} = (\boldsymbol{i} - \mathbf{1}) * \boldsymbol{TAM\_REG}.$$

Hay que tomar en cuenta acá al usuario, que generalmente va a numerar los registros empezando desde el 1 (uno), no desde el 0 (cero). Con esto en mente y de acuerdo a lo que es la que discutimos al inicio de este tema, el método queda como se puede observar en el listado 4.7.

Código 4.7 Cálculo de la posición donde empieza el i-ésimo registro **(Curso)**

```
60:     /**
61:      * Da el número de carácter en el que empieza el i-ésimo
62:      * registro.
63:      * @param i el ordinal del registro
64:      * @return el carácter en el que empieza
65:      */
66:     private int daPosI(int i) {
67:         return (i-1) * TAM_REG;
68:     }
```

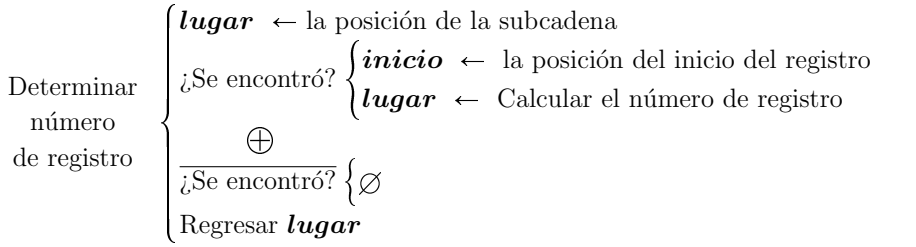
Los métodos que regresan un campo siguen todos el patrón dado en la figura 4.4 y su implementación se puede ver en el listado 4.8 en la página opuesta.

Código 4.8 Métodos que regresan el contenido de un campo (**Curso**)

```
69:     /**
70:      * Regresa el nombre completo del i-ésimo registro.
71:      * @param el ordinal del registro
72:      * @return la cadena que corresponde al nombre
73:      */
74:     public String daNombre(int i)  {
75:         int empza, termina;
76:         empza = daPosI(i) + OFF_NMBRE;
77:         termina = daPosI(i) + OFF_NMBRE + TAM_NMBRE;
78:         return lista.substring(empza, termina);
79:     }
80:     /**
81:      * Regresa la carrera del i-ésimo registro
82:      * @param i el ordinal del registro
83:      * @return la subcadena que corresponde a la carrera
84:      */
85:     public String daCarrera(int i)  {
86:         int empza, termina;
87:         empza = daPosI(i) + OFF_CARRE;
88:         termina = daPosI(i) + OFF_CARRE + TAM_CARRE;
89:         String clave = lista.substring(empza, termina);
90:         int cualClave = "actbiofismatcom"
91:     }
92:     /**
93:      * Regresa el número de cuenta del i-ésimo registro
94:      * @param i el ordinal del registro
95:      * @return la subcadena que corresponde al número de cuenta
96:      */
97:     public String daCuenta(int i)  {
98:         int empza, termina;
99:         empza = daPosI(i) + OFF_CTA;
100:        termina = daPosI(i) + OFF_CTA + TAM_CTA;
101:        return lista.substring(empza, termina);
102:    }
103:    /**
104:     * Regresa la clave de acceso del i-ésimo registro
105:     * @param i el ordinal del registro
106:     * @return la subcadena que corresponde a la clave de acceso
107:     */
108:    public String daClave(int i)  {
109:        int empza, termina;
110:        empza = daPosI(i)+OFF_CLVE;
111:        termina = daPosI(i)+OFF_CLVE + TAM_CLVE;
112:        return lista.substring(empza, termina);
113:    }
```

Los métodos que acabamos de programar asumen que se sabe el número de registro que se está buscando, por lo que tenemos que dar un método que dada una subcadena regrese el número del registro que se está buscando. Como existe la posibilidad de que no se encuentre la subcadena, se debe verificar esa posibilidad. El algoritmo para este método se muestra en la figura 4.5.

Figura 4.5 Encontrar el número de registro al que pertenece una subcadena.



Para encontrar la posición de la subcadena simplemente usamos métodos de la clase String. Para encontrar el inicio del registro, simplemente le quitamos lo que sobre del múltiplo más cercano del tamaño del registro. Y finalmente vemos cuántos registros caben en ese número.

Código 4.9 Método que da el primer registro con subcadena **(Curso)**

```
114:     /**
115:      * Da el ordinal que corresponde al primero registro que contiene
116:      * a la subcadena
117:      * @param nombre subcadena a buscar
118:      * @return el ordinal del registro, o -1 si no hay
119:      */
120:     public int daPosicion(String nombre)  {
121:         int lugar = lista.toLowerCase().indexOf(nombre.toLowerCase());
122:         int sobran = (lugar >= 0) ? (lugar % TAM_REG) : 0;
123:         return (lugar >= 0 ) ? ((lugar - sobran) / TAM_REG) +1
124:                                         : lugar;
125:     }
```

Supongamos ahora que no queremos al primero que contenga la subcadena, sino uno que esté después de cierta posición. El algoritmo es prácticamente el mismo, excepto que usamos otra firma de la función indexOf, la que toma en cuenta

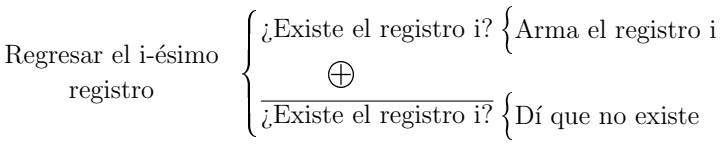
una posición inicial a partir de donde buscar. Le damos al método daPosicion otra firma que tome en cuenta este parámetro adicional. La programación se encuentra también en los listados 4.9 en la página opuesta y 4.10.

Código 4.10 Método que da el siguiente registro con subcadena (**Curso**)

```
126:     /**
127:      * Da el ordinal que corresponde al registro que contiene
128:      * a la subcadena, a partir de la posición dada
129:      * @param nombre subcadena a buscar
130:      * @param desde número de carácter a partir del cual buscar
131:      * @return el ordinal del registro, o -1 si no hay
132:      */
133:     public int daPosicion(String nombre, int desde)  {
134:         int nvoReg = (desde) * TAM_REG;
135:         int lugar = lista.toLowerCase().indexOf(nombre.toLowerCase(),
136:                                                 nvoReg);
137:         int sobran = lugar % TAM_REG;
138:         return (lugar >= 0 ) ? ((lugar - sobran ) / TAM_REG) +1
139:                              : lugar;
140:     }
```

El único método que nos falta de los que trabajan con un registro particular es el que arma un registro para mostrarlo. El algoritmo es sumamente sencillo, y lo mostramos en la figura 4.6. Lo único relevante es preguntar si el registro que nos piden existe o no.

Figura 4.6 Edición del i-ésimo registro, si es que existe.



La programación correspondiente se encuentra en el listado 4.11 en la siguiente página. El carácter \t que aparece entre cada dos elementos del listado es un tabulador, que lo que hace es alinear bien los campos para listarlos bonito.

Código 4.11 Edición de un registro individual (**Curso**)

```
141:     /**
142:      * Arma el registro para mostrar que se encuentra en la i-ésima
143:      * posición.
144:      * @param i la posición del registro
145:      * @return el registro armado o un mensaje de que no existe
146:      */
147:     public String armaRegistro(int i) {
148:         return (i > 0) ?
149:             daNombre(i) + "\t" + daCarrera(i) + "\t" +
150:             daCuenta(i) + "\t" + daClave(i)
151:             : "No se encontró al nombre buscado";
152:     }
```

De las funciones más comunes a hacer con una lista de un curso es listar toda la lista completa. Sabemos cuántos registros tenemos, todo lo que tenemos que hacer es *recorrer* la lista e ir mostrando uno por uno. A esto le llamamos *iterar* sobre la lista. El algoritmo podría ser el que se ve en la figura 4.7.

Figura 4.7 Algoritmos para listar el curso.

$$\text{lista curso} \begin{cases} \text{muestra i-ésimo registro} \\ (i = 1, \ldots, \textbf{numRegs}) \end{cases} \begin{cases} \text{arma el registro i} \\ \text{avanza} \end{cases}$$

En Java tenemos varios enunciados compuestos que iteran. El más general de ellos es de la forma

$$\begin{array}{c} \text{Ejecuta enunciado compuesto} \\ \text{(mientras se cumpla } \langle \textit{condición} \rangle\text{)} \end{array} \begin{cases} \langle \textit{enunciado simple} \rangle \\ \langle \textit{enunciado simple} \rangle \\ \ldots \\ \langle \textit{enunciado simple} \rangle \end{cases}$$

y su sintaxis es como se muestra en la figura 4.8 en la página opuesta.

Figura 4.8 Enunciado compuesto while.

SINTAXIS:

⟨*enunciado compuesto while*⟩::=

while (⟨*expresión booleana*⟩ **) {**
⟨*enunciado simple o compuesto*⟩
⟨*enunciado simple o compuesto*⟩
...
⟨*enunciado simple o compuesto*⟩
}

SEMÁNTICA:

Lo primero que hace el programa es evaluar la ⟨*expresión booleana*⟩. Si ésta se evalúa a verdadero, entonces se ejecutan los enunciados que están entre las llaves, y regresa a evaluar la ⟨*expresión booleana*⟩. Si se evalúa a falso, brinca todo el bloque y sigue con el enunciado que sigue al **while**.

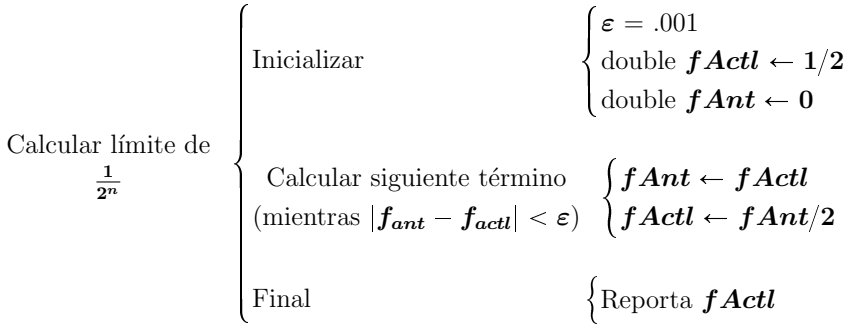
En cualquier iteración que utilicemos, y en particular en la que acabamos de presentar, hay varios aspectos que hay que tener presentes:

I. ¿Cuál es el estado de las variables involucradas cuando se llega por primera vez a evaluar la condición de la iteración?
II. ¿Cuál es el mínimo número de veces que se va a ejecutar el cuerpo de la iteración?
III. ¿Qué es lo que se hace dentro del cuerpo de la iteración que obliga a la iteración a terminar?

En el caso de la iteración while, se debe llevar a cabo un proceso de inicialización, que consiste en, de ser necesario declarar y, asignar valores iniciales que garanticen y dejen claro el estado al llegar a la cabeza de la iteración. Esta iteración puede no ejecutarse, ya que la condición puede no cumplirse desde la primera vez que se intenta iterar; por último, en el cuerpo de la iteración se debe cambiar el estado de una o más de las variables involucradas en la condición, para que haya posibilidad de salir de la iteración.

Un buen ejemplo del uso de esta iteración es, por ejemplo, encontrar el límite de una sucesión dado un margen de error. Lo que haremos será calcular sucesivamente términos, hasta que la diferencia entre el último término calculado y el actual sea menor que una cierta épsilon. El algoritmo para ello se encuentra en la figura 4.9 en la siguiente página.

Figura 4.9 Encontrar el límite de $\frac{1}{2^n}$, dado ε



La implementación de este pequeño método se muestra en el listado 4.12.

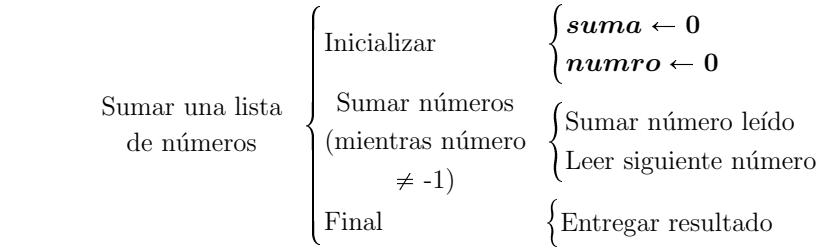
Código 4.12 Cálculo del límite de una sucesión

```
public double limite(double varepsilon)  {
  /* Inicializar */
  double epsilon = varepsilon;
  double fActl = 1 / 2;
  double fAnt  = 1;
  /* Iteración */
  while (Math.abs(fActl - fAnt) < epsilon)  {
    fAnt = FActl;
    fActl = fActl / 2;
  } // fin del while
  /* Final */
  return fActl;
}
```

Como se puede ver, esta iteración es ideal cuando no tenemos claro el número de iteraciones que vamos a llevar a cabo y deseamos tener la posibilidad de no ejecutar el cuerpo ni siquiera una vez. Por ejemplo, si el valor de ε que nos pasaran como parámetro fuera mayor que $\mathbf{1/2}$, la iteración no se llevaría a cabo ni una vez.

Hay ocasiones en que deseamos que un cierto enunciado se ejecute al menos una vez. Supongamos, por ejemplo, que vamos a sumar números que nos den desde la consola hasta que nos den un $-\mathbf{1}$. El algoritmo se puede ver en la figura 4.10.

Figura 4.10 Sumar número mientras no me den un -1



Veamos la descripción de este enunciado compuesto de iteración en la figura 4.11.

Figura 4.11 Enunciado compuesto **do . . . while**

SINTAXIS:

⟨*enunciado compuesto do. . . while*⟩::=

do {
⟨*enunciado simple o compuesto*⟩
⟨*enunciado simple o compuesto*⟩
. . .
⟨*enunciado simple o compuesto*⟩
} **while (** ⟨*expresión booleana*⟩ **);**

SEMÁNTICA:

Lo primero que hace el enunciado al ejecutarse es ejecutar los enunciados que se encuentran entre el do y el while. Es necesario aclarar que estos enunciados no tienen que estar forzosamente entre llaves (ser un bloque) pero las llaves me permiten hacer declaraciones dentro del enunciado, mientras que sin las llaves, como no tengo un bloque, no puedo tener declaraciones locales al bloque. Una vez ejecutado el bloque procede a evaluar la ⟨*expresión booleana*⟩. Si 'esta se evalúa a verdadero, la ejecución continúa en el primer enunciado del bloque; si es falsa, sale de la iteración y sigue adelante con el enunciado que sigue al do . . . while.

También en esta iteración tenemos que tener cuidado en inicializar y declarar

variables necesarias antes de entrar a la iteración. No podemos declararlas dentro porque entonces no las conoce en la ⟨*expresión booleana*⟩. Pero como primero hace y después pregunta, podemos hacer la inicialización como parte del bloque. Veamos cómo queda el pequeño algoritmo que se muestra en la figura 4.11 en el listado 4.13.

Código 4.13 Suma de números leídos

```
public int sumaLeidos(Console cons)  {
  int suma = 0;
  int numero = 0;
  do  {
    suma += numero;
    numero = cons.leeInt();
  }  while (numero != -1);
  return suma;
}
```

Es claro que lo que se puede hacer con un tipo de iteración se puede hacer con la otra. Si queremos que el bloque se ejecute al menos una vez usando un while, lo que hacemos es colocar el bloque inmediatamente antes de entrar a la iteración. Esto nos va a repetir el código, pero el resultado de la ejecución va a ser exactamente el mismo. Por otro lado, si queremos usar un do...while pero queremos tener la posibilidad de no ejecutar ni una vez, al principio del bloque ponemos una condicional que pruebe la condición, y como cuerpo de la condicional colocamos el bloque original. De esta manera si la condición no se cumple al principio el bloque no se ejecuta. Esto quiere decir que si un lenguaje de programación únicamente cuenta con una de estas dos iteraciones, sigue teniendo todo lo necesario para elaborar métodos pensados para la otra iteración.

Tenemos una tercera iteración conocida como for, que resulta ser el cañón de las iteraciones, en el sentido de que en un solo enunciado inicializa, evalúa una expresión booleana para saber si entra a ejecutar el enunciado compuesto e incrementa al final de la ejecución del enunciado compuesto. Lo veremos cuando sea propicio su uso. Por lo pronto volveremos a nuestro problema de manejar una pequeña base de datos.

La lista del curso debemos mostrarla en algún medio. Para usar dispositivos de entrada y salida utilizaremos por lo pronto una clase construida especialmente para ello, se llama Consola y se encuentra en el paquete icc1.interfaz. Tenemos que crear un objeto de tipo Consola, y a partir de ese momento usarlo para mostrar y/o recibir información del usuario. Veamos los principales métodos que vamos a usar por el momento en la tabla 4.3 en la página opuesta.

Podemos regresar ahora al problema de listar todo el curso en una pantalla proporcionada por el usuario, usando la clase Consola y el enunciado compuesto while. La programación se encuentra en el listado 4.14.

Código 4.14 Método que lista todo el curso (**Curso**)

```
153:     /**
154:      *  Lista todo el contenido de la base de datos
155:      *  @param consola el flujo de salida donde va a listar
156:      */
157:     public void listaCurso(Consola consola)  {
158:         int i=1;
159:         while ( i <= numRegs)     {
160:             consola.imprimeln(armaRegistro(i));
161:             i++;
162:          } // while
163:     } // listaCurso
```

Cuadro 4.3 Métodos de la clase Consola.

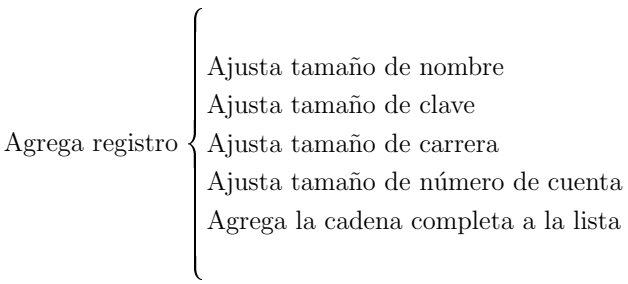
Firma	Descripción
Constructores	
Consola() Consola(String) Consola(int, int) Consola(int, int, String)	Construyen una pantalla para interaccionar con el programa. El valor por omisión (si no tiene argumentos enteros) es de 400×600 en pixeles, o del tamaño que se le indique si se dan argumentos enteros. Si se le proporciona una cadena, esta cadena aparece como título de la pantalla. La pantalla permanece hasta que se destruya con el botón correspondiente. Ningún otro proceso, más que el programa, dará atención mientras la consola esté activa. ¡Cuidado porque lo que se teclee se guarda para cuando desparezca la consola!
Escritura	
void imprime(String) void imprimeln(String)	El primero "coloca" la cadena en la pantalla que corresponde al objeto Consola (lo escribe). El segundo, una vez que terminó de escribir la cadena, manda un carácter de fin de línea. En el primero, una segunda escritura se hará a partir de donde terminó la primera, mientras que con la segunda firma, la siguiente escritura se hará empezando renglón nuevo.

Cuadro 4.3 Métodos de la clase Consola. (continúa)

Firma	Descripción
	Hay algunos carácteres que toman un significado especial. Ya vimos uno de ellos, el tabulador. Veamos algunos más que pueden ser útiles. \n línea nueva \ \\ Una sola \ \" comillas dentro del texto \' Apóstrofe dentro del texto
Lectura	
String leeString() String leeString(String)	Abre una pequeña pantallita en la que pide una cadena. Se selecciona con el ratón la pantallita y se teclea la cadena, terminando con Enter. En la segunda firma se coloca el mensaje como título de la pantallita. Ésta desaparece cuando se termina de teclear la cadena.

El proceso de agregar un estudiante es bastante simple, ya que en las especificaciones del problema no se menciona que la lista se tenga que mantener en orden. Simplemente armamos el registro y lo agregamos al final. El único problema es garantizar el tamaño de los campos, pues el usuario puede no tomarlo en cuenta. El algoritmo se muestra en la figura 4.12.

Figura 4.12 Algoritmo para agregar un estudiante.

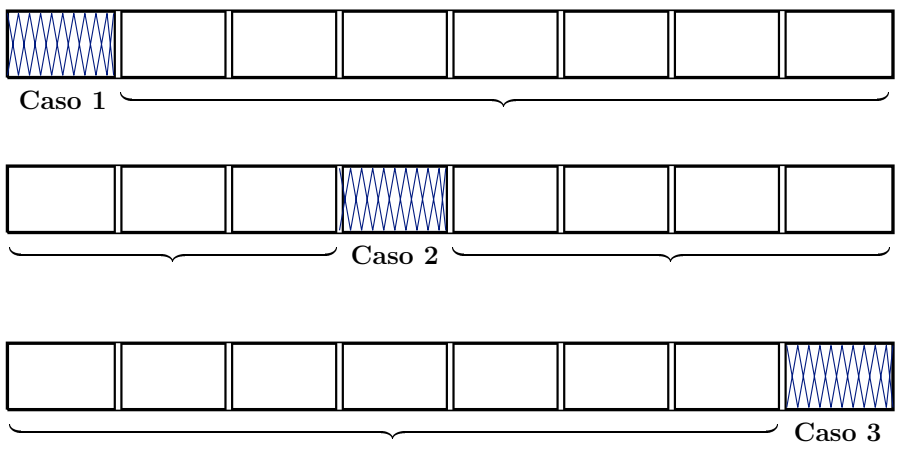


Para ajustar los tamaños de las cadenas, primero les agregamos al final un montón de blancos, para luego truncarla en el tamaño que debe tener. La programación se encuentra en el listado 4.15 en la página opuesta.

Código 4.15 Método que agrega un estudiante a la lista (**Curso**)

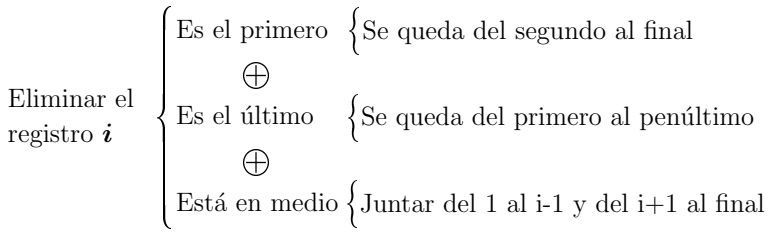
```
164:     /**
165:      * Agrega un estudiante a la base de datos.
166:      * @param nombre
167:      * @param cuenta
168:      * @param clve
169:      * @param carre
170:      */
171:     public void agregaEst (String nombre, String cuenta,
172:                            String clve,  String carre)  {
173:         String blancos = "                              ";
174:         String defNmbre =
175:                 nombre.trim().concat(blancos).substring(0,TAM_NMBRE);
176:         String defCuenta = cuenta.trim().concat(blancos).
177:                 substring(0,TAM_CTA);
178:         String defClave = clve.trim().concat(blancos).
179:                                                 substring(0,TAM_CLVE);
180:         String defCarre =
181:                 carre.trim().concat(blancos).substring(0,TAM_CARRE);
182:         String estudnte = defNmbre + defCarre + defCuenta + defClave;
183:         lista = lista.concat(estudnte);
184:         numRegs++;
185:     }
```

Figura 4.13 Posibles situaciones para eliminar a un registro.



Para eliminar a un estudiante de la lista, usando subcadenas, no es tan fácil. Si el estudiante que queremos quitar es el primero, la lista se convierte en lo que queda de la cadena al quitar el primero; si es el último el que deseamos eliminar, la lista nueva consiste de la parte hasta donde empieza el último; pero en cambio, si al que deseamos eliminar se encuentra en una posición intermedia, tenemos que partir la lista en tres pedazos: lo que va antes, el registro que deseamos eliminar y lo que va después – ver figura 4.13 en la página anterior. De la figura podemos ver que tenemos que hacer un análisis por casos. El algoritmo se muestra en la figura 4.14.

Figura 4.14 Algoritmo para eliminar a un estudiante de la lista.



Conocemos de cual de estas tres situaciones se trata (no hay otra posibilidad) dependiendo de la posición del registro:

- Es el primero si $\boldsymbol{i}$ vale 1.
- Es el último si $\boldsymbol{i}$ vale NUM_REGS.
- Está en medio si $\mathbf{1} < \boldsymbol{i} <$NUM_REGS.

Para este tipo de enunciado es conveniente que introduzcamos el ⟨*enunciado compuesto condicional*⟩ cuya sintaxis y semántica se encuentra en la figura 4.15 en la página opuesta. Con esto podemos ya pasar a programar el método que elimina al $\boldsymbol{i}$-ésimo registro de nuestra lista, en el listado 4.16 en la página opuesta.

El único método que nos falta, para terminar esta sección, es el que arma una lista con todos los registros que contienen una subcadena. En este caso todo lo que tenemos que hacer es, mientras encontremos la subcadena buscada, seguimos buscando, pero a partir del siguiente registro. El algoritmos lo podemos ver en la figura 4.16 en la página 128.

Figura 4.15 Enunciado compuesto condicional

SINTAXIS:

⟨*enunciado compuesto condicional*⟩::=

if (⟨*expresión booleana*⟩) **{**
⟨*enunciado simple o compuesto*⟩
...
⟨*enunciado simple o compuesto*⟩
} **else** **{**
⟨*enunciado simple o compuesto*⟩
...
⟨*enunciado simple o compuesto*⟩
}

SEMÁNTICA:

Lo primero que hace el programa es evaluar la ⟨*expresión booleana*⟩. Si ésta se evalúa a verdadero, entonces se ejecutan los enunciados que están entre las primeras llaves y si se evalúa a falso se ejecutan los enunciados que están a continuación del **else**. Pudiéramos en ambos casos tener un único enunciado, en cuyo caso se pueden eliminar las llaves correspondientes. También podemos tener que no aparezca una cláusula **else**, en cuyo caso si la expresión booleana se evalúa a falso, simplemente se continúa la ejecución con el enunciado que sigue al **if**.

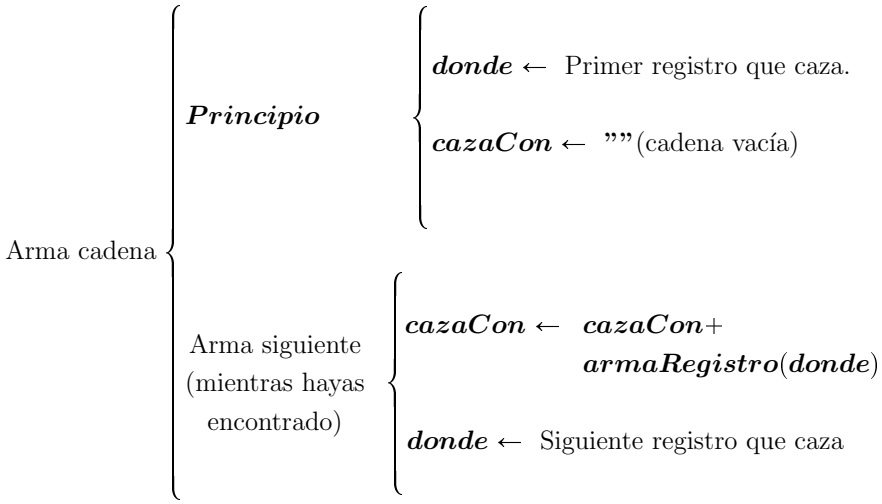
Código 4.16 Método que elimina al registro ***i*** (**Curso**)1/2

```
186:     /**
187:      * Quita al estudiante que se encuentra en el posición i
188:      *
189:      * Distingue entre quitar al primero, al 'ultimo o alguno de
190:      * en medio.
191:      * @param i posición del estudiante en la lista
192:      */
193:     public void quitaEst(int i)  {
194:         int anteriores;
195:         int posteriores;
196:         anteriores = (i - 1) * TAM_REG;      // terminan en
197:         posteriores = i  * TAM_REG;           //empiezan en
```

Código 4.16 Método que elimina al registro i. (Curso)2/2

```
198:        boolean noHay = (i <= 0 || i > numRegs); // posición inválida
199:        if (noHay)   {
200:          return;
201:        }
202:        boolean elPrimero = (i == 1);
203:        boolean elUltimo = (i == numRegs);
204:        if (elPrimero)      lista = lista.substring(posteriores);
205:        else
206:            if (elUltimo)   lista = lista.substring(0,anteriores);
207:            else            lista = lista.substring(0,anteriores)
208:                                  + lista.substring(posteriores);
209:        numRegs--;
210:    }
```

Figura 4.16 Método que encuentra TODOS los que contienen a una subcadena.



Lo único importante en este caso es darnos cuenta que ya tenemos dos versiones que nos dan la posición de una subcadena, así que la programación es prácticamente directa. La podemos ver en el listado 4.17 en la página opuesta.

Código 4.17 Método que lista a los que cazan con ... (Curso)

```
211:     /**
212:      * Construye una lista para mostrar con todos los registros que
213:      * tienen como subcadena al parámetro.
214:      * @param subcad Lo que se busca en cada registro
215:      * @return Una cadena que contiene a los registros que cazan
216:      */
217:     public String losQueCazanCon(String subcad)   {
218:          String cazanCon = "";
219:          int donde = daPosicion(subcad);
220:          while (donde != -1)   {
221:              cazanCon = cazanCon.concat(armaRegistro(donde)+"\n");
222:              donde = daPosicion(subcad,donde);
223:          } //  while encuentres
224:          return cazanCon;
225:     }
```

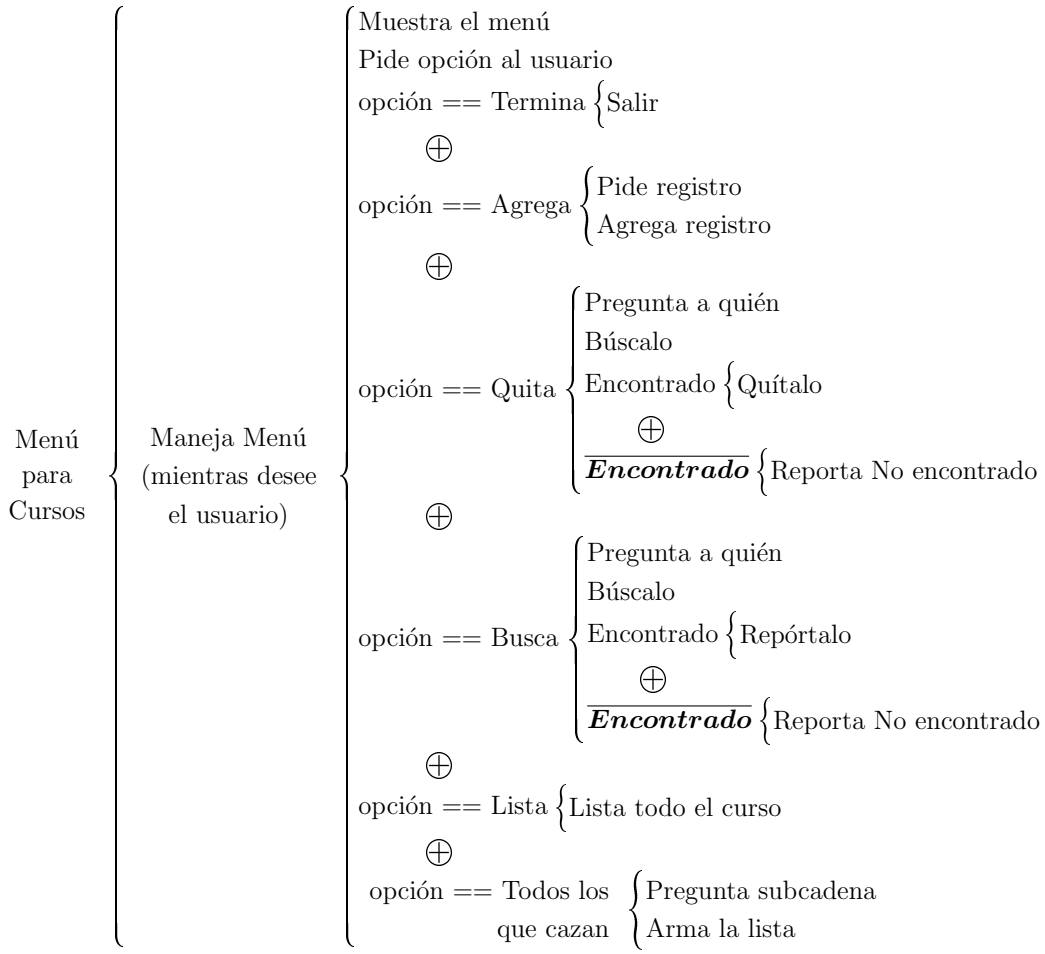
Podríamos, para probar que nuestros métodos funcionan, programar el método main, pero eso es aburrido. Mejor hagamos una clase que maneje un menú para esta base de datos en la siguiente sección.

4.3 Una clase Menu

Contar con una base de datos que tiene las funcionalidades usuales no es suficiente. Debemos contar con algún medio de comunicación con la misma, como pudiera ser un lenguaje de consulta (*query language*) o, simplemente, un menú que nos dé acceso a las distintas operaciones que podemos realizar sobre la base de datos. Construiremos un menú para tal efecto.

El funcionamiento de un menú queda esquematizado en el diagrama de la figura 4.17 en la siguiente página. Básicamente, mientras el usuario lo desee, debemos mostrarle una lista de las opciones que tiene, dejarle que elija una de ellas y proceder a hacer lo que corresponde a la opción.

Figura 4.17 Menú para uso de la clase Curso.



El menú que requerimos corresponde a una clase que va a hacer uso de la clase Curso que ya diseñamos. Por lo tanto, podemos empezar ya su codificación. El hacerlo en una clase aparte nos permitiría, en un futuro, tal vez manejar nuestra base de datos con un lenguaje de consulta o hasta, posiblemente, con un lenguaje de programación.

Para construir el código correspondiente al del diagrama de la Figura 4.17 tenemos en Java una condicional "calculada", en la cual, como su nombre lo indica, elige una opción entre varias dependiendo del valor de la variable selectora. La sintaxis de este enunciado se encuentra en la Figura 4.18.

Figura 4.18 Enunciado **switch**.

Sintaxis:

switch(⟨*expresión*⟩ **) {**
case ⟨$\boldsymbol{valor_1}$⟩:
⟨*lista de enunciados simples o compuestos*⟩;
case ⟨$\boldsymbol{valor_2}$⟩:
⟨*lista de enunciados simples o compuestos*⟩;
case ...
⟨*lista de enunciados simples o compuestos*⟩;
default:
⟨*lista de enunciados simples o compuestos*⟩;
}

Semántica:

Los valores $\boldsymbol{valor_1, \ldots}$ deben ser del mismo tipo que la expresión, a la que llamamos la *selectora* del switch y deben ser constantes de ese tipo. El enunciado switch elige un punto de entrada al arreglo de enunciados de la lista. Evalúa la expresión y va comparando contra los valores que aparecen frente a la palabra case. El primero que coincide, a partir de ese punto ejecuta todos los enunciados desde ahí hasta que se encuentre un enunciado break, o el final del switch, lo que suceda antes. Se acostumbra colocar un enunciado break al final de cada opción para que únicamente se ejecuten los enunciados relativos a esa opción. El switch tiene una cláusula de escape, que puede o no aparecer. Si aparece y la expresión no toma ninguno de los valores explícitamente listados, se ejecuta el bloque correspondiente a default. Si no hay cláusula de escape y el valor de la expresión no caza con ninguno de los valores en los cases, entonces el programa abortará dando un error de ejecución.

El comando break que mencionamos en la figura 4.18 es muy sencillo, y lo único que hace es sacar el hilo de ejecución del programa hacia afuera del switch. Su sintaxis y semántica se encuentran definidas con más precisión en la figura 4.19. Veamos algunos ejemplos de la forma que podrían tomar distintos switches.

Figura 4.19 Enunciado **break**.

SINTAXIS:

⟨*enunciado break*⟩ ::= **break**

SEMÁNTICA:

Hace que el hilo de ejecución del programa no siga con el siguiente enunciado, sino que salga del enunciado compuesto en el que está, en este caso el switch.

```
boolean esta = i != -1;
switch (esta)  {
case false:
      consola.imprimeln("NO␣está␣");
      break;
case true:
      consola.imprimeln("SI␣está");
}
```

Como las únicas dos posibilidades para el selector del switch, una expresión booleana, es falso o verdadero, no hay necesidad de poner una cláusula de escape, ya que no podrá tomar ningún otro valor. Este ejemplo es realmente un enunciado if disfrazado, ya que si la expresión se evalúa a verdadero (true) se ejecuta lo que corresponde al caso etiquetado con true, mientras que si se evalúa a falso, se ejecuta el caso etiquetado con false. Programado con ifs queda de la siguiente manera:

```
boolean esta = i != -1;
if (esta)
    consola.imprimeln("SI␣está");
else
    consola.imprimeln("SI␣está");
```

Otro ejemplo más apropiado, ya que tiene más de dos opciones, es el siguiente: supongamos que tenemos una clave que nos dice el estado civil de una persona y que queremos convertirlo a la cadena correspondiente. Las claves son:

s: soltero
c: casado
d: divorciado
v: viudo
u: unión libre

En una variable de tipo carácter tenemos una de estas opciones, y deberemos

proporcionar una cadena con el texto adecuado. La programación quedaría algo parecido a lo que se observa en el listado 4.18.

Código 4.18 Ejemplo de identificación del estado civil de un individuo

```
1:    char estado = daEstado();
2:              /* daEstado() asigna valor a esta variable */
3:    String texto;
4:    switch(estado) {   /* Elegimos de acuerdo al contenido
5:                          de estado  */
6:     case 's':
7:        texto = "soltero";
8:        break;
9:     case 'c':
10:       texto = "casado";
11:       break;
12:    case 'd':
13:       texto = "divorciado";
14:       break;
15:    case 'u':
16:       texto = "unión␣libre";
17:       break;
18:    default:
19:       texto = "no␣definido";
20:       break;
21:  } // En la variable texto queda el mensaje que corresponde
```

En general, la estructura switch lo que hace es ir comparando el valor del selector con cada uno de las constantes que se encuentran alrededor de la palabra case. Más de una etiqueta, se pueden aparecer frente a un bloque de código, para referirse a que esa es la acción a realizarse en más de un caso:

```
case 'A': case 'B': case C:
    return "Primeros␣tres␣casos''
```

Regresamos ahora a la programación de nuestro problema. La parte correspondiente al manejo del menú se da en un método, para no atiborrar al método principal (main) de la clase con código. La programación queda como se muestra en el listado 4.19 en la siguiente página.

Código 4.19 Encabezado de la clase **Menu** y el método **daMenu** (**MenuCurso**) 1/2

```
1: class MenuCurso   {
2:     /**
3:      * Maneja el menú para el acceso y manipulación de la base de
4:      * datos.
5:      * @param cons Dispositivo en el que se va a interaccionar
6:      * @param miCurso Base de datos con la que se va a trabajar
7:      * @return la opción seleccionada, después de haberla ejecutado
8:      */
9:     public int daMenu(Consola cons, Curso miCurso)    {
10:         String nombre, cuenta, carrera, clave;
11:         String subcad;
12:         String registros;
13:         int donde;
14:         String menu = "(0)\tTermina\n"+
15:                       "(1)\tAgrega\n"+
16:                       "(2)\tQuita\n"+
17:                       "(3)\tBusca\n"+
18:                       "(4)\tLista todos\n"+
19:                       "(5)\tLista subconjunto que contenga a ....";
20:         cons.imprimeln(menu);
21:         int opcion;
22:         String sopcion = cons.leeString("Elige una opción -->");
23:         opcion = "012345".indexOf(sopcion);
24:         switch(opcion) {
25:           case 0: // Salir
26:              cons.imprimeln("Espero haberte servido.\n"+
27:                             "Hasta pronto...");
28:              return -1;
29:           case 1: // Agrega Estudiante
30:              nombre = pideNombre(cons);
31:              cuenta = pideCuenta(cons);
32:              carrera = pideCarrera(cons);
33:              clave = pideClave(cons);
34:              miCurso.agregaEst(nombre, cuenta, clave, carrera);
35:              return 1;
36:           case 2: // Quita estudiante
37:              nombre = pideNombre(cons);
38:              donde = miCurso.daPosicion(nombre);
39:              if ( donde != -1)   {
40:                  nombre = miCurso.daNombre(donde);
41:                  miCurso.quitaEst(donde);
42:                  cons.imprimeln("El estudiante:\n\t"
43:                                 + nombre
44:                                 + "\n Ha sido eliminado ");
45:              }
```

Código 4.19 Encabezado de la clase bfseries Menu y el método **daMenu** (**MenuCurso**) 2/2

```
46:             else {
47:                 reportaNo(cons, nombre);
48:             }
49:             return 2;
50:          case 3: // Busca subcadena
51:             subcad = cons.leeString("Dame␣la␣subcadena␣a␣"
52:                             + "buscar:␣");
53:             donde = miCurso.daPosicion(subcad);
54:               if (donde != -1) {
55:                  cons.imprimeln(miCurso.armaRegistro(donde));
56:               }
57:               else  {
58:                  reportaNo(cons, subcad);
59:               }
60:               return 3;
61:           case 4: // Lista todos
62:               miCurso.listaCurso(cons);
63:               return 4;
64:           case 5: // Lista con criterio
65:               subcad = cons.leeString("Da␣la␣subcadena␣que␣" +
66:                               "␣quieres␣contengan␣los␣" +
67:                               "registros:");
68:               registros = miCurso.losQueCazanCon(subcad);
69:               if (registros.equals(""))  {
70:                  cons.imprimeln("No␣hubo␣ningún␣registro␣con␣" +
71:                                       "este␣criterio");
72:               }
73:               else  {
74:                  cons.imprimeln(registros);
75:               }
76:               return 5;
77:           default: // Error, vuelve a pedir
78:               cons.imprimeln("No␣diste␣una␣opción␣válida.\n" +
79:                               "Por␣favor␣vuelve␣a␣elegir.");
80:               return 0;
81:          }
82:       }
```

Para el menú construimos una cadena que se muestra separada por renglones, como se observa en las líneas68 a 73, y le pedimos al usuario que elija un número de opción, en la línea 76. En esta línea aparece algo nuevo, ya que estamos interaccionando con el usuario. La manera de hacerlo es a través de una consola (por eso aparece un objeto de esta clase como parámetro). Por lo pronto vamos a leer cadenas, y tenemos la opción de pedirle o no que escriba una cadena como

título de la pequeña pantalla en la que solicita el dato. Las dos posible firmas del método de lectura de cadenas de la clase consola son

```
String leeString()
String leeString(String)
```

En el primer caso simplemente aparecerá una pantallita en la que, una vez seleccionada, se deberá teclear la cadena que se desea proporcionar (recuérdese que podemos tener una cadena de un solo carácter) terminándola con Enter. La invocación de este método, desde un objeto del tipo Consola puede aparecer en cualquier lugar donde pudiera aparecer una expresión de cadenas.

Estamos asumiendo que el usuario nos tecleó un dígito. Procedemos a ver exactamente cuál de ellos fue, buscándolo en una cadena que contiene todas las opciones (si tuviéramos más de 10 opciones tendríamos que asignar letras para las siguientes, y así poder seguir utilizando este método). Esto lo hacemos con el método ya conocido por nosotros, indexOf – línea 77 del listado 4.19 en la página 134.

Una vez determinada la opción que solicitó el usuario, deberemos escoger entre un conjunto de opciones, numeradas del 0 al 5. Para ello vamos a utilizar una condicional especial, el switch, que se muestra en la figura 4.18 en la página 131.

En el caso de nuestro problema, cada bloque del switch termina con un return, ya que ése es el objetivo del proceso, avisar cuál es la última opción que se eligió. Adicionalmente, se ejecuta lo correspondiente a cada opción. Las revisaremos una por una.

4.3.1. Salir

En esta opción se emite un mensaje para el usuario, avisándole del final del proceso, y se regresa un valor de -1 para que el programa principal ya no siga mostrando el menú y termine.

4.3.2. Agrega estudiante

Esta opción tiene que funcionar como una interface entre el método de la base de datos y el usuario, para agregar de manera correcta a un estudiante. Por ello, deberemos primero llenar cada uno de los campos del registro (sería absurdo pedirle al usuario que conociera cómo está implementada nuestra base de datos).

Por ello se procede a solicitarle al usuario cada uno de los campos involucrados. Elegimos hacer un método distinto para cada campo, para poder indicarle al usuario que tipo de cadena estamos esperando. La codificación de cada uno de estos métodos se encuentran en el listado 4.20. Algunos de estos métodos los volveremos a usar, ya que nos proporcionan, por parte del usuario, información. Cada uno de estos métodos simplemente le dice al usuario qué es lo que espera y recibe una cadena, que, idealmente, deberá ser lo que el usuario espera.

Código 4.20 Métodos para agregar un estudiante a la base de datos. (**MenuCurso**)1/2

```
83:     /**
84:      * Pide en el input stream el nombre del alumno.
85:      * @param cons el input stream
86:      * @return El nombre leído
87:      */
88:     private String pideNombre(Consola cons)     {
89:         String nombre = cons.leeString("Dame␣el␣nombre␣del␣"
90:                 + "estudiante,␣empezando␣por␣apellido␣paterno:");
91:         return nombre;
92:     }
93:     /**
94:      * Pide en el input stream la carrera del alumno.
95:      * @param cons el input stream
96:      * @return La carrera leída
97:      */
98:     private String pideCarrera(Consola cons)     {
99:         String carrera = cons.leeString("Dame␣una␣de␣las␣"
100:                + "carreras:\tMate\tComp\tFisi\tActu\tBiol:");
101:        return carrera;
102:    }
103:    /**
104:     * Pide en el input stream la clave de acceso del alumno.
105:     * @param cons el input stream
106:     * @return La clave de acceso leída
107:     */
108:    private String pideClave(Consola cons)     {
109:        String clave = cons.leeString("Dame␣la␣clave␣de␣acceso␣del␣"
110:                                + "estudiante:");
111:        return clave;
112:    }
```

Código 4.20 Métodos para agregar un estudiante a la base de datos (**MenuCurso**) 2/2

```
113:     /**
114:      * Pide en el input stream el numero de cuenta del alumno.
115:      * @param cons el InputStream
116:      * @return El numero de cuenta leído
117:      */
118:     private String pideCuenta(Consola cons)     {
119:         String cuenta =
120:               cons.leeString("Dame el numero de cuenta del "
121:                       + "estudiante, de 9 dígitos");
122:         return cuenta;
123:     }
```

4.3.3. Quita estudiante

En esta opción, también, la parte interesante la hace el método de la base de datos, que ya revisamos. Debemos notar, sin embargo, que es acá donde vamos a verificar que las operaciones sobre la base de datos se hayan realizado de manera adecuada, preguntando siempre por el resultado de las mismas. El único método que no hemos visto es uno realmente sencillo que reporta que el estudiante solicitado no se encuentra en la base de datos. La programación de este método se encuentra en el listado 4.21.

Código 4.21 Método que reporta estudiante inexistente (**MenuCurso**)

```
124:     /**
125:      * Reporta que el estudiante buscado no fue localizado.
126:      * @param cons flujo de salida en el que se escribe
127:      * @param nombre subcadena que no fue localizada
128:      */
129:     private void reportaNo(Consola cons, String nombre)     {
130:         cons.imprimeln ("El estudiante :\n\t"
131:                         + nombre
132:                         + "\n No esta en el grupo");
133:     }
```

4.3.4. Busca estudiante

También realiza su tarea usando métodos que ya explicamos. Al igual que las otras opciones, verifica que las operaciones sobre la base de datos se realicen de manera adecuada.

4.3.5. Lista todo el curso

Simplemente invoca al método correspondiente de la base de datos.

4.3.6. Lista los que cumplan con algún criterio

Este método verifica si algún estudiante cumplió o no con el criterio solicitado, preguntando si la cadena resultante tiene algo o no. Noten que tiene que usar el método equals(String), ya que de otra manera no estaría comparando contenidos de cadenas y nos diría siempre que no son iguales.

4.3.7. Valor por omisión

En este caso, regresaremos un valor que permita al usuario saber que no dio una opción correcta y que debe volver a elegir.

4.3.8. Método principal de MenuCurso

En este método es en el que tenemos que declarar nuestros objetos, tanto la base de datos como el menú. Una vez hecho esto, simplemente entraremos en una iteración mostrando el menú y recibiendo la opción, mientras el usuario no decida terminar. La programación de este método se encuentra en el listado 4.22 en la siguiente página.

Lo único interesante del método en el listado 4.22 en la siguiente página es que el enunciado del while es un enunciado vacío, ya que todo se hace cuando se invoca al menú para que diga si ya le pidieron que salga o no.

Código 4.22 Método principal de la clase **MenuCurso** (**MenuCurso**)

```
134:    public static void main(String[] args)    {
135:        int opcion;
136:        Consola consola = new Consola();
137:        Curso miCurso =  new Curso(
138:                "7001",                          // Curso
139:                   "Aguilar␣Solís␣Aries␣Olaf␣" + "Mate"
140:                           + "400001528" + "aaguilar␣␣"
141:                 + "Cruz␣Cruz␣Gil␣Noé␣␣␣␣␣␣␣␣" + "Comp"
142:                           + "098034011" + "ncruz␣␣␣␣␣"
143:                 + "García␣Villafuerte␣Israel" + "Mate"
144:                           + "098159820" + "igarcia␣␣␣"
145:                 + "Hubard␣Escalera␣Alfredo␣␣" + "Comp"
146:                           + "099586197" + "ahubard␣␣␣"
147:                 + "Tapia␣Vázquez␣Rogelio␣␣␣␣" + "Comp"
148:                           + "098162075" + "rtapia␣␣␣␣", // lista
149:             5);
150:        MenuCurso miMenu = new MenuCurso();
151:        while ((opcion = miMenu.daMenu(consola, miCurso)) != -1);
152:    }
```

Con esto damos por terminada nuestra primera aproximación a una base de datos. En lo que sigue haremos más eficiente y flexible esta implementación, buscando que se acerque más a lo que es una verdadera base de datos.

Datos estructurados 5

La implementación que dimos en el capítulo anterior a nuestra base de datos es demasiado alejada de como abstraemos la lista del grupo. Realmente, cuando pensamos en una lista es una cierta colección de registros, donde cada registro tiene uno o más campos. Tratemos de acercarnos un poco más a esta abstracción.

El tipo de colección que usaremos en esta ocasión es una *lista*. La definición de una lista es la siguiente:

Definición 5.6 Una **lista** es:

(a) La lista vacía, aquella que no tiene ningún elemento; o bien

(b) el primer elemento de la lista, seguido de una lista.

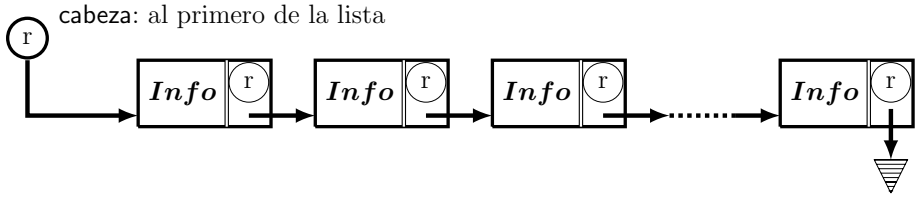
Por ejemplo, si una lista nada más tiene un elemento, ésta consiste del primer elemento de una lista, seguido de una lista vacía.

Toda lista es una referencia a objetos de cierto tipo que contienen determinada información y donde al menos uno de sus campos es una referencia, para acomodar allí a la lista que le sigue. En Java, si una lista es la lista vacía tendrá el valor de `null`, que corresponde a una referencia nula. Generalmente representamos las listas como se muestra en la figura 5.1 en la siguiente página, con la letra "r" representando un campo para guardar allí una referencia, y el símbolo  representando que no sigue nadie (una referencia `nula`).

Como la definición de la lista es recursiva, debemos siempre tener "anclado"

al primer elemento de la lista, la cabeza de la misma, como se puede ver en la figura 5.1.

Figura 5.1 Ilustración de una lista



Otros nombres que se usan en la literatura sinónimos de referencia son liga, apuntador, cadena[1].

5.1 La clase para cada registro

Como en el capítulo anterior, queremos que cada estudiante de la lista tenga un campo para nombre, cuenta, carrera y clave, pero en esta ocasión lo haremos independiente cada uno. Además requerimos el campo donde se guarde la referencia al siguiente de la lista. Esta referencia es una referencia a objetos de la misma clase que los registros: es autoreferida. Entonces, la clase que vamos a tener para nuestros registros la llamaremos Estudiante. Veamos por lo pronto los atributos (datos) que tiene esta clase en el listado 5.1.

Código 5.1 Atributos de la clase **Estudiante**

```
1: import icc1.interfaz.Consola;
2:  /**
3:   * Base de datos, a base de listas de registros, que emula la
4:   * lista de un curso de licenciatura. Tiene las opciones
5:   * normales de una base de datos y funciona mediante un Menú
6:   */
7: class Estudiante   {
8:     private String nombre, clave, carrera, cuenta;
9:     private Estudiante siguiente;
10:    static public final int NOMBRE = 1;
11:    static public final int CUENTA = 2;
12:    static public final int CARRERA = 3;
13:    static public final int CLAVE = 4;
```

[1]Del inglés, *chain*.

Combinamos en una sola declaración los 4 campos (atributos) de información que tiene el registro, ya que todos son del mismo tipo; el atributo siguiente es el que es una referencia al siguiente de la lista (a la lista que le sigue). Deben observar que este campo es de tipo Estudiante, que, como todas las variables del tipo de alguna clase, constituyen referencias.

Todas las constantes lo son de la clase (static final). Eso quiere decir que podrán ser accesibles desde cualquiera de los objetos, y sólo existe una copia, la de la clase.

Debemos tener un constructor que inicialice a un objeto de esta clase con las cadenas adecuadas. Como tenemos al menos un constructor, si queremos uno sin parámetros también lo tenemos que programar nosotros. Los constructores se encuentran en el listado 5.2.

Código 5.2 Constructores para la clase **Estudiante** (**Estudiante**)

```
14:     /**
15:      * Constructor sin parámetros
16:      */
17:     public Estudiante()    {
18:           nombre = carrera = clave = cuenta = null;
19:     }
20:     /**
21:      * Constructor a partir de datos de un estudiante.
22:      * los campos vienen separados entre sí por comas,
23:      * mientras que los registros vienen separados entre sí
24:      * por punto y coma.
25:      * @param String, String, String, String los valores para
26:      *           cada uno de los campos que se van a llenar.
27:      * @return Estudiante una referencia a una lista
28:      */
29:     public Estudiante(String nmbre, String cnta,
30:                       String clve,  String crrera)  {
31:         nombre = nmbre.trim();
32:         cuenta = cnta.trim();
33:         clave = clve.trim();
34:         carrera = crrera.trim();
35:         siguiente = null;
36:     }
```

Una vez que tenemos los constructores, tenemos que proveer, para cada campo al que queramos que se tenga acceso, un método de consulta y uno de actualización. La programación de estos métodos se encuentra en el listado 5.3 en la siguiente página.

Código 5.3 Métodos de acceso y actualización de la clase **Estudiante** 1/2

```
37:     /**
38:      * Regresa el contenido del campo nombre.
39:      *  @return String
40:      */
41:     public String getNombre()   {
42:         return nombre;
43:     }
44:     /**
45:      * Actualiza el campo nombre.
46:      * @param String el valor que se va a asignar.
47:      */
48:     public void setNombre(String nombre)    {
49:         this.nombre = nombre;
50:     }
51:     /**
52:      * Regresa el contenido del campo carrera.
53:      *  @return String
54:      */
55:     public String getCarrera()    {
56:        return carrera;
57:     }
58:     /**
59:      * Actualiza el campo carrera.
60:      * @param String el valor que se va a asignar.
61:      */
62:     public void setCarrera(String carre)    {
63:         carrera = carre;
64:     }
65:     /**
66:      * Regresa el contenido del campo cuenta.
67:      *  @return String
68:      */
69:     public String getCuenta()   {
70:         return cuenta;
71:     }
72:     /**
73:      * Actualiza el campo cuenta.
74:      * @param String el valor que se va a asignar.
75:      */
76:     public void setCuenta(String cnta)    {
77:         cuenta = cnta;
78:     }
```

Código 5.3 Métodos de acceso y actualización de la clase **Estudiante** 2/2

```
79:     /**
80:      * Regresa el contenido del campo calve.
81:      * @return String
82:      */
83:     public String getClave()    {
84:         return clave;
85:     }
86:     /** Actualiza el campo clave.
87:      * @param String el valor que se va a asignar.
88:      */
89:     public void setClave(String clve)  {
90:         clave = clve;
91:     }
92:     /** Regresa el contenido del campo siguiente.
93:      * Corresponde a la lista que sigue a este elemento.
94:      * @return Estudiante Una referencia al siguiente objeto de la
95:      * lista
96:      */
97:     public Estudiante getSiguiente()    {
98:         return siguiente;
99:     }
100:    /** Actualiza el campo siguiente.
101:     * @param Estudiante la referencia que se va a asignar.
102:     */
103:    public void setSiguiente(Estudiante sig)    {
104:        siguiente = sig;
105:    }
```

Podemos también poner métodos más generales, dado que todos los atributos son del mismo tipo, que seleccionen un campo a modificar o a regresar. Los podemos ver en el código 5.4.

Código 5.4 Métodos que arman y actualizan registro completo **(Estudiante)**1/2

```
106: class Estudiante {
107:     /**
108:      * Regresa el campo solicitado.
109:      * @param cual identificador del campo solicitado
110:      * @return La cadena con el campo solicitado
111:      */
112:     public String getCampo(int cual) {
113:         switch (cual) {
114:         case NOMBRE:    return getNombre();
115:         case CUENTA:    return getCuenta();
```

Código 5.4 Métodos que arman y actualizan registro completo **(Estudiante)** 2/2

```
116:         case CARRERA:    return getCarrera();
117:         case CLAVE:      return getClave();
118:         default:         return "Clave invalida";
119:         } // end of switch (cual)
120:     }
121:     /**
122:      * Modifica el campo solicitado.
123:      * @param cuál el campo solicitado.
124:      * @param cadena el nuevo valor
125:      */
126:     public void setCampo(int cual, String cadena) {
127:         switch (cual) {
128:         case NOMBRE:  setNombre(cadena);
129:             break;
130:         case CUENTA:  setCuenta(cadena);
131:             Break;
132:         case CARRERA: setCarrera(cadena);
133:             break;
134:         case CLAVE:   setClave(cadena);
135:             break;
136:         default:
137:             break;
138:         } // end of switch (cual)
139:     }
```

Finalmente, de esta clase únicamente nos faltan dos métodos, uno que regrese todo el registro armado, listo para impresión, y uno que actualice todos los campos del objeto. Podemos ver su implementación en el listado 5.5.

Código 5.5 Métodos que arman y actualizan registro completo (2) **(Estudiante)**1/2

```
140:     /** Arma una cadena con el contenido de todo el registro.
141:          * @return String El registro armado.
142:      */
143:     public String getRegistro()    {
144:         return nombre.trim()+"\t"+
145:             cuenta.trim()+"\t"+
146:             carrera.trim()+"\t"+
147:             clave.trim();
148:     }
```

Código 5.5 Métodos que arman y actualizan registro completo (2) **(Estudiante)**2/2

```
149:     /** Actualiza todo el registro de un jalón.
150:      * @param  String  el nombre,
151:      *                 String cuenta
152:      *                 String carrera
153:      *                 String clave.
154:      */
155:     public void setRegistro(String nmbre, String cnta,
156:                             String clve, String crrera)    {
157:         nombre = nmbre.trim();
158:         cuenta = cnta.trim();
159:         clave = clve.trim();
160:         carrera = crrera.trim();
161:     }
```

Como se puede observar, no hay necesidad de "diseñar" los métodos previo a su programación, ya que son, en general, muy pequeños y concisos. Es importante notar que en esta clase no interviene ningún método que maneje ningún tipo de colección de objetos, sino únicamente lo que concierne a cada registro en sí mismo.

5.2 La lista de registros

Como acabamos de mencionar, la interrelación entre los distintos registros de nuestra base de datos la vamos a manejar desde una clase separada a la de los registros mismos, la *lista de registros*.

Para toda lista requerimos, como mencionamos en relación con la figura 5.1 en la página 142, un "ancla" para la lista, algo que sea la referencia al primer elemento de la lista. De no tener esta referencia, como la lista se va a encontrar en el heap, no habrá manera de saber dónde empieza o dónde están sus elementos. Hagan de cuenta que el heap es el mar, y las declaraciones en una clase son un barco. Cada vez que se crea un objeto, éste se acomoda en el heap (mar), por lo que tenemos que tener una cuerda (cadena, referencia, apuntador) desde el barco hacia el primer objeto, y de éste al segundo, etc. De esa manera, si "jalamos la cuerda" podemos tener acceso a todos los elementos que conforman la lista. "Jalar la cuerda" quiere decir, en este contexto, ir tomando referencia por referencia, como si se tratara de un collar de cuentas. Al igual que en nuestra implementación anterior, usaremos un identificador lista que nos indique el primer elemento de la

lista, una referencia al primer elemento. Por supuesto que esta lista estará vacía en tanto no le agreguemos ningún objeto del tipo Estudiante. Por lo tanto, la única diferencia entre las declaraciones de nuestra implementación anterior y ésta es el tipo de la lista, quedando las primeras líneas de la clase como se puede apreciar en el listado 5.6.

Código 5.6 Atributos de la clase **ListaCurso**

```
1: import icc1.interfaz.Consola;
2: /**
3:  * Restructura a la clase Curso para que sea
4:  * manejada a través de una lista
5:  */
6: class ListaCurso    {
7:       /* La cabeza de la lista:   */
8:     private Estudiante lista;
9:       /* El número de grupo      */
10:    private String grupo;
11:      /* Cuenta el número de registros */
12:    private int numRegs;
```

Al igual que con la clase Curso, tenemos dos constructores, uno que únicamente coloca el número del grupo y otro inicializa la lista y coloca el número de grupo. Debemos recordar que lo primero que hace todo constructor es poner los valores de los atributos que corresponden a referencias en null y los que corresponden a valores numéricos en 0. Los constructores se encuentran en el listado 5.7.

Código 5.7 Constructores para la clase **ListaCurso** 1/2

```
13:    /** Constructor que únicamente asigna número de grupo a la lista.
14:     * @param String gr El número del grupo
15:     */
16:    public ListaCurso(String gr)    {
17:        lista = null;
18:        grupo = gr;
19:        numRegs = 0;
20:    }
```

Código 5.7 Constructores para la clase **ListaCurso** 2/2

```
21:     /** Constructor que asigna el número de grupo y una sublista}
22:      * inicial.
23:      * @param Estudiante sublista.
24:      * @param String gr Número del grupo.
25:      */
26:     public ListaCurso(Estudiante iniciales, String gr)    {
27:         grupo = gr;
28:         lista = iniciales;
29:         numRegs = cuentaRegs();
30:     }
```

El segundo constructor tiene dos parámetros; el primero de ellos es una referencia a un objeto de la clase Estudiante y el segundo es una referencia a una cadena. El constructor inicializa la lista con esta referencia (copia la referencia a la variable lista) y asigna el número de grupo.

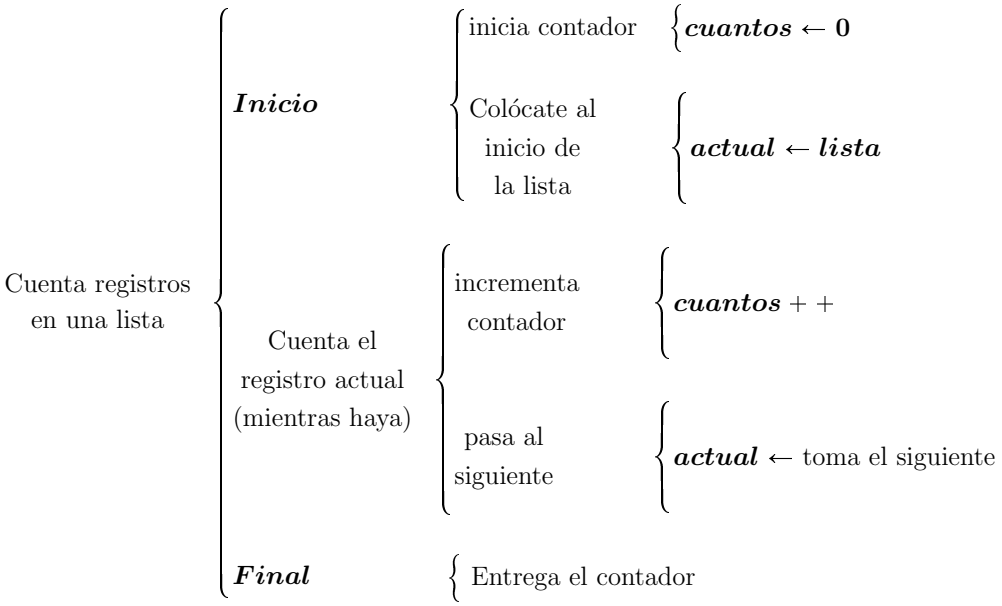
Dado que tenemos 3 atributos, tenemos que definir tres métodos de acceso a los atributos. Dos de ellos son muy sencillos: el que me regresa el número del grupo y el que me regresa la cabeza de la lista, pues éstos únicamente regresan el valor del atributo. Los podemos ver en el listado 5.8.

Código 5.8 Métodos que dan valores de los atributos (**ListaCurso**)

```
31:     /** Regresa la lista de estudiantes.
32:      * @return Estudiante La cabeza de la lista.
33:      */
34:     public Estudiante getLista()    {
35:         return lista;
36:     }
37:    /** Proporciona el número de grupo.
38:      * @return String El grupo
39:      */
40:     public String getGrupo()      {
41:         return grupo;
42:     }
```

Para regresar el número de registros en la lista, preferimos contarlos “a pie”. Todo lo que tenemos que hacer es colocarnos al principio de la lista e incrementar u8n contador por cada registro al que podamos llegar. Podemos ver un diagrama de Warnier-Orr muy sencillo en la figura 5.2 en la siguiente página y su programación correspondiente en el listado 5.9 en la siguiente página.

Figura 5.2 Contando los registros de una lista

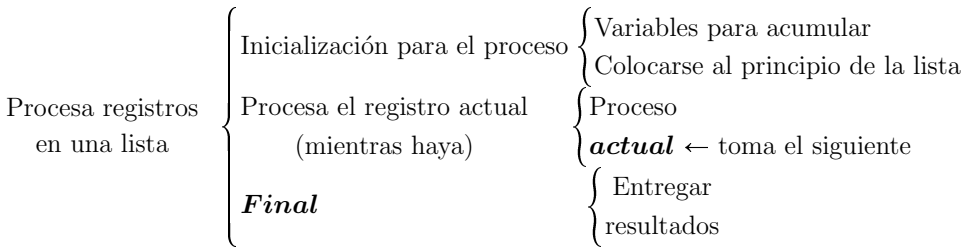


Código 5.9 Recorrido de una lista para contar sus registros (**ListaCurso**)

```
43:     /** Calcula el número de registros en la lista.
44:      * @return int El número de registros
45:      */
46:     public int getNumRegs()     {
47:         int cuantos = 0;
48:         Estudiante actual = lista;
49:         while ( actual != null) {
50:             cuantos++;
51:             actual = actual.getSiguiente();
52:         }
53:         return cuantos;
54:     }
```

Este método nos da el patrón que vamos a usar casi siempre para recorrer una lista, usando para ello la referencia que tiene cada registro al registro que le sigue. El patrón general es como se muestra en la figura 5.3.

Figura 5.3 Procesando los registros de una lista



Junto con los métodos de acceso, debemos tener métodos que alteren o asignen valores a los atributos. Sin embargo, tanto el atributo numRegs como lista deberán ser modificados por las operaciones de la base de datos, y no directamente. En cambio, podemos querer cambiarle el número de grupo a una lista. Lo hacemos simplemente indicando cuál es el nuevo número de grupo. Podemos ver este método en el listado 5.10.

Código 5.10 Modifica el número de grupo (**ListaCurso**)

```
55:     /** Modifica el número de grupo.
56:      * @param String gr  El nuevo número de grupo.
57:      */
58:     public void setGrupo(String gr)    {
59:         grupo = gr;
60:     }
```

Podemos empezar ya con las funciones propias de la base de datos, como por ejemplo, agregar un estudiante a la lista. Realmente podríamos agregarlo de tal manera que se mantenga un cierto orden alfabético, o simplemente agregarlo en cualquier posición de la misma. Haremos esto último, y el mantener la lista ordenada se quedará como ejercicio.

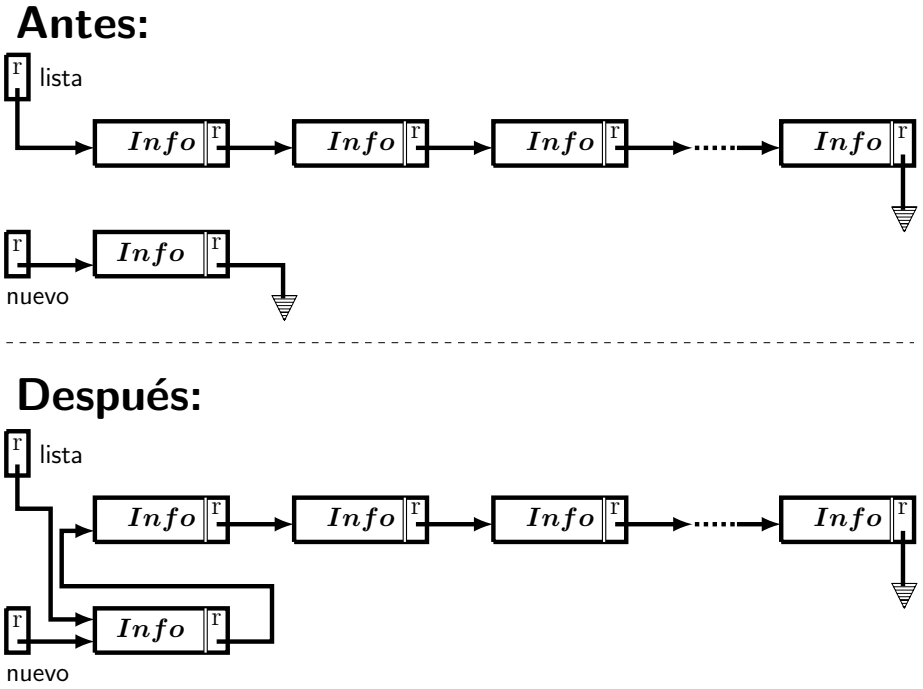
Hay dos posiciones "lógicas" para agregar un registro. La primera de ellas y la más sencilla es al inicio de la lista. Básicamente lo que tenemos que hacer es poner al registro que se desea agregar como el primer registro de la lista. Esto quiere

decir que la nueva lista consiste de este primer registro, seguido de la vieja lista. Veamos en la figura 5.4 qué es lo que queremos decir. El diagrama de Warnier-Orr para hacer esto está en la figura 5.5 y la programación del método se encuentra en el listado 5.11.

Figura 5.4 Agregando al principio de la lista

Agrega registro al principio
- Pon a nuevo a apuntar a lista
- Pon a lista a apuntar a nuevo

Figura 5.5 Esquema del agregado un registro al principio de la lista

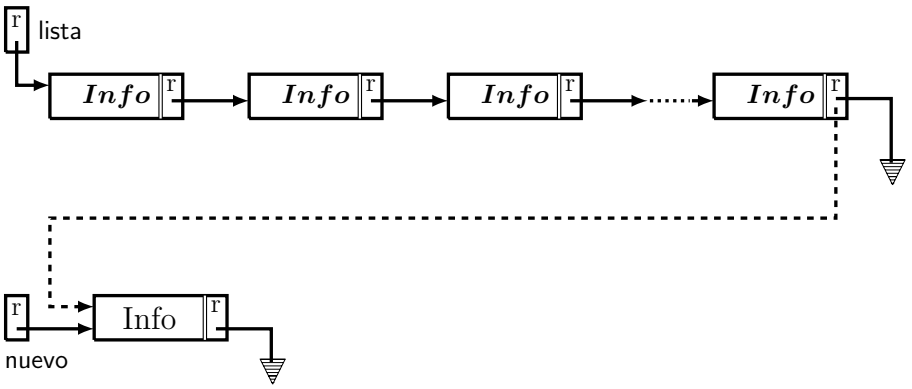


Código 5.11 Agregar un registro al principio de la lista (**ListaCurso**)

```
61:     /** Agrega un registro al principio de la lista
62:      * @param Estudiante nuevo A quien se va a agregar
63:      */
64:     public void agregaEst(Estudiante nuevo)  {
65:         nuevo.setSiguiente(lista);
66:         lista = nuevo;
67:     }
```

Si lo quisiéramos acomodar al final de la lista las cosas se complican un poco más, aunque no demasiado. Queremos hacer lo que aparece en la figura 5.6.

Figura 5.6 Agregando al final de la lista



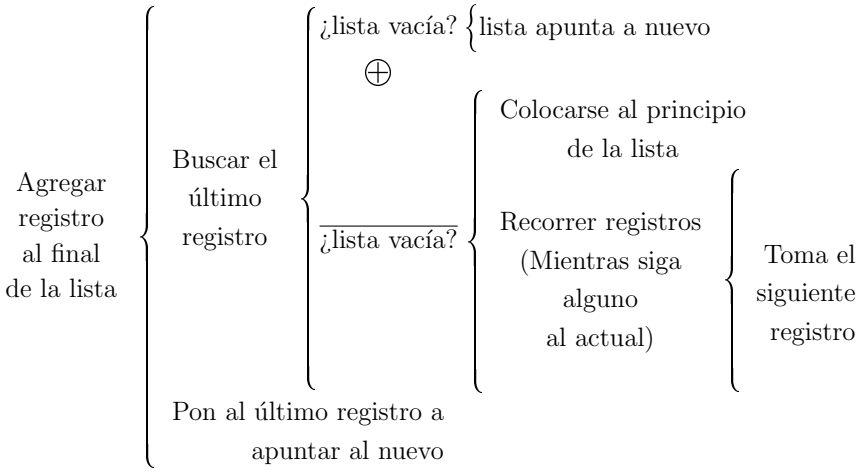
Lo que tenemos que hacer es localizar en la lista al registro al que ya no le sigue ninguno, y poner a ese registro a apuntar al nuevo. El algoritmo queda como se muestra en la figura 5.7 en la siguiente página.

En este algoritmo la condición para que pare la iteración es que al registro actual no le siga nada. Esto, traducido a nuestro entorno, equivale a la expresión booleana

```
actual.getSiguiente() != null
```

La programación del algoritmo queda como se ve en el listado 5.12 en la siguiente página.

Figura 5.7 Agregando al final de la lista



Código 5.12 Agregar un registro al final de la lista (**ListaCurso**)

```
68:     /**
69:      * Agrega el registro al final de la lista.
70:      * @param Estudiante nuevo El registro por agregar
71:      */
72:     public void agregaEstFinal(Estudiante nuevo)  {
73:         Estudiante actual;
74:         if ( lista == null) {// No hay nadie en la lista
75:             lista = nuevo;
76:         }
77:         else {
78:             actual = lista;
79:             while (actual.getSiguiente() != null)  {
80:                 actual = actual.getSiguiente();
81:             }
82:             /* Estamos frente al último registro  */
83:             actual.setSiguiente(nuevo);
84:         }  // del else
85:     }
```

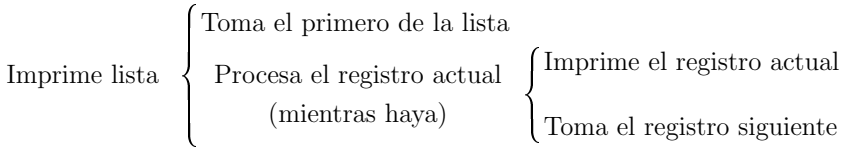
Como se puede ver, seguimos el mismo patrón que recorre la lista, pero nuestra

condición de parada es un poco distinta. Vamos a parar cuando ya no siga nadie al actual, o sea, cuando la referencia al siguiente sea nula.

Sin embargo, si la lista está vacía, nos da un error al tratar de ver el siguiente de alguien que no existe. Adicionalmente, si la lista está vacía, agregar al nuevo registro al principio o al final es equivalente.

Sigamos implementando aquellas funciones que responden al patrón de proceso que dimos. Por ejemplo, el método que lista todos los registros es exactamente este patrón. El algoritmo lo podemos ver en la figura 5.8 y la programación en el listado 5.13.

Figura 5.8 Imprimiendo todos los registros



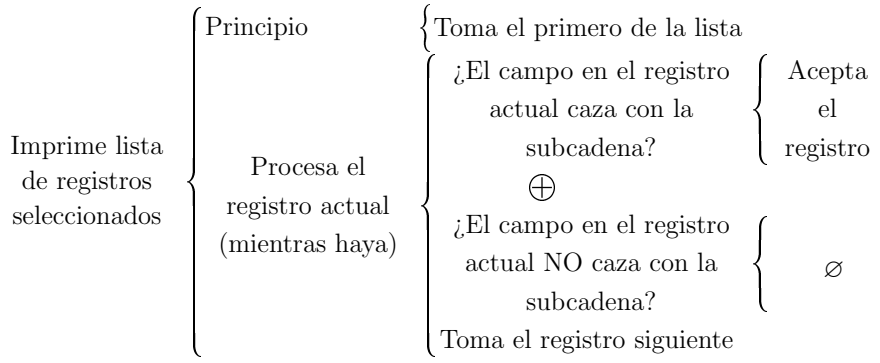
Para programar este método debemos tomar en consideración que debemos escribir en algún dispositivo, por lo que requerimos como parámetro un objeto de la clase Consola. Imprimir el registro es realmente sencillo, porque tenemos un método que nos regresa el registro en forma de cadena.

Código 5.13 Imprimiendo toda la lista (**ListaCurso**)

```
86:   /**
87:    * Lista todos los registros del Curso.
88:    *
89:    * @param Consola cons dónde escribir.
90:    */
91:   public void listaTodos(Consola cons)    {
92:     Estudiante actual = lista;
93:     int i = 0;
94:     while (actual != null) {
95:       i++;
96:       cons.imprimeln(actual.getRegistro());
97:       actual = actual.getSiguiente();
98:     }
99:     if (i == 0)  {
100:      cons.imprimeln("No hay registros en la base de datos");
101:    }
102:  }
```

Para el método que escribe todos los registros que cazan con una cierta subcadena también vamos a usar este patrón, pues queremos revisar a todos los registros y, conforme los vamos revisando, decidir para cada uno de ellos si se elige (imprime) o no. El algoritmo se encuentra en la figura 5.9 y la programación en el listado 5.14.

Figura 5.9 Imprimiendo registros seleccionados



Código 5.14 Imprimiendo registros seleccionados (**ListaCurso**)

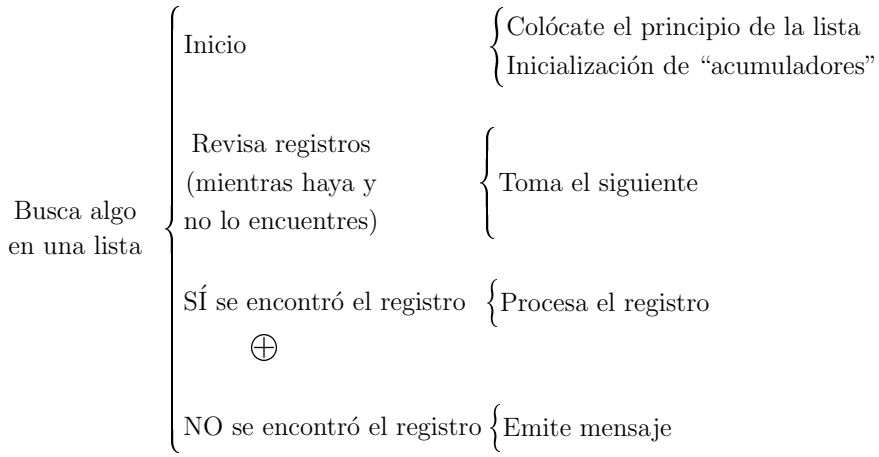
```
103:     /**
104:      * Imprime los registros que  cazan con un cierto patrón.
105:      * @param Consola cons Dispositivo en el que se va a escribir.
106:      * @param int cual  Con cuál campo se desea comparar.
107:      * @param String subcad Con el que queremos que cace.
108:      */
109:     public void losQueCazanCon(Consola cons, int cual,
110:                                           String subcad)      {
111:         int i = 0;
112:         subcad = subcad.toLowerCase();
113:         Estudiante actual = lista;
114:         while (actual != null) {
115:             if (actual.getCampo(cual).indexOf(subcad) != -1) {
116:                 i++;
117:                 cons.imprimeln(actual.getRegistro());
118:             }
119:             actual = actual.getSiguiente();
120:         }
121:         if(i == 0)  {
122:             cons.imprimeln("No se encontró ningún registro "
123:                            + "que  cazara");
124:         }
125:     }
```

En el listado 5.14 en la página opuesta hacemos referencia al método getCampo que implementamos ya en el listado 5.4. En su momento explicamos las características del mismo, por lo que ya no hay necesidad de repetirlas.

Para los métodos buscaEst y quitaEst el patrón que debemos seguir al recorrer la lista es un poco distinto, porque tenemos que parar en el momento en que encontremos lo que estamos buscando, y ya no seguir recorriendo la lista. Pero deberemos prevenir el que lo que buscamos no se encuentre en la lista. El patrón general para este tipo de búsquedas se muestra en la figura 5.10.

Para el caso en que se busca un registro que tenga como subcadena en un campo determinado a cierta subcadena, el proceso del registro que no caza no es ninguno, mientras que el proceso del que caza es simplemente regresar la referencia del registro que cazó. La programación de este método queda como se muestra en el listado 5.15 en la siguiente página.

Figura 5.10 Patrón de búsqueda de un registro que cumpla con ...



Como se puede observar, la condición de parada es doble: por un lado se debe verificar que no se acabe la lista, y por el otro si el registro actual cumple la condición que buscamos, que en este caso es que contenga a la subcadena en el campo indicado.

Código 5.15 Método que busca un registro (**ListaCurso**)

```
126:     /**
127:      * Busca al registro que contenga a la subcadena.
128:      *
129:      * @param int cual Cual es el campo que se va a comparan.
130:      * @param String subcad La cadena que se está buscando.
131:      * @return Estudiante El registro deseado o null.
132:      */
133:     public Estudiante buscaSubcad(int cual, String subcad)    {
134:         Estudiante actual;
135:         subcad = subcad.trim().toLowerCase();
136:         actual = lista;
137:         while (actual != null &&
138:                          (actual.getCampo(cual).
139:                           indexOf(subcad.toLowerCase())) == -1)
140:                 actual = actual.getSiguiente();
141:         return actual;
142:     }
```

5.2.1. Paso por valor y por referencia

Es el momento adecuado para saber qué pasa cuando dentro de un método o función se altera el valor de un parámetro, como sucede en la línea 135: en el listado 5.15. En Java se dice que los parámetros pasan *por valor*. Esto quiere decir que al llamar a un método o función, se hace una *copia* del valor de los parámetros, y esto es lo que se entrega al método. Por lo tanto, todo lo que el método haga con ese parámetro no se reflejará fuera de él, ya que trabaja con una copia. Cuando un parámetro se pasa *por valor*, se dice que es un parámetro de entrada.

A lo mejor esto resulta un poco confuso tratándose de objetos que pasan como parámetros. Cuando se pasa un objeto como parámetro, se pasa la copia de una variable que contiene una referencia. Por lo tanto, las modificaciones que se hagan al objeto referido por la copia de la variable sí se van a reflejar fuera del método, ya que copia o no, contienen una dirección válida del heap. Los cambios a los que nos referimos son aquéllos que se refieren al valor de la referencia, o sea, ponerlos a apuntar a otro lado. Estos últimos cambios, en el punto de llamada del método, no se reflejan. Por ejemplo, en la línea 135: del listado 5.15, los cambios a subcad se van a ver sólo mientras permanezcamos en buscaSubcad. Una vez que salgamos, podremos observar que la cadena que se utilizó como parámetro no sufrió ningún cambio.

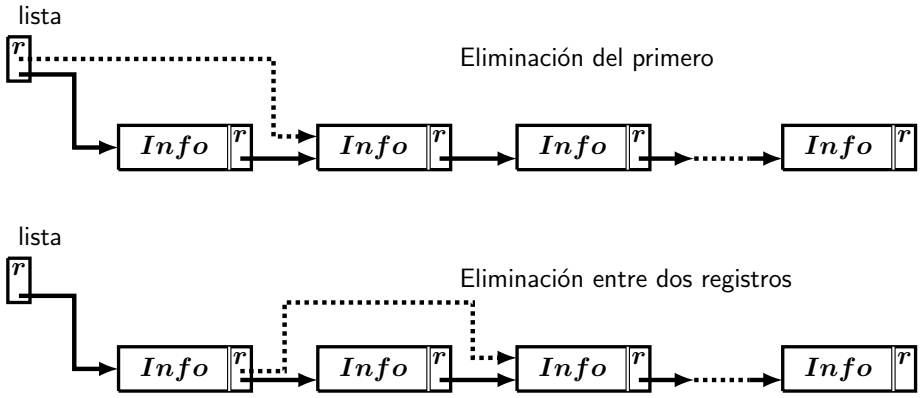
Algunos lenguajes de programación pueden pasar parámetros *por referencia*. Con este método lo que se pasa al método es la dirección (referencia) donde se encuentra la variable en cuestión. Si este método funcionara en Java. lo que estaríamos pasando sería la dirección en memoria donde se encuentra la variable que contiene a la referencia. Java no tiene este método de pasar parámetros, pero se lo pueden encontrar en lenguajes como C++ y Pascal. Cuando un parámetro se pasa *por referencia*, generalmente se hace para dejar ahí algún valor, por lo que se dice que es un parámetro de salida, o de entrada y salida.

En los lenguajes mencionados se usan parámetros por referencia para que los métodos puedan devolver más de un valor. En Java esto se haría construyendo un objeto que tuviera a los campos que queremos regresar, y se regresa al objeto como resultado del método.

5.2.2. Eliminación de registros

Tal vez el proceso más complicado es el de eliminar un estudiante, pues se tienen que verificar varios aspectos de la lista. No es lo mismo eliminar al primer estudiante que eliminar a alguno que se encuentre en posiciones posteriores a la primera. Veamos en la figura 5.11 los dos casos.

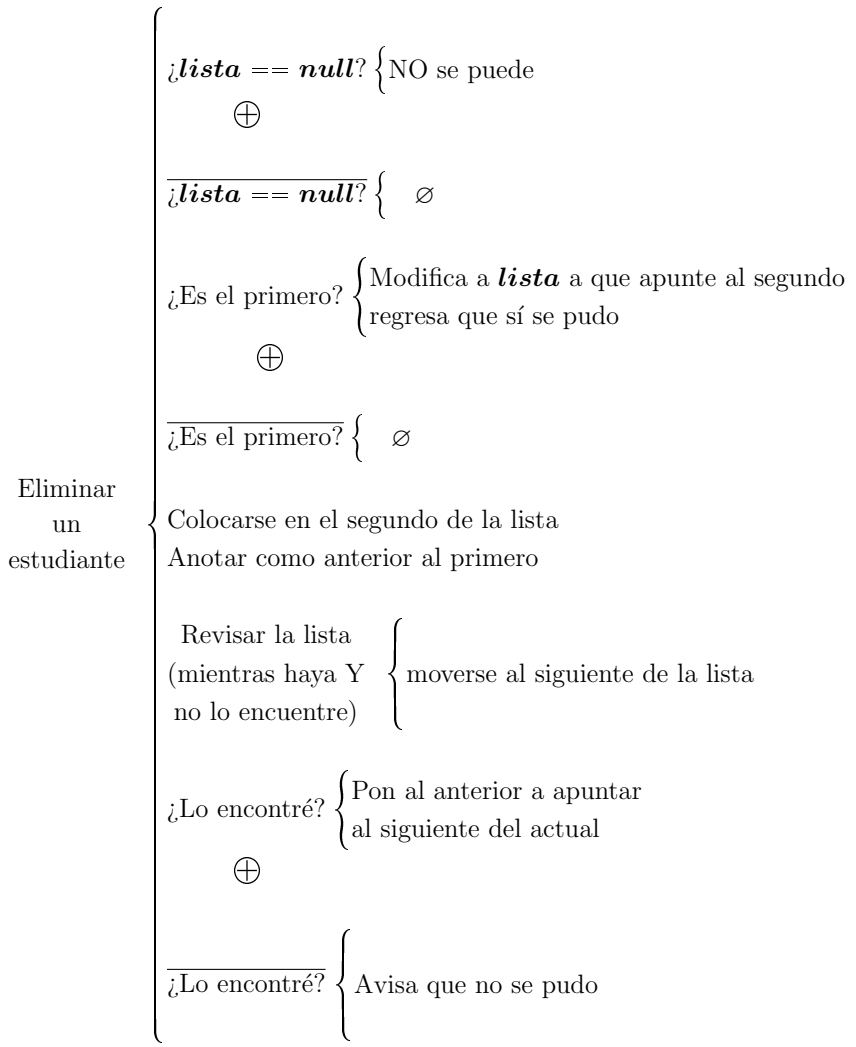
Figura 5.11 Eliminación de un estudiante



En el primer caso tenemos que "redirigir" la referencia lista a que ahora apunte hacia el que era el segundo registro. En el segundo caso tenemos que conservar la

información del registro anterior al que queremos eliminar, para poder modificar su referencia al siguiente a que sea a la que apuntaba el registro a eliminar. El algoritmo para eliminar un registro se encuentra en la figura 5.12.

Figura 5.12 Eliminación de un registro en una lista



Código 5.16 Eliminación de un registro en una lista (**ListaCurso**)

```
143:     /**
144:      * Quita el registro solicitado (por nombre) de la lista
145:      *
146:      * @param String nombre El nombre del registro
147:      */
148:     public boolean quitaEst(String nombre)    {
149:         Estudiante anterior, actual;
150:         nombre=nombre.trim().toLowerCase();
151:         if (lista == null) { // Está vacía la lista: no se pudo
152:             return false;
153:         }
154:         if (lista.getNombre().trim().toLowerCase().equals(nombre)) {
155:             lista = lista.getSiguiente();
156:             return true;
157:         }
158:         // No es igual al primero.
159:         anterior = lista;                // El primero
160:         actual = lista.getSiguiente();  // El segundo
161:         while (actual != null &&
162:                     !(actual.getNombre().toLowerCase().
163:                      equals(nombre.toLowerCase()))) {
164:             anterior = actual;
165:             actual = actual.getSiguiente();
166:         }
167:         if (actual == null) { // No se encontró
168:             return false;
169:         }
170:         anterior.setSiguiente(actual.getSiguiente());
171:         // Procesa
172:         return true;
173:     }
```

Estamos aprovechando que podemos salir de un método a la mitad del mismo para que quede un esquema más simple de la programación, la que se muestra en el listado 5.16.

No hay mucho que aclarar en este método, ya que corresponde directamente al algoritmo. En general, se verifica que el estudiante a eliminar no sea el primero; si lo es, se le elimina; si no lo es, se procede a buscar su ubicación en la lista, manteniendo siempre la referencia al anterior, para poder modificar su referencia al estudiante que se desea eliminar.

5.2.3. La clase MenuLista

La programación del menú para tener acceso a la base de datos del curso es sumamente similar al caso en que los registros eran cadenas, excepto que ahora, para agregar a cualquier estudiante hay que construir un objeto de la clase Estudiante. El algoritmo es el mismo, por lo que ya no lo mostramos. La programación de encuentra en el listado 5.17.

Es en esta clase donde realmente se van a crear objetos nuevos para poderlos ir enlazando en la lista. Por ejemplo, para agregar un estudiante, una vez que tenemos los datos (líneas 95: en la página 164- 98: en la página 164 en el listado 5.17), procedemos a invocar al método agrega de la clase ListaCurso. Este método tiene como argumento un objeto, que es creado en el momento de invocar a agrega – ver línea 99: en la página 164 del listado 5.17. Este es el único método en el que se crean objetos, y esto tiene sentido, ya que se requiere crear objetos sólo cuando se desea agregar a un estudiante.

Para el caso de los métodos de la clase ListaCurso que regresan una referencia a un objeto de tipo estudiante, es para lo que se declaró la variable donde – línea 76: en la página 164 del listado 5.17. En estos casos el objeto ya existe, y lo único que hay que pasar o recibir son las variables que contienen la referencia.

Código 5.17 Menú para el manejo de la lista (**MenuLista**)1/5

```
1: import icc1.interfaz.Consola;
2:
3: class MenuLista  {
4:     /**
5:      * Reporta que el estudiante buscado no fue localizado.
6:      *
7:      * @param cons OutputStream en el que se escribe
8:      * @param nombre subcadena que no fue localizada
9:      */
10:     static final String blancos =
11:                         "                                        ";
12:     private static final int SALIR = 0,
13:                              AGREGAR = 1,
14:                              QUITAR = 2,
15:                              BUSCAR = 3,
16:                              LISTARTODO = 4,
17:                              LISTARALGUNOS = 5;
```

Código 5.17 Menú para el manejo de la lista (**MenuLista**) 2/5

```
18:     private void reportaNo(Consola cons, String nombre) {
19:         cons.imprimeln("El␣estudiante␣:\n\t" +
20:                        nombre +
21:                        "\n␣No␣está␣en␣el␣grupo");
22:     }
23:     /**
24:      * Pide en el InputStream el nombre del alumno.
25:      * @param cons el InputStream
26:      * @return El nombre leído
27:      */
28:     private String pideNombre(Consola cons) {
29:         String nombre = cons.leeString("Dame␣el␣nombre␣del␣" +
30:                     "estudiante,␣empezando␣por␣apellido␣paterno:");
31:         return nombre;
32:     }
33:     /**
34:      * Pide en el InputStream la carrera del alumno.
35:      * @param cons el InputStream
36:      * @return La carrera leída
37:      */
38:     private String pideCarrera(Consola cons) {
39:         String carrera = cons.leeString("Dame␣una␣de␣las␣" +
40:                              "carreras:\t" +
41:                              "Mate\tComp\tFisi\tActu\tBiol:");
42:         return carrera;
43:     }
44:     /** Pide en el InputStream la clave de acceso del alumno.
45:      * @param cons el InputStream
46:      * @return La clave de acceso leída
47:      */
48:     private String pideClave(Consola cons) {
49:         String clave =
50:             cons.leeString("Dame␣la␣clave␣de␣acceso␣del␣" +
51:                                                 "estudiante:");
52:         return clave;
53:     }
54:     /** Pide en el InputStream el numero de cuenta del alumno.
55:      * @param cons el InputStream
56:      * @return El número de cuenta leído
57:      */
58:     private String pideCuenta(Consola cons) {
59:         String cuenta =
60:             cons.leeString("Dame␣el␣número␣de␣cuenta␣del" +
61:                            "estudiante,␣de␣9␣dígitos");
62:         return cuenta;
63:     }
```

Código 5.17 Menú para el manejo de la lista (**MenuLista**)3/5

```
64:     /**
65:      * Maneja el menú para el acceso y manipulación de la base de
66:      * datos.
67:      *
68:      * @param cons Dispositivo en el que se va a interaccionar
69:      * @param miCurso Base de datos con la que se va a trabajar
70:      * @return la opción seleccionada, después de haberla ejecutado
71:      */
72:     public int daMenu(Consola cons, ListaCurso miCurso)  {
73:         String nombre, cuenta, carrera, clave;
74:         String subcad;
75:         String registros;
76:         Estudiante donde;
77:         int cual;
78:         String scual;
79:         String menu = "(0)\t Termina \n " +
80:             "(1)\t Agrega \n " +
81:             "(2)\t Quita \n " +
82:             "(3)\t Busca \n " +
83:             "(4)\t Lista todos \n "
84:             "(5)\t Lista subconjunto que contenga a ....";
85:         cons.imprimeln(menu);
86:         int opcion;
87:         String sopcion = cons.leeString("Elige una opción -->");
88:         opcion = "012345".indexOf(sopcion);
89:         switch(opcion)  {
90:         case SALIR:                  // Salir
91:             cons.imprimeln("Espero haberte servido.\n" +
92:                            "Hasta pronto...");
93:             return -1;
94:         case AGREGAR:                 // Agrega Estudiante
95:             nombre = pideNombre(cons);
96:             cuenta = pideCuenta(cons);
97:             carrera = pideCarrera(cons);
98:             clave = pideClave(cons);
99:             miCurso.agregaEst(new Estudiante(nombre, cuenta,
100:                                              clave, carrera));
101:            return 1;
102:        case QUITAR:                        // Quita estudiante
103:            nombre = pideNombre(cons);
104:            donde = miCurso.buscaSubcad(Estudiante.NOMBRE, nombre);
105:            if ( donde != null) {
106:                nombre = donde.getNombre();
107:                miCurso.quitaEst(nombre);
108:                cons.imprimeln("El estudiante:\n \t" +
109:                               nombre + "\n Ha sido eliminado");
110:            }
```

Código 5.17 Menú para el manejo de la lista (**MenuLista**) 4/5

```
111:             else reportaNo(cons, nombre);
112:             return 2;
113:         case BUSCAR:                          // Busca subcadena
114:             subcad = cons.leeString("Dame la subcadena a "+
115:                                     "buscar: ");
116:             do {
117:                 scual = cons.leeString(
118:                     "Ahora dime de cuál campo : 1:Nombre"
119:                     + "2:Cuenta, 3:Carrera, 4:Clave");
120:                     cual = "01234".indexOf(scual);
121:                     if (cual < 1)  {
122:                         cons.imprimeln("Opción no válida");
123:                     }
124:             } while (cual  < 1);
125:             donde = miCurso.buscaSubcad(cual, subcad);
126:             if (donde != null)  {
127:                 cons.imprimeln(donde.getRegistro());
128:             }
129:             else  {
130:                 reportaNo(cons, subcad);
131:             }
132:             return 3;
133:         case LISTARTODOS:                     // Lista todos
134:              miCurso.listaTodos(cons);
135:              return 4;
136:         case LISTARALGUNOS:                    // Lista con criterio
137:             subcad = cons.leeString("Da la subcadena que " +
138:                                     " quieres contengan los " +
139:                                     "registros:");
140:             do {
141:                 scual = cons.leeString(
142:                     "Ahora dime de cuál campo : 1:Nombre, "
143:                     + "2:Cuenta, 3:Carrera, 4:Clave");
144:                 cual = "01234".indexOf(scual);
145:                 if (cual < 1)  {
146:                     cons.imprimeln("Opción no válida");
147:                 }
148:             } while (cual  < 1);
149:             miCurso.losQueCazanCon(cons, cual, subcad);
150:             return 5;
151:         default:                 // Error, vuelve a pedir
152:             cons.imprimeln("No diste una opción válida.\n " +
153:                            "Por favor vuelve a elegir.");
154:             return 0;
155:         } // switch
156:     } // daMenu
```

Código 5.17 Menú para el manejo de la lista (**MenuLista**) 5/5

```
157:     public static void main(String[] args)  {
158:         int opcion;
159:         Consola consola = new Consola();
160:         ListaCurso miCurso = new ListaCurso("4087");
161:         MenuLista miMenu = new MenuLista();
162:         while ((opcion = miMenu.daMenu(consola, miCurso)) != -1);
163:     }
164: }
```

Con esto damos por terminado este capítulo. Se dejan como ejercicios las siguientes mejoras:

- Conseguir que al agregar un registro, éste se agregue en el lugar que le toca alfabéticamente.
- Si la lista está ordenada, al buscar a un registro, se sabe que no se encontró en cuanto se localiza el primer registro con llave mayor a la llave del que se busca, por lo que se puede hacer la búsqueda más eficiente.

Herencia 6

En los lenguajes orientados a objetos, la *herencia*, que conlleva la capacidad de reutilizar código de manera inteligente, es una de las características más importantes de este enfoque.

6.1 Extensión de clases

Una de las características más valiosas en el código que hacemos es la posibilidad de *reutilizarlo*, esto es, tomar un código que ya trabaja para cierta aplicación y usarlo en otra aplicación parecida pero no idéntica. En el contexto de lenguajes tradicionales esto se logrando copiando aquellas partes del programa que nos sirven y modificándolas para que se ajusten a nuevas situaciones. Estos mecanismos son, en el mejor de los casos muy laboriosos, y en el peor muy susceptibles a errores.

Uno de los mayores beneficios de la Orientación a Objetos es la posibilidad de extender clases ya construidas – construir *subclases* – de tal manera que se pueda seguir utilizando la clase original en el contexto original, pero se pueda, asimismo, agregarle atributos a la subclase, redefinir algunos de los métodos para que tomen en cuenta a los nuevos atributos o agregar métodos nuevos.

La clase original es la *superclase* con respecto a la nueva; decimos que la

subclase *hereda* los atributos y métodos de la superclase. Que herede quiere decir que, por el hecho de extender a la superclase, tiene al menos todos los atributos y métodos de la superclase.

Si regresamos por unos momentos a nuestro ejemplo de la base de datos para listas de cursos, veremos que hay muchas otras listas que se nos ocurre hacer y que tendrían la misma información básica. Quitemos la clave de usuario, porque esa información no la necesitan más que en el centro de cómputo, y entonces nuestra clase EstudianteBasico quedaría definida como se muestra en el listado 6.1. Omitimos los comentarios para tener un código más fluido.

Código 6.1 Superclase **EstudianteBasico** 1/2

```
1: import icc1.interfaz.Consola;
2: /**
3:  * Registro básico para una Base de Datos de estudiantes
4:  * de la Facultad de Ciencias.
5:  */
6: class EstudianteBasico    {
7:     private String nombre, carrera, cuenta;
8:     private EstudianteBasico siguiente;
9:
10:    static public final int NOMBRE = 1;
11:    static public final int CUENTA = 2;
12:    static public final int CARRERA = 3;
13:
14:    public EstudianteBasico()  {
15:        nombre = carrera = cuenta = null;
16:    }
17:    public EstudianteBasico(String nmbre, String cnta,
18:                            String crrera)   {
19:        nombre = nmbre.trim();
20:        cuenta = cnta.trim();
21:        carrera = crrera.trim();
22:        siguiente = null;
23:    }
24:    public String getNombre()     {
25:        return nombre;
26:    }
27:    public void setNombre(String nombre)  {
28:        this.nombre = nombre;
29:    }
30:    public String getCarrera()        {
31:        return carrera;
32:    }
```

Código 6.1 Superclase **EstudianteBasico** 2/2

```
33:     public void setCarrera(String carre) {
34:         carrera = carre;
35:     }
36:     public String getCuenta()        {
37:         return cuenta;
38:     }
39:     public void setCuenta(String cnta)        {
40:         cuenta = cnta;
41:     }
42:     public EstudianteBasico getSiguiente()        {
43:         return siguiente;
44:     }
45:     public void setSiguiente(EstudianteBasico sig)   {
46:         siguiente = sig;
47:     }
48:     public String getRegistro()        {
49:         return nombre.trim()+"\t" +
50:             cuenta.trim()+"\t" +
51:             carrera.trim();
52:     }
53:     public String getCampo(int cual)        {
54:         String cadena;
55:         switch(cual) {
56:         case EstudianteBasico.NOMBRE:
57:             cadena = getNombre().trim().toLowerCase();
58:             break;
59:         case EstudianteBasico.CUENTA:
60:             cadena = getCuenta().trim().toLowerCase();
61:             break;
62:         case EstudianteBasico.CARRERA:
63:             cadena = getCarrera().trim().toLowerCase();
64:             break;
65:         default:
66:             cadena = "Campo no existente";
67:         }
68:         return cadena;
69:     }
70:     public void setRegistro(String nmbre, String cnta,
71:                             String crrera)        {
72:          nombre = nmbre.trim();
73:          cuenta = cnta.trim();
74:          carrera = crrera.trim();
75:     }
76: }
```

Se nos ocurren distintos tipos de listas que tomen como base a EstudianteBasico. Por ejemplo, un listado que le pudiera servir a una biblioteca tendría, además de la información de EstudianteBasico, los libros en préstamo. Un listado para la División de Estudios Profesionales debería tener una historia académica y el primer año de inscripción a la carrera. Un listado para calificar a un grupo debería tener un cierto número de campos para las calificaciones. Por último, un listado para las salas de cómputo debería contar con la clave de acceso.

En todos los casos que listamos, la información original debe seguir estando presente, así como los métodos que tiene la superclase. Debemos indicar, entonces, que la clase que se está definiendo *extiende* a la superclase, *heredando* por lo tanto todos los métodos y atributos de la superclase. La sintaxis en Java se muestra a continuación.

SINTAXIS:

⟨*subclase que hereda*⟩ ::= **class** ⟨*subclase*⟩ **extends** ⟨*superclase*⟩ **{**
⟨*declaraciones de campos y métodos*⟩
}

SEMÁNTICA:

La ⟨*subclase*⟩ es una nueva declaración, que incluye (hereda) a todos los campos de la superclase, junto con todos sus métodos. Las ⟨*declaraciones de campos y métodos*⟩ se refiere a lo que se desea *agregar* a la definición de la superclase en *esta* subclase.

Si alguna de las declaraciones hechas en la ⟨*subclase*⟩ coincide en tipo o firma con algo declarado en la ⟨*superclase*⟩, entonces los campos o métodos de la superclase quedarán "tapados" o escondidos (redefinidos) para la ⟨*subclase*⟩.

Pongamos por ejemplo la programación de una subclase para las listas con calificaciones. El registro deberá contener lo que contiene EstudianteBasico, agregándole nada más lugar para las calificaciones de los exámenes.El encabezado de la clase quedaría como se ve en el listado 6.2.

Código 6.2 Encabezado para la subclase **EstudianteCurso**

```
class EstudianteCurso extends EstudianteBasico    {
```

A partir de ese momento podemos asumir que EstudianteCurso contiene todos los campos y métodos que tiene EstudianteBasico. Los únicos métodos que no here-

da implícitamente son los constructores con parámetros. Si existe un constructor sin parámetros, ya sea el que está por omisión o alguno que haya declarado el programador, éste va a ser invocado **siempre** que se invoque a algún constructor de la subclase EstudianteCurso. Regresaremos a esto en cuanto programemos los constructores para la subclase.

Cuando tenemos extensión de clases tiene sentido hablar del tipo de acceso que hemos mencionado muy poco, el acceso *protegido* (protected). Este tipo de acceso se usa para cuando queremos que las clases que extienden a la actual puedan tener acceso a ciertas variables o métodos, ya que funciona como si fuera acceso público para las clases que extienden, pero como si fuera privado para las clases ajenas a la clase actual. Los datos con acceso privado no son accesibles más que para la clase actual. El acceso protegido funciona como el privado para clases que no heredan, y como público para las clases que heredan.

Cuando un método de una superclase tiene un cierto acceso, si se redefine ese método en una subclase el acceso no puede ser restringido. Esto es, si en la superclase el método tenía acceso público, en la subclase deberá mantener ese mismo acceso. Si en la superclase tenía acceso de paquete, en la subclase puede tener acceso de paquete o público, pero el acceso no puede ser ni protegido ni privado.

6.2 Arreglos

En esta sección estudiaremos estructuras repetitivas de datos, organizados como vectores, matrices, etc.

6.2.1. Arreglos de una dimensión

Podemos pensar en las calificaciones de cada estudiante como un vector,

$$\boldsymbol{calif}_0, \boldsymbol{calif}_1, \ldots, \boldsymbol{calif}_{14}$$

donde estamos asumiendo que cada estudiante tiene lugar para, a lo más, 15 calificaciones (del 0 al 14). Se ve, asimismo, muy útil el poder manejar a cada una de las calificaciones refiriéndonos a su subíndice, en lugar de declarar calif0, calif1, etc., imitando el manejo que damos a los vectores en matemáticas.

Es importante notar que lo que tenemos es una colección de datos del mismo tipo que se distinguen uno de otro por el lugar que ocupan. A estas colecciones les llamamos en programación *arreglos*. La declaración de arreglos tiene la siguiente sintaxis:

SINTAXIS:
⟨*declaración de arreglo*⟩ ::= ⟨*tipo*⟩[ ] ⟨*identificador*⟩;
SEMÁNTICA:
Se declara una variable que va a ser una referencia a un arreglo de objetos o datos primitivos del tipo dado.

Veamos algunos ejemplos:

```
int [ ] arregloDeEnteros;            // Arreglo de enteros
EstudianteBasico [ ] estudiantes ;   // Arreglo de objetos del tipo
                                     //          EstudianteBasico.
float [ ] vector ;                   // Arreglo de reales.
String [ ] cadenas;                  // Arreglo de cadenas
```

Noten como en la declaración no estamos diciendo el tamaño del arreglo. Esto es porque en este momento Java no tiene por qué saber el número de elementos, ya que lo único que está reservando es una variable donde guardar la referencia de donde quedará, en el heap, el arreglo.

Podemos en la declaración de los arreglos determinar el tamaño que van a tener. Lo hacemos como en las cadenas, inicializando el arreglo en el momento de declararlo, proporcionando una lista de los elementos que queremos en el arreglo. En ese caso, el tamaño del arreglo quedará fijado al número de elementos que se dé en la lista. La sintaxis para este tipo de declaración es:

SINTAXIS:
⟨*declaración de arreglos con inicialización*⟩ ::=
⟨*tipo*⟩[ ] ⟨*identificador*⟩ = { ⟨*lista*⟩ };
SEMÁNTICA:
Para inicializar el arreglo se dan, entre llaves, valores del tipo del arreglo, separados entre sí por coma. El arreglo tendrá tantos elementos como aparezcan en la lista, y cada elemento estará creado de acuerdo a la lista. En el caso de un arreglo de objetos, la lista deberá contener objetos que ya existen, o la creación de nuevos mediante el operador new.

Extendamos los ejemplos que acabamos de dar a que se inicialicen en el momento de la declaración:

- Arreglo de enteros con 5 elementos, todos ellos enteros. Corresponden a los primeros 5 números primos.

```
int[] primos = {2,3,5,7,11};
```

- Arreglo de reales con tres elementos.

```
float[] vector = { 3.14, 8.7, 19.0 };
```

- Arreglo de dos cadenas, la primera de ellas con el valor "Sí" y la segunda con el valor "No".

```
String[] cadenas = { "Sí", "No"};
```

- Arreglo de objetos del tipo EstudianteBasico con tres elementos, el primero y tercero con el constructor por omisión y el segundo con los datos que se indican.

```
EstudianteBasico paco =
                new Estudiante("Paco",
                               "095376383",
                               "Compu");

EstudianteBasico[] estudiantes =
                              {new Estudiante(),
                               paco,
                               new Estudiante()
                              };
```

Veamos en las figuras 6.1 a 6.3 a continuación cómo se crea el espacio para los arreglos que se mostraron arriba. En los esquemas marcamos las localidades de memoria que contienen una referencia con una "@"en la esquina superior izquierda. Esto quiere decir que su contenido es una dirección en el heap. Las localidades están identificadas con rectángulos. El número que se encuentra ya sea inmediatamente a la izquierda o encima del rectángulo corresponde a la dirección en el heap.

Figura 6.1 **int**[ ] primos = {2,3,5,7,11};

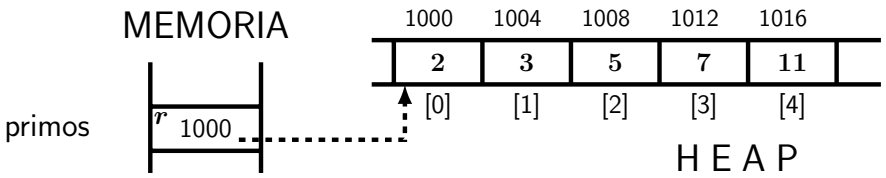


Figura 6.2 **float [ ]** vector = { 3.14, 8.7, 19.0};

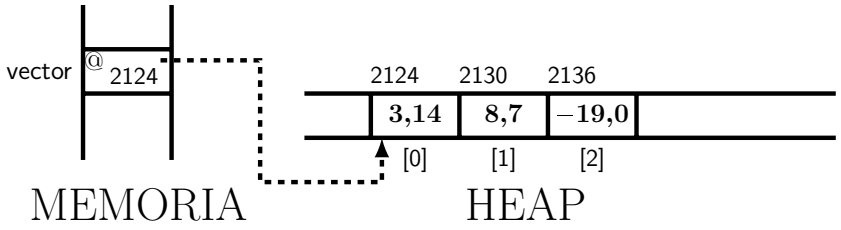


Figura 6.3 **String [ ]** cadenas = { "Sí", "No"};

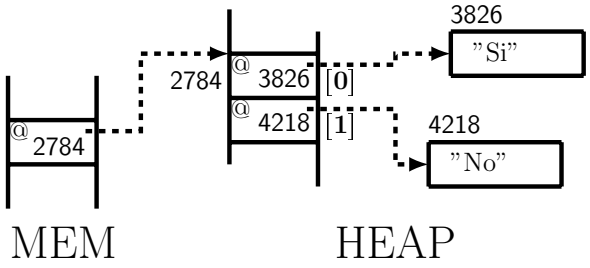
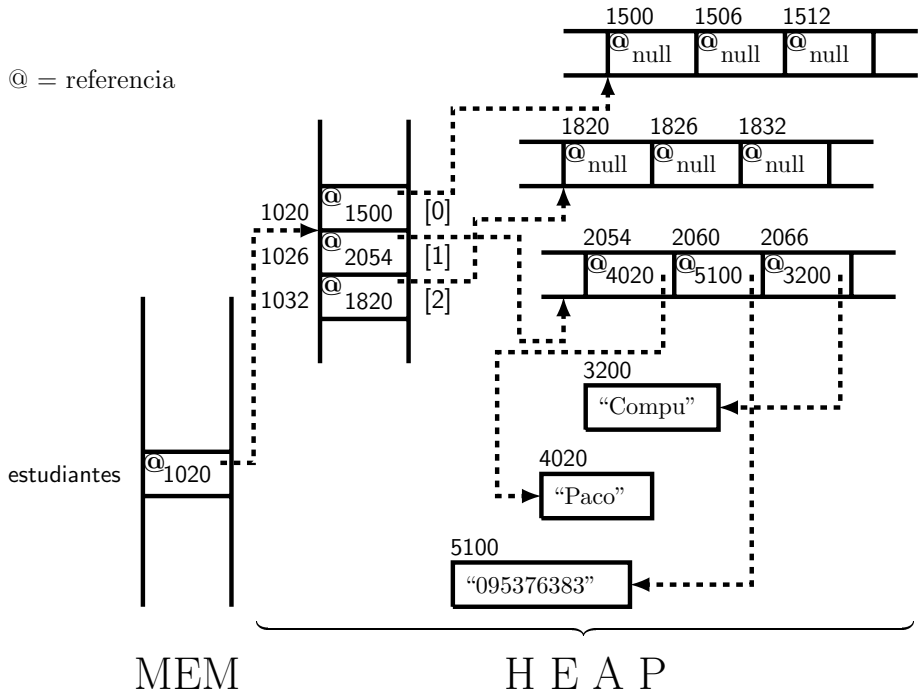


Figura 6.4 Declaración del contenido de un arreglo de objetos




```
EstudianteBasico[ ] estudiantes = {%
                new Estudiante(),
                new Estudiante("Paco", "095376383", "Compu"),
                                new Estudiante()   };
```

La manera "normal" de darle tamaño a un arreglo es creándolo, con el enunciado new y especificando el número de elementos que va a tener el arreglo. En este caso las referencias a objetos se inicializan en null mientras que los datos numéricos se inicializan en 0 (cero). La sintaxis precisa se da a continuación.

SINTAXIS:

⟨*declaración de un arreglo con tamaño dado*⟩ ::=

⟨*tipo*⟩ ⟨*identificador*⟩ = **new** ⟨*tipo*⟩[⟨*expresión entera*⟩];

SEMÁNTICA:

Se inicializa la referencia a un arreglo de referencias en el caso de objetos, o de datos en el caso de tipos primitivos.

Si los ejemplos que dimos antes los hubiéramos hecho sin la inicialización serían:

```
1:  int[] primos = new int[5];
2:  EstudianteBasico[] estudiantes = new EstudianteBasico[3];
3:  float[] vector = new float[3];
4:  String[] cadenas = new String[2];
```

y los esquemas de su organización en memoria quedaría como se muestra en las figuras 6.5 a 6.8.

La creación de los arreglos – en la modalidad que estamos presentando – no tiene por qué darse en la declaración, sino que, como con cualquier otro dato, ya sea éste primitivo u objeto, se puede hacer como un enunciado de asignación común y corriente. Lo único que hay que tener en mente es que a una misma referencia se le pueden asignar distintos espacios en el heap.

Figura 6.5 **int**[ ] primos = **new int**[5];

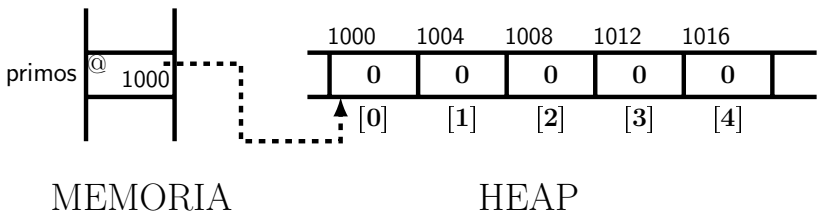


Figura 6.6 EstudianteBasico[ ] estudiantes = **new** EstudianteBasico[3];

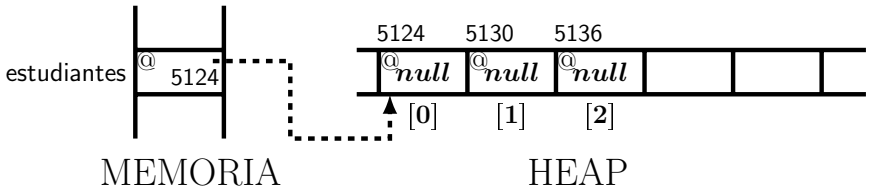


Figura 6.7 **float**[ ] vector = **float**[3];

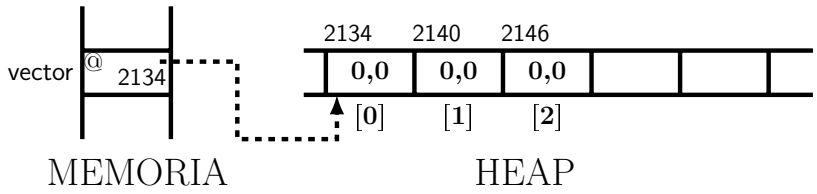
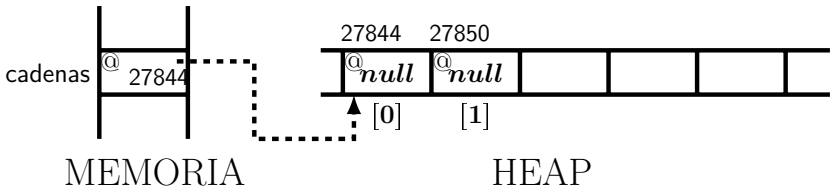


Figura 6.8 String[ ] cadenas = **new** String[2];



Uso de los elementos de un arreglo

En el caso de los arreglos de una dimensión, para usar a un elemento del arreglo debemos seleccionarlo, de manera similar a como seleccionamos a un atributo o método de un objeto. Mientras que en el caso de los objetos se utiliza el operador punto (.), en el caso de los arreglos de una dimensión se usa el operador [ ], cuya sintaxis es:

SINTAXIS:

⟨*selección de un elemento de un arreglo*⟩ ::=
⟨*id. de arreglo*⟩ [⟨*expresión entera*⟩]

SEMÁNTICA:

El operador [ ] es el de mayor precedencia de entre los operadores de Java. Eso indica que evaluará la expresión dentro de ella antes que cualquier otra operación (en la ausencia de paréntesis). Una vez obtenido el entero correspondiente a la ⟨*expresión entera*⟩ – que pudiera ser una constante entera – procederá a elegir al elemento con ese índice en el arreglo. El resultado de esta operación es del tipo de los elementos del arreglo. De no existir el elemento al que corresponde el índice calculado, el programa abortará con el mensaje *ArrayIndexOutOfBoundsException*. Al primer elemento del arreglo le corresponde **siempre** el índice **0** (cero) y al último elemento el índice $\boldsymbol{n-1}$, donde el arreglo se creó con $\boldsymbol{n}$ elementos.

Es importante apreciar que el tamaño de un arreglo no forma parte del tipo. Lo que forma parte del tipo es el número de dimensiones (vector, matriz, cubo, ...) y el tipo de sus elementos.

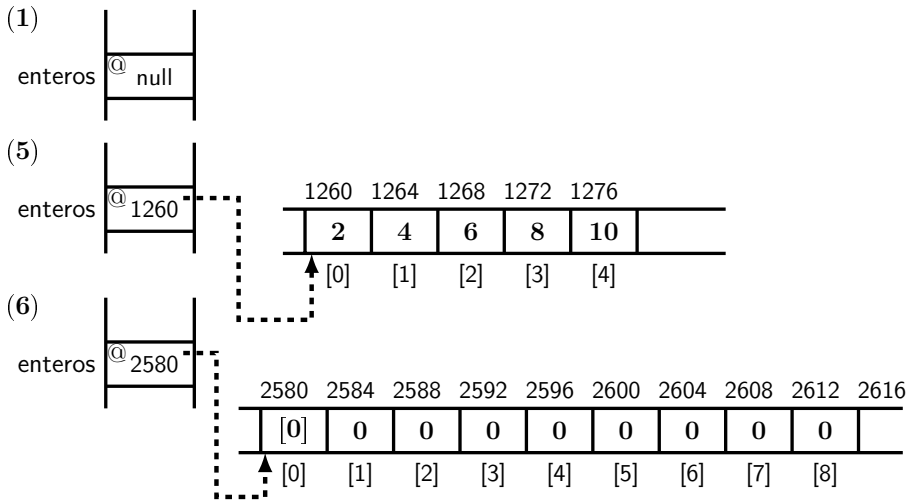
Los arreglos son estructuras de datos estáticas

Cuando decimos que un arreglo no puede cambiar de tamaño una vez creado, nos estamos refiriendo al espacio asignado en el heap. Sin embargo, es perfectamente válido que una misma referencia apunte a un arreglo de un cierto tamaño, para pasar después a apuntar a uno de otro tamaño. Supongamos, por ejemplo, que tenemos la siguiente secuencia de enunciados:

```
1:  int[] enteros;
2:  ...
3:  enteros = new int[5];
4:  int i = 0;
5:  while (i < 5)  {
6:      enteros[i] = (i + 1) * 2;
7:      i++;
8:  }
9:  enteros = new int[9];
```

La secuencia de pasos que se dan durante la ejecución se puede observar en la figura 6.9 en la página opuesta. El número entre paréntesis a la izquierda de cada subesquema corresponde al número de instrucción que se terminó de ejecutar.

Figura 6.9 Reasignación de arreglos



Como se puede observar en esta figura, el arreglo que se crea en la línea 3 del código no es el mismo que el que se crea en la línea 7: no se encuentran en la misma dirección del heap, y no contienen lo mismo. Insistimos: no es que haya cambiado el tamaño del arreglo, sino que se creó un arreglo nuevo.

6.2.2. Iteración enumerativa

En el código que acabamos de presentar utilizamos una iteración que no habíamos visto (pero que se entiende qué hace) y que es una de las formas de iteración más común en los lenguajes de programación. Como el título de esta sección lo indica, se utiliza para hacer un recorrido asociado a un tipo discreto y que se pueda numerar (podríamos utilizar también carácteres). La sintaxis general de este enunciado se da en el cuadro que sigue.

SINTAXIS:

⟨*enunciado de iteración enumerativa*⟩ ::=

for (⟨*enunciado de inicialización*⟩ **;**
⟨*expresión booleana*⟩ **;**
⟨*lista de enunciados*⟩ **)**
⟨*enunciado simple o compuesto*⟩

SEMÁNTICA:

En la ejecución va a suceder lo siguiente:

1. Se ejecuta el ⟨*enunciado de inicialización*⟩.
2. Se evalúa la ⟨*expresión booleana*⟩.
   a) Si es verdadera, se continúa en el paso 3.
   b) Si es falsa, se sale de la iteración.
3. Se ejecuta el ⟨*enunciado simple o compuesto*⟩.
4. Se ejecuta la ⟨*lista de enunciados*⟩.
5. Se regresa al paso 2.

Cualquiera de las tres partes puede estar vacía, aunque el **;** sí tiene que aparecer. En el caso de la primera y tercera parte, el que esté vacío indica que no se hace nada ni al inicio del enunciado ni al final de cada iteración. En el caso de una ⟨*expresión booleana*⟩, se interpreta como la constante true.

Vimos ya un ejemplo sencillo del uso de un while para recorrer un arreglo unidimensional. Sin embargo, para este tipo de tareas el for es el enunciado indicado. La misma iteración quedaría de la siguiente forma:

```
1:  int[] enteros;
2:  ...
3:  enteros = new int[5];
4:  for (int i = 0; i < 5; i++)  {
5:      enteros[i] = (i + 1) * 2;
6:  }
```

Hay una pequeña diferencia entre las dos versiones. En el caso del for, la variable i es local a él, mientras que en while tuvimos que declararla fuera. En ambos casos, sin embargo, esto se hace una única vez, que es el paso de inicialización.

Otra manera de hacer esto con un for, usando a dos variables enumerativas, una para i y otra para i + 1, pudiera ser como sigue:

```
1:  int[] enteros;
2:  ...
3:  enteros = new int[5];
4:  for(int i = 0, j = 1; i < 5; i++,j++)  {
5:      enteros[i] = j * 2;
6:  }
```

Del ejemplo anterior hay que notar que tanto i como j son variables locales al for; para que esto suceda se requiere que la primera variable en una lista de este estilo aparezca declarada con su tipo, aunque la segunda (tercera, etc.) variable **no** debe aparecer como declaración. Si alguna de las variables está declarada antes y fuera del for aparecerá el mensaje de que se está repitiendo la declaración, aunque esto no es correcto. Lo que sí es válido es tener varios for's, cada uno con una variable local i.

Por supuesto que el ⟨*enunciado simple o compuesto*⟩ puede contener o consistir de, a su vez, algún otro enunciado for o while o lo que queramos. La ejecución va a seguir el patrón dado arriba, terminando la ejecución de los ciclos de adentro hacia afuera.

EJEMPLO 6.2.7

Supongamos que queremos calcular el factorial de un entero n que nos pasan como parámetro. El método podría estar codificado como se muestra en el código 6.3.

Código 6.3 Cálculo de $n!$

```
public long factorial(int n)  {
  long fact = 1;
  for (int i = 2; i <= n; i++)  {
     fact = fact * i;
  }
  return fact;
}
```

EJEMPLO 6.2.8

Supongamos ahora que queremos tener dos variables para controlar la iteración, donde la primera nos va a decir en cuál iteración va (se incrementa de 1 en

1) y la segunda se calcula sumándose uno al doble de lo que lo que llevaba. El encabezado del for sería:

```
for (int i=1, j=1; i <= 8 && j < 200; i++, j = 2 * j + 1)  {
   cons.imprimeln("i=\t" + i + "\tj=\t" + j);
}
```

En este ejemplo tenemos dos inicializaciones simultáneas y dos enunciados de iteración que se ejecutan al final del bloque, antes de regresar a volver a iterar. Lo que escribiría este segmento de programa es lo siguiente:

```
i=  1    j=  1
i=  2    j=  3
i=  3    j=  7
i=  4    j=  15
i=  5    j=  31
i=  6    j=  63
i=  7    j=  127
```

Cuando $\boldsymbol{i}$ vale **8**, se deja de cumplir la expresión booleana $\boldsymbol{i < 8 \&\& j < 200}$. Al terminar la iteración donde escribe $\boldsymbol{\ldots j = 127}$, inmediatamente después, antes de regresar al encabezado, calcula $\boldsymbol{j = j * 2 + 1 = 255}$, que ya no cumple la expresión booleana, por lo que ya no vuelve a entrar para $\boldsymbol{i = 8}$.

Es conveniente mencionar que en el encabezado que acabamos de ver se declaran dos variables, $\boldsymbol{i}$ y $\boldsymbol{j}$ que sólo van a ser reconocidas dentro del cuerpo del for. Regresaremos más adelante a utilizar este enunciado. Sin embargo, recomendamos que cuando la expresión booleana no sea numérica o no tenga que ver con relaciones entre enteros, mejor se use alguna de las otras iteraciones que presentamos.

Si estuviéramos en un lenguaje de programación que únicamente tuviera la iteración while, el código sería como se muestra en el listado 6.4.

Código 6.4 Codificación de iteración **for** con **while**

```
int i = 1,
    j = 1;       /*  Inicialización  */
while (i <= 8 && j < 200)  { /* expresión booleana */
   cons.imprimeln("i=\t" + i  "\tj␣=␣" + j);  /* cuerpo  */
   i++;          /* enunciados de iteración */
   j j * 2 + 1;  /* enunciados de iteración */
} //  while
```

EJEMPLO 6.2.9

Podemos tener iteraciones anidadas, como lo dice la descripción de la sintaxis. Por ejemplo, queremos producir un triángulo con la siguiente forma:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

En este caso debemos recorrer renglón por renglón, y en cada renglón recorrer tantas columnas como renglones llevamos en ese momento. El código para conseguir escribir esto se encuentra en el listado 6.5.

Código 6.5 Construcción de un triángulo de números

```
for (int i = 1; i <= 9; i++)  { /* recorre los renglones */
  for (j = 1; j <= i; j++)    { /* recorre las columnas   */
    cons.imprime(j + "\t");     /* escribe el valor de la */
                                /* columna, sin terminar  */
                                /* el renglón.            */
  }
  cons.imprimeln();  /* cambia de renglón */
}
```

Conforme resolvamos más problemas utilizaremos todo tipo de iteraciones y aprenderemos a elegir la más apropiada para cada caso.

6.2.3. Arreglos de más de una dimensión

Es obvio que podemos necesitar arreglos de más de una dimensión, como por ejemplo matrices. Para Java los arreglos de dos dimensiones son arreglos de arre-

glos, y los de tres dimensiones son arreglos de arreglos de arreglos, y así sucesivamente. La declaración se hace poniendo tantas parejas de corchetes como dimensiones queramos en un arreglo, como se puede ver a continuación.

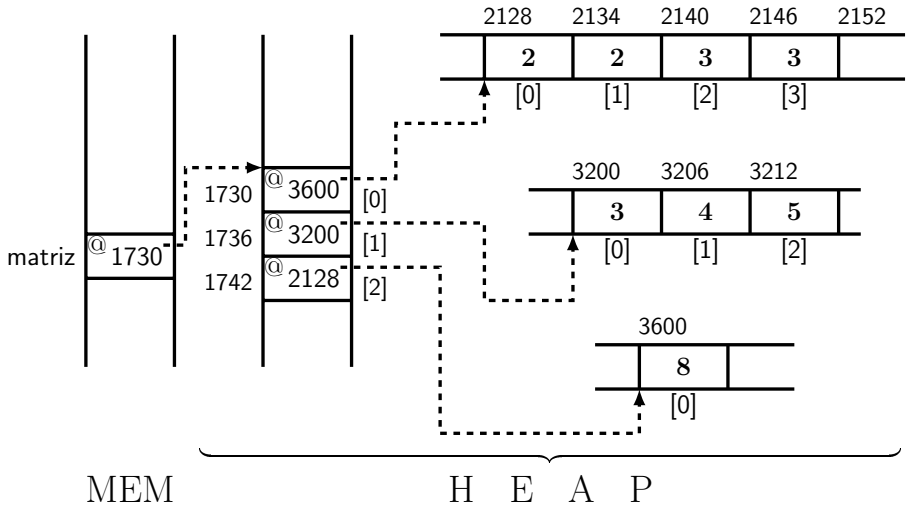
```
int[][] matriz;                          // 2 dimensiones: matriz
EstudianteBasico[][][] facultad;// 3 dimensiones: cubo
```

Por la manera en que maneja Java los arreglos, si quiero inicializar un arreglo, por ejemplo, de dos dimensiones, tendré que darle entre llaves las inicializaciones para cada uno de los renglones:

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

Reconocemos inmediatamente tres sublistas encerradas entre llaves, lo que indica que el arreglo matriz tiene tres renglones. Cada uno de los renglones tiene tamaño distinto, y esto se vale, pues siendo un arreglo de arreglos, cada uno de los arreglos en la última dimensión puede tener el número de elementos que se desee. Un esquema de cómo quedarían en memoria se puede ver en la figura 6.10.

Figura 6.10 Acomodo en memoria de un arreglo de dos dimensiones



Podemos, para un arreglo de 3 dimensiones, tener la siguiente sucesión de enunciados:

```
int[][][] cubos;
cubos = new int[3][][];
cubos[0] = new int[6][];
cubos[1] = new int[3][3];
cubos[2] = new int[2][];
```

y tendríamos un arreglo de 3 dimensiones; en la primera tenemos tres elementos: el primer elemento es un arreglo de 6 arreglos, aunque no sabemos todavía de qué tamaño es cada uno de los 6 arreglos; el segundo elemento es un arreglo de tres arreglos de tres elementos; el tercer elemento es un arreglo de dos arreglos y no sabemos cuantos elementos en cada arreglo.

Como podemos tener también variables o expresiones enteras que nos den el tamaño de un arreglo, es importante que en cualquier momento podamos saber el tamaño de alguna de las dimensiones del mismo. Para eso, la clase que corresponde a los arreglos tiene una variable `length` que nos regresa el tamaño del arreglo. Por ejemplo, para los enunciados que acabamos de hacer, `cubos[0].length` es 6, `cubos[1][1].length` es 3 y `cubos[2].length` es 2.

Resumiendo, los arreglos en Java tienen las siguientes características:

(a) Un arreglo es un objeto. Esto quiere decir que cuando declaramos un arreglo simplemente estamos reservando espacio en memoria para la referencia, la dirección en el heap donde se encontrarán los elementos del arreglo.
(b) Es una estructura de datos *estática*: una vez dado su tamaño, el arreglo no puede achicarse o agrandarse.
(c) Podemos tener arreglos del número de dimensiones y del tipo que queramos.
(d) Los arreglos son estructuras de datos *homogéneas*, porque todos sus elementos son del mismo tipo.
(e) Los arreglos de dos dimensiones no tienen por qué tener el mismo número de elementos en cada renglón del arreglo.Esto se extiende en todas las dimensiones que tenga el arreglo.

Uso de arreglos de dos o más dimensiones

De manera similar a como lo hacemos en arreglos de una dimensión, podemos seleccionar a un elemento de un arreglo de más de una dimensión dando una expresión entera para cada una de las dimensiones. Los elementos se van eligiendo de izquierda a derecha, en el mismo orden en que fueron creados. No hay que perder de vista que en el caso de Java una matriz es un arreglo de arreglos, y por la manera en que los representa internamente se puede dar un error de índice en cualquiera de las dimensiones, por lo que es importante mantener presente la construcción del arreglo cuando se seleccionan elementos dentro de él. Las primeras $\boldsymbol{k-1}$ dimensiones, si se selecciona a un elemento de ellos, obtendremos

una referencia a arreglos; es únicamente en la última dimensión (la del extremo derecho) donde obtendremos ya un elemento del tipo dado para el arreglo.

6.2.4. Los arreglos como parámetros o valor de regreso de un método

En algunos otros lenguajes el paso de arreglos como parámetros es realmente complicado. Afortunadamente, por el manejo de arreglos que tiene Java (de guardar las referencias nada más) este aspecto es muy sencillo en este lenguaje.

Para pasar un arreglo como parámetro, únicamente hay que indicarle a Java el número de dimensiones que deberá tener el argumento, y el tipo de los elementos. Esto es necesario porque tiene que saber cuántos brincos tiene que dar, a través de las listas de referencias, para encontrar a los elementos solicitados. Por ejemplo, en el Listado 6.6 en la página opuesta, en la línea 7: se indica que se van a pasar dos arreglos de enteros como argumentos, cada uno de ellos de dos dimensiones; mientras que en la línea 22: (del mismo listado) se indica que los argumentos serán dos arreglos de enteros, cada uno de una dimensión. **No se puede** especificar el tamaño de una dimensión en un parámetro, pues el *tipo* del parámetro tiene que ver nada más con el tipo de los elementos y el número de dimensiones; lo que hay que especificar para cada parámetro de un método es, únicamente, el tipo del parámetro. Lo mismo sucede con el valor que regresa un método: únicamente hay que especificar el tipo del valor que regresa el método, que en el caso de los arreglos, repetimos una vez más, consiste del tipo de los elementos y el número de dimensiones. El método `suma` en la línea 7: regresa un arreglo de enteros de dos dimensiones, mientras que el método con el mismo nombre `suma` de la línea 22: regresa un arreglo de enteros de una dimensión.

Cuando se pasa un arreglo como argumento (en el momento de invocar al método) tenemos que pasar el nombre de lo que espera el método. Por ejemplo, en el Listado 6.6 en la página opuesta, podemos ver las invocaciones a los métodos que se llaman `suma`. La primer invocación es en la línea 41: y se le pasan como argumentos los identificadores `uno` y `dos`, que están declarados como arreglos de enteros de dos dimensiones, que coincide con el tipo de los parámetros.

La otra invocación está en la línea 12:, dentro del método cuya firma es `suma(int[][],int[][])`. En este caso le estamos pasando al método dos arreglos de enteros, cada uno de una dimensión. Como Java maneja los arreglos a través de referencias y vectores de referencias (como ya vimos), se le pasa simplemente una de las referencias (de la primera dimensión) a uno de los arreglos de una dimensión. Cuando en Java declaramos una matriz, en realidad estamos construyendo un arreglo en el que cada elemento es un arreglo; cuando declaramos un cubo, es-

tamos declarando un arreglo en el que cada elemento es un arreglo en el que cada elemento es un arreglo en el que cada elemento es un entero; y así sucesivamente.

Código 6.6 Arreglos como parámetros y valor de regreso de una función 1/2

```
1: public class Arreglos {
2:     /**
3:       * Suma dos arreglos de dos dimensiones.
4:       * @param Los arreglos
5:       * @return el arreglo que contiene a la suma
6:       */
7:     public int[][] suma(int[][] A, int[][] B) {
8:         int min1 = Math.min(A.length,B.length);
9:         int[][] laSuma = new int[min1][];
10:        /* Invocamos a quien sabe sumar arreglos de una dimensión */
11:        for (int i=0; i<min1; i++) {
12:            laSuma[i] = suma(A[i], B[i]);
13:        } // end of for ((int i=0; i<min1; i++)
14:
15:        return laSuma;
16:    }// fin suma(int[][], int[][])
17:    /**
18:     * Suma dos arreglos de una dimensión
19:     * @param Dos arreglos de una dimensión
20:     * @return Un arreglo con la suma
21:     */
22:    public int[] suma(int[] A, int[] B) {
23:        int tam = Math.min(A.length, B.length);
24:
25:        int[] result = new int[tam];
26:        for (int i=0; i<tam; i++) {
27:            result[i] = A[i] + B[i];
28:        }
29:        return result;
30:    }// fin suma(int[], int[])
31:    public static void main(String[] args) {
32:        Consola cons = new Consola();
33:        /* Los dos arreglos a sumar */
34:        int[][] uno = {{3,4,5}, {2,8,7,5},{3,4,7}};
35:        int[][] dos = {{8,4,2,1},{8,7,2,3},{4,3,5,1}};
36:        /* Arreglo para guardar la suma */
37:        int[][] miSuma;
```

Código 6.6 Arreglos como parámetros y valor de regreso de una función 2/2

```
38:         /* Objeto para poder usar los métodos */
39:         Arreglos pruebita = new Arreglos();
40:         /* Invocación a la suma de dos matrices */
41:         miSuma = pruebita.suma(uno, dos);
42:         /* Impresión de los resultados */
43:         for (int i=0; i<miSuma.length; i++) {
44:             for (int j=0; j<miSuma[i].length; j++) {
45:                 cons.imprime(miSuma[i][j]+"\t");
46:             } // end of for (int j=0; j<miSuma[i].length; j++)
47:             cons.imprimeln();
48:         } // end of for (int i=0; i<miSuma.length; i++)
49:     } // fin main()
50: }
```

El resultado de la invocación de esta clase se puede ver en la Figura 6.11.

Figura 6.11 Ejecución de la clase **Arreglos**

```
11    8    7
10    15   9  8
7     7    12
```

En el listado anterior se pregunta por el tamaño de cada arreglo que se va a manejar para no abortar el programa por índices inválidos (aborta dando el mensaje ArrayindexOutOfBoundsException). Queda pendiente el uso de los arreglos. ¿Cuándo hay que construirlos y cuándo no? Por ejemplo, en la línea 9: se construye la primera dimensión, pero la segunda no, y en la línea 25: también se construye el arreglo. Esto se debe a que en el primer caso el i-ésimo renglón aparece del lado izquierdo de una asignación (línea 12:) y lo mismo sucede con cada uno de los elementos en la línea 27: para los elementos del arreglo result. Sin embargo, en la línea 12: no se dice cuántos elementos va a tener la segunda dimensión, porque cuando suma regrese la referencia a un arreglo de una dimensión, esta referencia se copiará en el lugar correspondiente (el arreglo de referencias de la primera dimensión). Lo mismo sucede en la línea 41:, donde al regresar el método a un arreglo de dos dimensiones, como lo que está regresando es una referencia, ésta simplemente se copia.

Con esto damos por terminado el tema de arreglos, que tienen un manejo muy uniforme y flexible en Java.

6.3 Aspectos principales de la herencia

Regresamos a hablar más sobre la herencia en lenguajes orientados a objetos, que es una de las características más importantes de esta clase de lenguajes de programación.

6.3.1. Super y subclases

Tenemos ya las herramientas necesarias para hacer las declaraciones de nuestras subclase, así que proseguiremos donde nos quedamos en la sección 6.1 en la página 167, que era en la declaración de un arreglo para guardar las calificaciones del estudiante. También desarrollaremos el constructor de la subclase – ver listado 6.7. Cuando se crea un objeto cuyo tipo es una subclase, lo primero que hace el objeto es invocar al constructor sin parámetros de la superclase. Si no hay tal – porque hayamos declarado constructores con parámetros nada más – el constructor de la subclase deberá llamar explícitamente a algún constructor de la superclase. Esto se hace con el identificador `super` seguido por la lista de argumentos entre paréntesis.

Supongamos, por ejemplo, que el número de calificaciones que deseamos guardar para un estudiante dado depende del curso que esté tomando. Entonces, la creación del arreglo de calificaciones se haría en el constructor, como se puede ver en el listado 6.7.

Código 6.7 Campos y constructor de **EstudianteCurso**

```
1:  class EstudianteCurso extends EstudianteBasico {
2:      float[] califs;
3:      /** Constructor.
4:       * @params String nombre     El nombre del estudiante.
5:       * @params String cuenta     Número de cuenta.
6:       * @params String carrera    La carrera en la que está inscrito.
7:       * @params int numCalifs     Número de calificaciones que vamos a
8:       *                           guardar para el estudiante.
9:       */
10:     public EstudianteCurso(String nombre, String cuenta,
11:                            String carrera, int numCalifs)  {
12:         super(nombre, cuenta, carrera);
13:         califs = new float[numCalifs];
14:         for(int i=0; i < numCalifs; i++)  califs[i] = 0;
15:     }
```

Una vez hecho y entendido el constructor, se agregan toda clase de métodos que manejan a la parte que se extendió en la subclase. La mayoría de los métodos nuevos únicamente manipulan al nuevo campo, por lo que no tienen nada merecedor de atención – ver listado 6.8.

Código 6.8 Métodos nuevos para la subclase **EstudianteCurso**

```
16:     /** Regresa las calificaciones que tiene el estudiante.
17:      * @returns int[] un arreglo con las calificaciones.
18:      */
19:     public  float[] getCalifs()  {
20:         return califs;
21:     }
22:     /** Regresa una determinada calificación .
23:      * @params int i Regresar la i-ésima calificación.
24:      * @returns float califs[i]. */
25:     public float getCalifs(int i)  {
26:         if ((i >= 0) && (i < califs.length))   return califs[i];
27:         else return -1;
28:     }
29:     /** Pone una determinada calificación.
30:      * @params int i El lugar de la calificación.
31:      * @params float x La calificación.
32:      */
33:     public void setCalif(int i, float x)  {
34:       if(i >= 0 && i < califs.length) {
35:         califs[i] = x;
36:       }
37:     }
38:     /** Da el promedio del estudiante.
39:      * @returns float El promedio
40:      */
41:     public float getPromedio()  {
42:         float suma = 0;
43:         for(int i = 0; i < califs.length; i++)  {
44:           suma += califs[i];
45:         }
46:         return suma /califs.length;
47:     }
```

El único método interesante es getRegistro, ya que existe un método con la misma firma en la superclase. El efecto de declarar un método en la subclase con la misma firma que el de la superclase es que, de hecho, se esconde el método de la superclase. Cualquier uso que se quiera dar a ese método dentro de la subclase tendrá que ser identificado por el prefijo super seguido de un punto – obsérvese la

línea 62 del listado 6.9.

Código 6.9 Redefinición del método **getRegistro()**

```
48:     /**
49:      * Vacía el registro. Este método utiliza al de la superclase
50:      * para darle parte de la cadena, y después completa con las
51:      *  calificaciones.
52:      * @returns String la cadena que contiene al registro
53:      */
54:     public String getRegistro()  {
55:         String cadena = super.getRegistro();
56:         for(int i = 0; i < califs.length; i++) {
57:             if (i %5 == 0)  cadena += "\n";
58:                                 // Cambia de renglón cada 5
59:             cadena += califs[i] + "\t";
60:         }
61:         return cadena;
62:     }
```

6.4 Polimorfismo

Una de las grandes ventajas que nos ofrece la herencia es el poder manipular subclases a través de las superclases, sin que sepamos concretamente de cuál subclase se trata. Supongamos, para poner un ejemplo, que declaramos otra subclase de EstudianteBasico que se llama EstudianteBiblio y que de manera similar a como lo hicimos con EstudianteCurso extendemos adecuadamente y redefinimos nuevamente el método getRegistro(). Podríamos tener el código que sigue:

```
1:         EstudianteBasico[] estudiantes = {
2:             new EstudianteCurso("Pedro'',...)
3:             new EstudianteBiblio(...)
4:         };
```

En estas líneas declaramos un arreglo con dos elementos del tipo EstudianteBasico, donde el primero contiene a un EstudianteCurso mientras que el segundo contiene a un EstudianteBiblio. Como las subclases contienen todo lo que contiene la superclase, y como lo que guardamos son referencias, podemos guardar en un arreglo de la superclase elementos de las subclases. Es más; si hacemos la siguiente solicitud

```
String cadena = estudiantes[0].getRegistro();
```

se da cuenta de que lo que tiene ahí es una referencia a algo del tipo EstudianteCurso, por lo que utilizará la implementación dada en esa subclase para dar respuesta a esta solicitud. Decimos que gobierna el tipo de la referencia, no el tipo de la declaración. La decisión de a cuál de las implementaciones de getRegistro() debe ser invocada se toma en ejecución, ya que depende de la secuencia de ejecución el tipo de la referencia guardada en la localidad del arreglo. A esta capacidad de los lenguajes orientados a objetos de resolver dinámicamente el significado de un nombre de método es a lo que se llama *polimorfismo*, ya que el mismo nombre (de hecho, la misma firma) puede tomar significados distintos dependiendo del estado del programa.

El operador instanceof

Supongamos que estamos en la sección escolar de la Facultad, donde tienen registros de alumnos de distintos tipos, y que le piden al coordinador que por favor le entregue una lista de todos los registros que tiene para la biblioteca. El coordinador deberá tener un programa que *identifique* de qué clase es cada uno de los objetos que se encuentran en la lista (arreglo, lista ligada, etc.). El operador instanceof hace exactamente esto. Es un operador binario, donde las expresiones que lo usan toman la siguiente forma:

⟨*expresión de objeto*⟩ instanceof ⟨*identificador de clase*⟩

y regresa el valor booleano verdadero si, en efecto, el objeto dado en la expresión de la izquierda es un ejemplar de la clase dada a la derecha; y falso si no. Para un proceso como el que acabamos de describir tendríamos un código como el que sigue, suponiendo que tenemos los registros en una lista, como se puede observar en el listado 6.10.

Código 6.10 Registros en un arreglo

```
EstudianteBasico[] estudiantes;
/* Acá se inicializa el arreglo y se van agregando los
 * estudiantes.
 */
int i;
String reporte;
for ( i = 0, reporte = ""; i < estudiantes.length; i++) {
    if ( estudiantes[i] instanceof EstudianteBiblio ) {
        reporte += estudiantes[i].getRegistro();
    }
}
...
```

Supongamos ahora que queremos una lista de estudiantes con su calificación final del curso, pero sin las calificaciones parciales. El método getRegistro no satisface esto. Tendríamos un código similar al anterior en el listado 6.11.

Código 6.11 Otra versión del método **getRegistro**

```
EstudianteBasico[] estudiantes;
/**
 * Acá se inicializa el arreglo y se van agregando los
 * estudiantes.
 */
int i;
String reporte;
for ( i = 0, reporte = ""; i < estudiantes.length; i++) {
    if ( estudiantes[i] instanceof EstudianteCurso ) {
      /**
       * ¡ Acá hay problemas, pues queremos únicamente el
       * nombre del estudiante y su promedio! El nombre y
       * sus datos, pero no hay manera de acceder al método
       * que corresponde a la superclase. Por lo tanto hay
       * que armarlo a pie:
       */
       reporte += estudiantes[i].getNombre() + "\t"
                + ((EstudianteCurso).estudiantes[i]).
                                              getPromedio();
    }
}
...
```

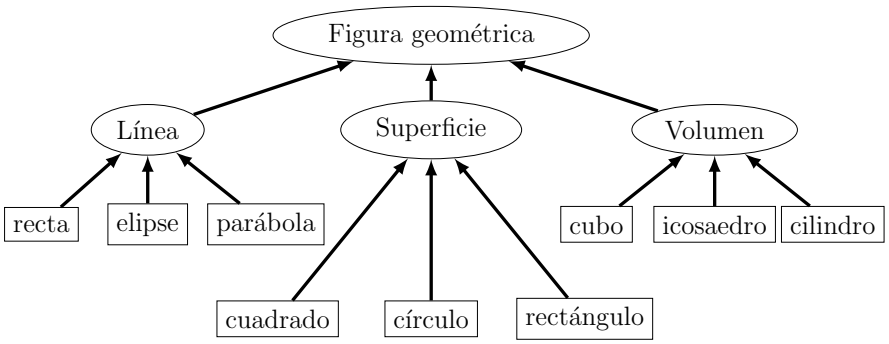
El *casting* que se hizo en la última línea es necesario, porque en el momento de compilación no se sabe la subclase que corresponderá al elemento en cuestión, por lo que en ejecución se deberá pedir que se "interprete" el registro de esa manera. Si no lo puede hacer, porque ése no sea el caso, habrá un error de ejecución de *InvalidClassException*.

En general, cuando tenemos una declaración asociada con una superclase y queremos hacer uso de métodos o atributos declarados para la subclase (desde un programa principal u otra clase que no esté relacionada jerárquicamente con la super y subclase) tendremos que hacer un casting con el nombre de la subclase para que en tiempo de compilación identifique que estamos hablando de la subclase.

6.5 Clases abstractas

Supongamos que tenemos una jerarquía de clases para las figuras geométricas, que se podría ver como en la figura 6.12.

Figura 6.12 Jerarquía de clases



La superclase es Figura Geométrica y reconocemos que toda figura geométrica debe tener los siguientes servicios:

- dibujarse,
- moverse,
- borrarse,
- crearse.

En el caso de las líneas podemos obtener su longitud. En el caso de las superficies, podemos obtener su perímetro y su área. En el caso de los volúmenes, podemos querer obtener su volumen. Por ejemplo, en el caso de las superficies podemos tener una lista de puntos que las definen. Y así sucesivamente.

Cuando estamos en la raíz del árbol realmente todo lo que podemos decir es que se trata de una figura geométrica. Se trata de una *clase abstracta* de la que no podemos decir mucho todavía y de la que no puede haber objetos que sean, en abstracto, una figura geométrica. Conforme vamos bajando por el árbol hacia las hojas se van *concretando* algunas de las características, pero en el caso de la jerarquía que nos ocupa, en el segundo nivel todavía no podemos tener objetos que sean una "superficie" si no decimos qué clase de superficie es. En la figura 6.12 los nodos ovalados representan clases abstractas, mientras que los nodos rectangulares representan clases concretas.

En una jerarquía cualquiera de clases tenemos las mismas características que hemos visto hasta ahora, donde cada subclase hereda todo lo heredable de la superclase, excepto que para uno o más métodos de la clase no damos su implementación. Si una clase tiene al menos un método sin implementación la clase es abstracta y se tiene que indicar eso como se indica a continuación:

SINTAXIS:
⟨*encabezado de clase abstracta*⟩ ::= **abstract class** ⟨*identificador*⟩
SEMÁNTICA:
Le estamos indicando al compilador dos cosas:
- No pueden crearse objetos de esta clase.
- Contiene al menos un método cuya implementación no está definida.

Aquellos métodos que no se puede definir su implementación en el nivel de la jerarquía en la que está la clase, se les precede también de la palabra abstract y se escribe únicamente su encabezado seguido de un ;. En el listado 6.12 podemos ver un ejemplo con la jerarquía de clases que dimos antes. No pondremos las clases completas porque no es el objetivo en este momento.

Código 6.12 Clases abstractas y concretas

```
/* Raíz de la jerarquía. Clase abstracta */
abstract class FiguraGeometrica {
    /* Esquina inferior izquierda */
    int x1, y1;
    abstract public void pinta();
    abstract public void mueve(int x2, int y2);
    abstract public void copia(int x2, int y2);
    ......
}
/* Siguiente nivel de la jerarquía. También abstracta */
abstract class Linea extends FiguraGeometrica {
    abstract public void longitud();
    ......
}
......
/* Tercer nivel de la jerarquía. Clase concreta */
class Recta extends Linea {
    public void pinta() {
        ...
    }
    ...
}
...
```

No hay la obligación de que todos los métodos en una clase abstracta sean abstractos. Dependerá del diseño y de las posibilidades que tenga la superclase para definir algunos métodos para aquellas clases que hereden. Aún cuando la clase no tenga métodos abstractos, si queremos que no se creen objetos de esa clase la declaramos como abstracta.

6.6 Interfaces

La herencia en Java es simple, esto es, cada clase puede heredar de a lo más una superclase. Pero muchas veces necesitamos que hereden de más de una clase. Las *interfaces* corresponden a un tipo, como las clases, que definen exclusivamente comportamiento. Lo único que pueden tener las interfaces son constantes estáticas y métodos abstractos, por lo que definen el contrato que se establece con aquellas clases que implementen esa interfaz. Se dice que una clase *implementa* una interfaz, porque cuando una clase hereda de una interfaz debe dar la implementación de los métodos declarados en la interfaz. La sintaxis para la declaración de una interfaz es:

SINTAXIS:
⟨*encabezado de interfaz*⟩ ::= **interface** ⟨*identificador*⟩
...
SEMÁNTICA:
Se declara un tipo que corresponde a constantes y encabezados de métodos.

Como todo lo que se declare dentro de una interfaz es público, pues corresponde *siempre* a lo que puede hacer un objeto, no se usa el calificativo public en los enunciados dentro de la declaración de la interfaz. Como forzosamente todos los métodos son abstractos, ya que no tienen implementación, tampoco se pone este calificativo frente a cada método. Y como sólo se permiten constantes estáticas, los calificativos de static y final se omiten en las declaraciones de constantes dentro de una interfaz. Por lo tanto, las constantes y métodos en una interfaz serán declarados nada más con el tipo de la constante o el tipo de valor que regrese el método, los identificadores y, en el caso de las constantes el valor; en el caso de los métodos, los parámetros.

Pensemos en el comportamiento de una lista, como las que hemos estado vien-

do. Sabemos que no importa de qué sea la lista, tiene que tener un método que agregue, uno que busque, etc. Dependiendo de los objetos en la lista y de la implementación particular, la implementación de cada uno de los métodos puede variar. Tenemos acá un caso perfecto para una interfaz. En el listado 6.13 podemos ver la declaración de una interfaz de este tipo.

Código 6.13 Interfaz para manejar una lista

```
1:  interfaz Lista {
2:      void agrega(Object obj);
3:      boolean quita(String cad);
4:      Object busca(int cual, String cad);
5:      String armaRegistro();
6:      void listaTodos(Consola cons);
7:      void losQueCazan(int cual, String cad);
8:  }
```

Como se ve en el listado 6.13 ninguno de los métodos tiene su implementación. Cualquier clase que implemente a esta interfaz tiene que dar la implementación de todos y cada uno de los métodos dados en esta declaración. Por ejemplo, si deseamos una Lista de EstudianteCurso podríamos tener declaraciones como las que se ven en el listado 6.14.

Código 6.14 Herencia con una interfaz

```
1:  class ListaCurso implements Lista {
2:      EstudianteCurso cabeza;
3:      ...
4:      Object busca(int cual, String cad) {
5:          ...
6:      }
7:      ...
8:  }
```

Dado que las interfaces corresponden a tipos, igual que las clases, podemos declarar variables de estos tipos. Por ejemplo,

```
Lista miLista;
```

y usarse en cualquier lugar en que se pueda usar un objeto de una clase que implementa a la interfaz Lista, de la misma manera que se puede usar una variable de la clase Object en cualquier lugar de cualquier objeto. Sin embargo, si deseamos

que el objeto sea visto como la subclase, tendremos que aplicar lo que se conoce como *"casting"*, que obliga al objeto a comportarse como de la subclase. Se logra poniendo el tipo al que deseamos conformar al objeto de tipo Object entre paréntesis, precediendo a la variable:

```
EstudianteCurso nuevo = (EstudianteCurso)busca(1,"Pedro");
```

El método busca nos regresa un objeto de tipo Object (la referencia), pero sabemos, por la implementación de busca que en realidad nos va a regresar un una referencia a un objeto de tipo EstudianteCurso, por lo que podemos aplicar el *casting*.

Una clase dada puede extender a una sola superclase, pero puede implementar a tantas interfaces como queramos. Como no tenemos la implementación de los métodos en las interfaces, aún cuando una misma firma aparezca en más de una interfaz (o inclusive en la superclase) la implementación que se va a elegir es la que aparezca en la clase, por lo que no se presentan conflictos.

Las interfaces también pueden extender a una o más interfaces de la misma manera que subclases extienden a superclases. La sintaxis es la misma que con la extensión de clases:

SINTAXIS:

interface $\langle identif_1 \rangle$ extends $\langle identif_2 \rangle, \ldots, \langle identif_n \rangle$

SEMÁNTICA:

De la misma manera que con las clases, la subinterfaz hereda todos los métodos de las superinterfaces.

En adelante se hará un uso más intenso de herencia y seguiremos, como hasta ahora, insistiendo en el uso de interfaces.

7 Administración de la memoria durante ejecución

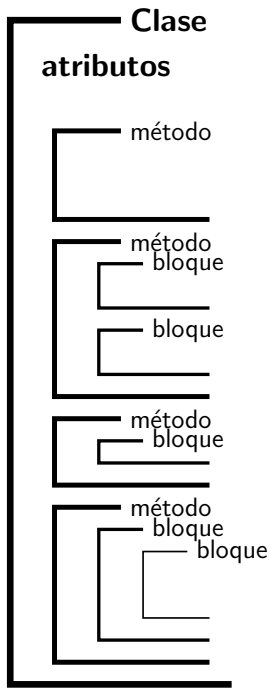
7.1 El stack y el heap

Revisitaremos el tema asignación de espacio durante la ejecución del programa, pues hay varios grandes detalles a los que no les hemos dado la importancia adecuada. Empecemos por lo que sucede en la memoria de la máquina durante la ejecución del programa y para ello veamos el concepto de la estructura de bloques de Java y qué sucede con ella durante ejecución. La estructura de bloques sintáctica (estáticamente) tiene la forma que se ve en la figura 7.1 en la siguiente página. Un archivo de Java puede tener una o más clases declaradas. Hasta ahora hemos hablado extensamente de los campos que se declaran en la clase y cómo son accesibles desde cualquier método de la misma. También hemos visto el papel que juegan los parámetros en los métodos de las clases, que corresponde a variables locales de las mismas. Adicionalmente a los parámetros tenemos las variables declaradas dentro de un método, a las que únicamente dentro de ese método, al igual que los parámetros, se tiene acceso – de ahí viene el nombre de *locales*.

Adicionalmente a esto, cuando se abren bloques de enunciados en las condicionales o en las iteraciones se pueden declarar variables que únicamente son locales dentro de ese bloque de enunciados. Sólo son conocidas dentro del bloque[1].

Esto nos define dos niveles lexicográficos distintos: el global, que se refiere a lo que se puede utilizar desde cualquier punto de cualquiera de nuestras clases, dados los permisos adecuados, y el local, que es aquello que se encuentra dentro de un método. Adicionalmente, dentro de los métodos podemos tener bloques de enunciados que incluyan declaraciones. Estas declaraciones son únicamente visibles dentro del bloque de instrucciones[2].

Figura 7.1 Estructura de bloques de un programa.



[1]Sin embargo, si existe alguna declaración de una variable con el mismo nombre fuera del bloque y que la precede, el compilador dará error de sintaxis por identificador ya declarado.

[2]No pueden tener el mismo identificador que una declaración previa y fuera del bloque.

Para los métodos estáticos, como es el caso del método main de las clases esto funciona un poco distinto, ya que este tipo de métodos no tiene acceso más que a los atributos o métodos estáticos de la misma clase, y a los métodos o atributos públicos o de paquete de las clases a las que se tenga acceso. Olvidándonos un poco de los métodos estáticos (de clase) podemos decir que la estructura de bloques nos da lo que conocemos como el *rango* de una variable, que se refiere a aquellos puntos de la clase donde la variable puede ser utilizada. Si regresamos a nuestro esquema de los espejos/cristales, tenemos que desde dentro de un bloque podemos ver hacia afuera: desde el nivel local tenemos acceso a las variables de la clase - el rango de las variables de la clase es toda la clase. Desde dentro de un método, sin embargo, no podemos ver lo que está declarado dentro de otro método o bloques. Pero, ¿qué pasa cuando el nombre de una variable de clase se repite dentro de un bloque como parámetro o variable local? En este caso, decimos que la variable local bloquea a la variable global: si existe una variable local con el mismo nombre, todas las referencias a esa variable en ese bloque se refieren a la variable local. El compilador utiliza a la variable que le "queda más cerca", siempre y cuando tenga acceso a ella, la pueda ver.

El rango de una variable es estático, pues está definido por la estructura del programa: de ver el listado podemos decir cuál es el rango de una variable dada, y definirlo como público (global, pero atado a un objeto) o local (declarado dentro de un método o como privado para una clase). Es el compilador el que se encarga de resolver todo lo relacionado con el rango de las variables, siguiendo la estructura de bloques.

Durante la ejecución del programa, esta estructura presenta un cierto "anidamiento", distinto del anidamiento sintáctico o lexicográfico, donde desde dentro de un método se puede llamar a cualquiera de los que está en rango, o sea, cualquiera de los métodos declarados en la clase o que sean públicos de otras clases. Se lleva a cabo una anidamiento dinámico durante ejecución, donde se establece una cadena de llamadas tal que el último método que se llamó es el primero que se va a abandonar.

Supongamos que tenemos la clase del listado 7.1.

En el esquema de la figura 7.2 en la página 203 tenemos los bloques en el orden en que son invocados, mostrando el anidamiento en las llamadas. El nombre del método junto con el valor de sus parámetros se encuentra sobre la línea que lo demarca. En la esquina superior derecha de cada bloque se encuentra el nivel de anidamiento dinámico que tiene cada método. El valor de las variables, tanto de los atributos como de las variables locales, dependerá del anidamiento lexicográfico. Por ejemplo, dentro del método B en el que se le asigna valor a una variable entera a, ésta es una variable local, por lo que el valor del atributo a no se va a ver modificado. Por ello, en la llamada de la línea 29: al método B, los valores con

los que es llamado son los originales de a y b, o sea 3 y 2.

Código 7.1 Clase que ilustra el anidamiento dinámico

```
1:  class Cualquiera {
2:
3:      private int a = 3, b = 2;
4:       ...
5:      public void A(int i) {
6:         ...
7:
8:         B(i, a);
9:         ...
10:     }
11:     ...
12:     public void B(int i, int j) {
13:
14:        int a = i + j;
15:        ...
16:        C();
17:        ...
18:     }
19:     public void C() {
20:
21:        int k = 2 * a;
22:        ...
23:     }
24:
25:     public static void main(String[] args) {
26:         int m = 10, n = 5;
27:         Cualquiera objeto = new Cualquiera();
28:         objeto.A(m);
29:         objeto.B(a, b);
30:         objeto.C();
31:     }
32: }
```

Al esquema de bloques le sigue un diagrama de Warnier en la figura 7.3 en la página opuesta que nos muestra la relación entre las llamadas.

Figura 7.2 Diagrama de anidamiento dinámico.

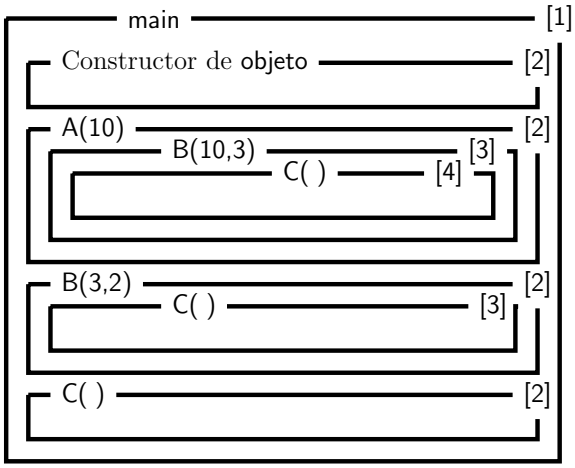
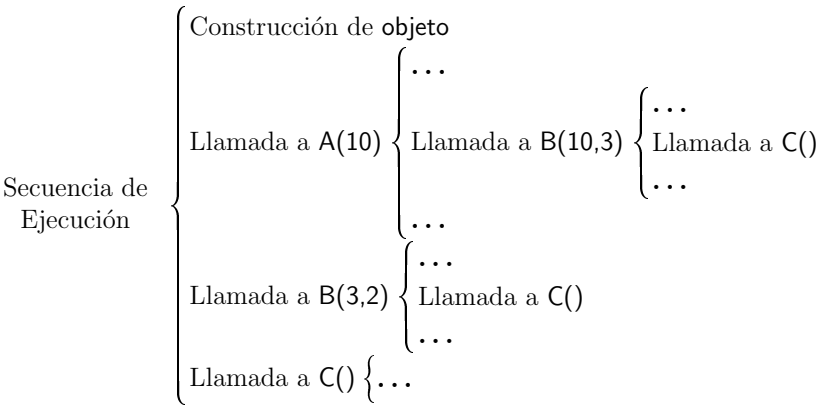


Figura 7.3 Secuencia de llamadas en el listado 7.1.



Lo que me indican los dos diagramas anteriores, es que la ejecución del programa debe proseguir de la siguiente manera:

1... Entrar a ejecutar main.

- **2...** Entrar a ejecutar el constructor de Cualquiera.
- **2...** Salir de ejecutar el constructor de Cualquiera.
- **2...** Entrar a ejecutar el método A(10).
  - **3...** Entrar a ejecutar el método B(10,3).
    - **4...** Entrar a ejecutar el método C().
    - **4...** Salir de ejecutar el método C().
  - **3...** Salir ejecutar el método B(10,3).
- **2...** Salir de ejecutar el método A(10).
- **2...** Entrar a ejecutar el método B(3,2).
  - **3...** Entrar a ejecutar el método C().
  - **3...** Salir de ejecutar el método C().
- **2...** Salir ejecutar el método B(3,2).
- **2...** Entrar a ejecutar el método C().
- **2...** Salir de ejecutar el método C().

1... Salir de ejecutar main.

Tanto en el esquema como en la secuencia de ejecución (donde omitimos para cada función la ejecución de lo que no fuera llamadas a métodos), asociamos un entero a cada llamada. Esto es con el objeto de identificar los anidamientos dinámicos - los que están definidos por la secuencia de ejecución del programa. Este esquema muestra varios aspectos importantes que tienen que ver con la ejecución de un programa. Revisemos algunos de ellos:

1. El anidamiento dinámico (en ejecución) no forzosamente coincide con el estático (sintáctico). Mientras que lexicográficamente hablando únicamente tenemos el nivel global y el local, dinámicamente podemos tener tantos niveles como queramos, uno por cada vez que desde dentro de una función llamamos a otra.
2. La última rutina a la que entramos es la primera de la que salimos.
3. Cuando aparece una función f como argumento de una función g, la llamada a f se hace inicia y termina antes que la llamada a g. Para poder llamar a g debemos tener el valor de sus argumentos, por lo que es necesario que antes de entrar a g obtengamos el valor de f.
4. El nivel dinámico que le corresponde a una función f que aparece como argumento de una función g es el mismo que el de la función g.

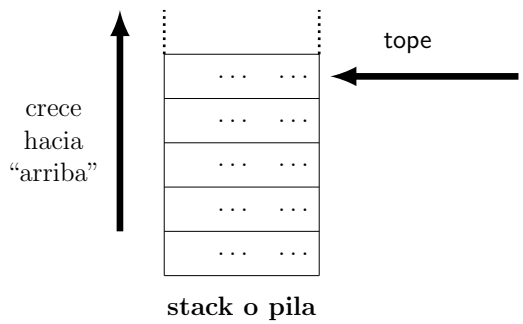
Para poder hacer esto, la ejecución del programa se lleva a cabo en la memoria de la máquina, organizada ésta como un stack, que es una estructura de datos con las siguientes características:

a) Respecto a su estructura:
   - La estructura es lineal, esto es, podemos pensarla con sus elementos "formados" uno detrás del otro.
   - Es una estructura homogénea, donde todos sus elementos son del mismo tipo.
   - Es una estructura dinámica, esto es, crece y se achica durante ejecución.
   - Tiene asociado un tope, que corresponde al último elemento que se colocó en el stack.

b) Respecto a su uso:
   - Un stack empieza siempre vacío, sin elementos.
   - Conforme progresa la ejecución, se van colocando elementos en el stack y se van quitando elementos del stack, siguiendo siempre esta regla: los elementos se colocan siempre en el tope del stack y cuando se remueven, se hace también del tope del stack.

Veamos un esquema de un stack en la figura 7.4.

Figura 7.4 Esquema de una stack o pila.

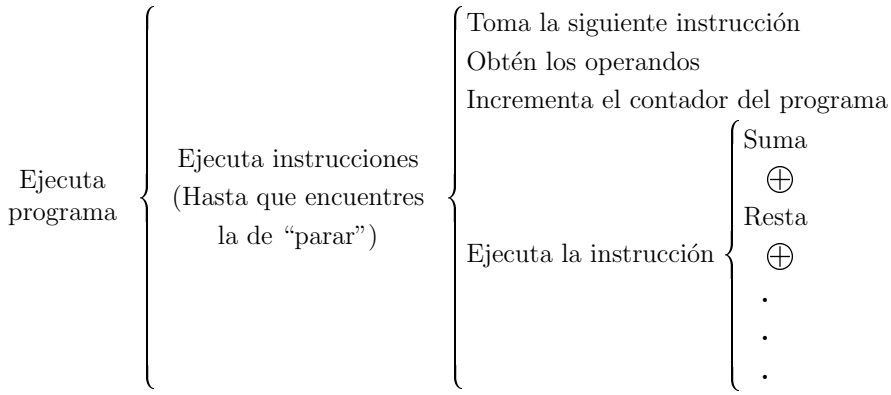


El *tope* del stack corresponde a un apuntador que me indica cuál es el siguiente lugar en el que se van a colocar datos en el stack. Suponiendo que la primera posición a ocupar en un stack es la 0, si el tope vale 0 quiere decir que el stack está vacío.

Para poder ejecutar un programa, el sistema cuenta con un contador de programa (*Program Counter:PC*) que apunta a (contiene la dirección de) la siguiente

instrucción a ejecutarse. El código del programa se encuentra en una sección de memoria, y las variables y la ejecución se hace sobre el stack. Los objetos se encuentran en el heap. El algoritmo para ejecutar un programa se encuentra en la figura 7.5.

Figura 7.5 Algoritmo para ejecutar un programa.



Antes de empezar a ejecutar un programa, el sistema operativo debe cargar en el stack de ejecución todo lo que corresponde a lo que está accesible para la clase que se va a ejecutar, que incluye los nombres de las clases accesibles y las variables y métodos de la clase que se va a ejecutar. A esto le llamamos el *paso 0* en la ejecución de un programa.

Cuando se está ejecutando un programa un método se puede invocar desde distintos puntos del programa. En el punto de llamada de un método la ejecución debe transferirse a ejecutar ese método, y una vez terminada le ejecución del mismo, regresar al punto desde donde se hizo la invocación, para continuar con la ejecución del programa. A la posición en la que se encuentra la llamada se le conoce como punto de llamada e indica el punto al que debe regresar la ejecución del programa una vez que termine la función. Esta última característica hace que se le utilice como dirección de regreso. La ejecución del programa, como ya mencionamos, se lleva a cabo en el stack. Cada vez que se invoca a un método - el programa principal main es una función invocada por el sistema operativo - se tiene que "montar" al método en el stack, anotando muy claramente a donde debe regresar la ejecución al terminar la rutina. Al terminar la ejecución del método,

se "desmonta" del stack al mismo. Para "montar" un método al stack hay que construir lo que se conoce como su registro de activación, que es una tabla en la que hay lugar para los parámetros y las variables locales del método, de tal manera que durante la ejecución se encuentren siempre en la parte superior del stack.

Al invocar un método (para transferirse a ejecutarlo), el sistema debe realizar los siguientes pasos:

1. Dejar un lugar en el stack para que el método coloque ahí el valor que va a regresar, si es que regresa valor.

2. Hacer la marca en el stack, copiando ahí el contenido del contador del programa.

3. Buscar en el stack, en el registro de activación global, la dirección de código donde se encuentra definida ese método. Copiar esa dirección al contador del programa (es donde va a continuar la ejecución cuando se termine de montar al método en el stack).

4. Construir el registro de activación del método, dejando un lugar para cada parámetro, en el orden en que están declarados, y un lugar para cada variable local (o estructura de datos pública o privada, si se trata de una clase).

5. Evaluar los argumentos, para entregarle al método una lista de valores e irlos colocando en el registro de activación.

6. Copiar al stack, en el orden en que aparece, el registro de activación (el último es el que queda en el tope del stack).

7. Copiar los valores de los argumentos a los parámetros.

8. Conforme se va ejecutando el método, se van colocando en el stack las variables y objetos que se van declarando.

Un método tiene acceso a su bloque local (lo que se encuentra a partir de la última marca en el stack) y al bloque global (lo que se encuentra en la base o fondo del stack y hasta la primera marca en el stack). En la base del stack se encuentran los identificadores de las clases a las que se tiene acceso desde la clase en ejecución.

Cuando termina la ejecución del método, el control debe regresar al punto de llamada. La ejecución del método termina cuando se llega a un enunciado de `return` o bien se llega al final del bloque que define al método. Antes de continuar la ejecución en el punto de llamada, el sistema tiene que hacer lo siguiente:

1. Localizar la marca del stack más cercana al tope, la última que se colocó.

2. Colocar en el contador del programa la dirección de regreso que se encuentra en esa marca.

3. Si el enunciado que causa la terminación de la rutina es un return, colocar el valor en el lugar inmediatamente abajo de la marca del stack correspondiente.

4. Quitar del stack todo lo que se encuentra a partir de la marca, incluyéndola. Se quita algo del stack simplemente "bajando" el apuntador al tope del stack a que apunte al último registro que se quitó (recordar que lo último que se colocó es lo primero que se quita). No hay necesidad de borrar la información pues la ejecución solo va a tomar en cuenta aquella información que se encuentre antes del tope del stack.

5. Continúa la ejecución en el lugar del código al que apunta el contador del programa.

Para ilustrar estos pasos, vamos a seguir el programa que escribimos, y que tiene los renglones numerados. Por supuesto que la ejecución del programa no se lleva a cabo directamente sobre el texto fuente. El compilador y ligador del programa producen un programa en binario (lenguaje de máquina) que se coloca en un cierto segmento de la memoria. El contador del programa va apuntando a direcciones de este segmento de memoria y en cada momento apunta a una instrucción de máquina. Es suficiente para nuestros propósitos manejar el programa a nivel de enunciado. En los esquemas del stack que presentamos a continuación, lo que corresponde a direcciones en memoria de datos (el stack) se preceden con un "*" mientras que lo que corresponde a memoria de programa se precede con un "#". El contador del programa apunta a la siguiente instrucción a ejecutarse. El tope del stack apunta al primer lugar vacío en el stack (si el tope del stack contiene un 0, quiere decir que no hay nadie en el stack).

Al ejecutar el paso 0, se cargan al stack todos los atributos y nombres de métodos de la clase[3], quedando el stack como se observa en la figura 7.6.

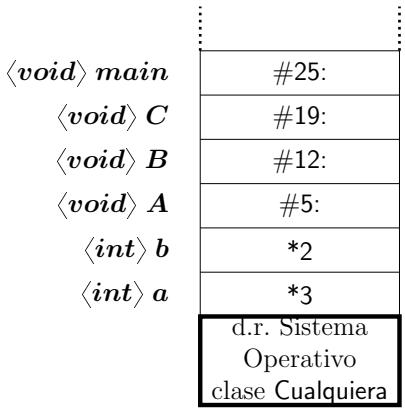
El sistema operativo sabe que el primer método a ejecutarse es main, por lo que inicia la ejecución con él. Sigamos los pasos, uno a uno, para ver como lo hace:

1. Como main no entrega valor, no deja espacio en el stack.

2. Marca el stack para poder montar al método.

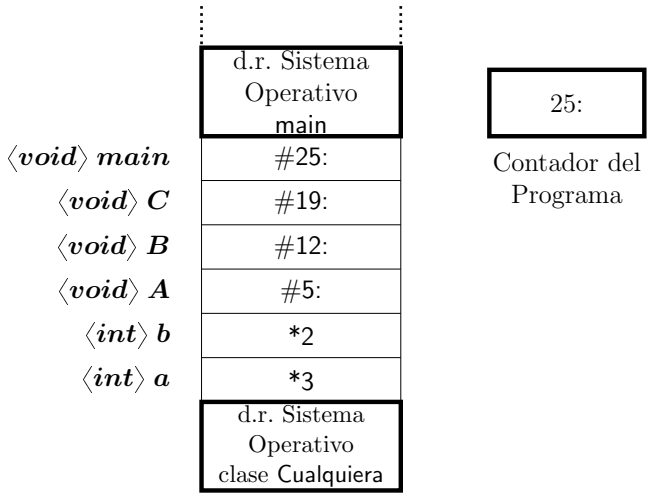
[3]No ilustraremos las clases a las que tiene acceso para ahorrar espacio, y porque sería prácticamente interminable.

Figura 7.6 Estado del stack al iniciarse la ejecución de una clase.



3. Localiza la dirección de main en el stack, y ve que es la dirección de código 24. El stack y los contadores quedan como se ve en la figura 7.7.

Figura 7.7 El stack al iniciarse la llamada a main.



4. Construye el registro de activación para main. El registro de activación construido se puede ver en la figura 7.8 en la siguiente página. En el registro se

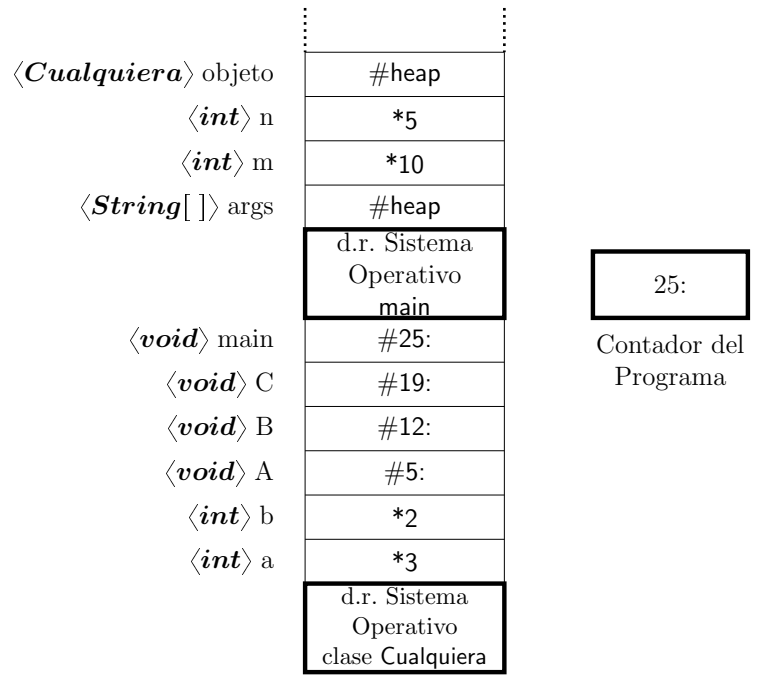
va dando lugar para cada una de las declaraciones locales. En el caso de declaraciones de objetos, se colocan en el stack las referencias a los objetos que se van a localizar en el heap.

Figura 7.8 Registro de activación para main.

⟨***Cualquiera***⟩ ***objeto***	# heap
⟨***int***⟩ ***n***	*5
⟨***int***⟩ ***m***	*10
⟨***String***[ ]⟩ ***args***	# heap

5 y 6 Se monta el registro de activación en el stack. El stack queda como se ve en la figura 7.9.

Figura 7.9 El stack listo para iniciar la ejecución de main.

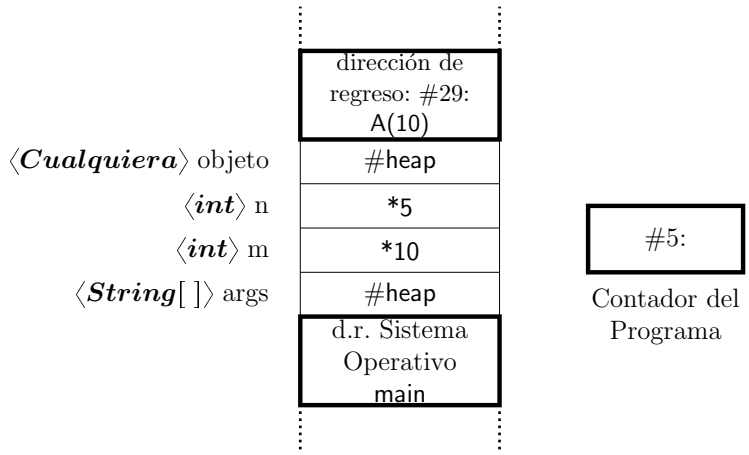


Una vez armado el stack, se procede a ejecutar la rutina. En este momento es accesible todo lo que corresponde a las variables y métodos públicos de las clases a las que se tiene acceso, a través de los objetos construidos, y lo que está desde la última marca hasta el tope del stack.

7. Empieza la ejecución en el enunciado #25:, con las declaraciones locales de main ya montadas en el stack.
   Las líneas de código 26: y 27: corresponden a las declaraciones que ya hicimos, así que procedemos a ejecutar la línea 28:. Para ello debemos invocar el método A del objeto objeto. Volvamos a seguir la ejecución, en lo que se refiere al manejo del stack.

   1 a 3: Como el método no regresa valor, no dejamos un lugar en el stack. Ponemos la marca, colocando en ella la dirección de código que se encuentre en el contador del programa. Asimismo, se coloca en el contador del programa la dirección del método. Todo esto se puede ver en la figura 7.10 en la siguiente página.

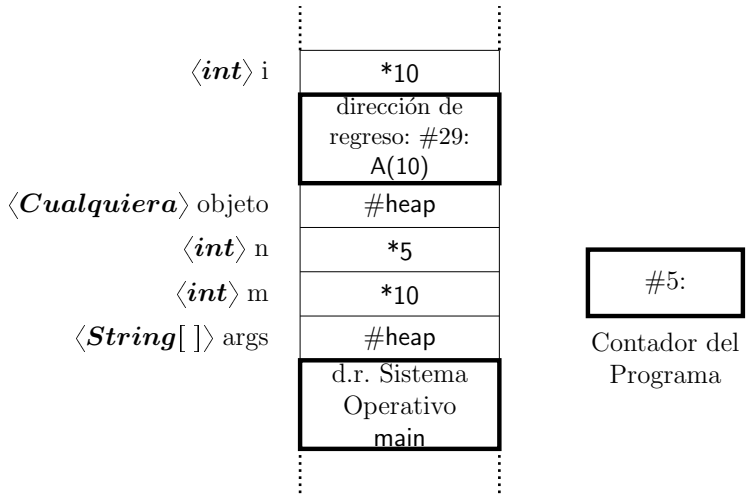
Figura 7.10 El stack durante la ejecución de main.



   4 a 6 Al evaluar los argumentos, tenemos la lista (10). Montamos en el stack el registro de activación y colocamos los valores de la lista en el espacio reservado para los argumentos. El contenido del stack en este momento se puede ver en la figura 7.11.

8. Continuar la ejecución del programa en la línea de código #5:.

Figura 7.11 El stack durante la ejecución de A.



En la línea #8: tenemos una llamada al método B(i,a), por lo que nuevamente marcamos el stack, copiamos la dirección del PC a la marca, armamos el registro de activación para B y lo montamos en el stack, colocamos la dirección donde empieza B a ejecutarse en el PC y proseguimos la ejecución en ese punto. En el momento inmediato anterior a que se ejecute B, el stack se presenta como se puede observar en la figura 7.12.

Al llegar a la línea de código #16: hay una llamada desde B al método C, por lo que nuevamente se marca el stack, se actualiza el contador del programa y se monta en el stack el registro de activación de C(). El resultado de estas acciones se pueden ver en la figura 7.13 en la página 214.

Se ejecuta el método C() y al llegar al final del mismo el sistema desmonta el registro de activación de C() del stack, coloca en el contador del programa la dirección que se encuentra en la marca y elimina la marca puesta por esta invocación. El stack se ve como en la figura 7.14 en la página opuesta.

Se termina la ejecución de B(10,3) en la línea #18:, por lo que se desmonta el registro de activación de b(10,3) del stack, se copia la dirección de regreso de la marca al PC y se quita la marca del stack, quedando el stack como se muestra en la figura 7.15.

Figura 7.12 El stack antes de empezar a ejecutar B.

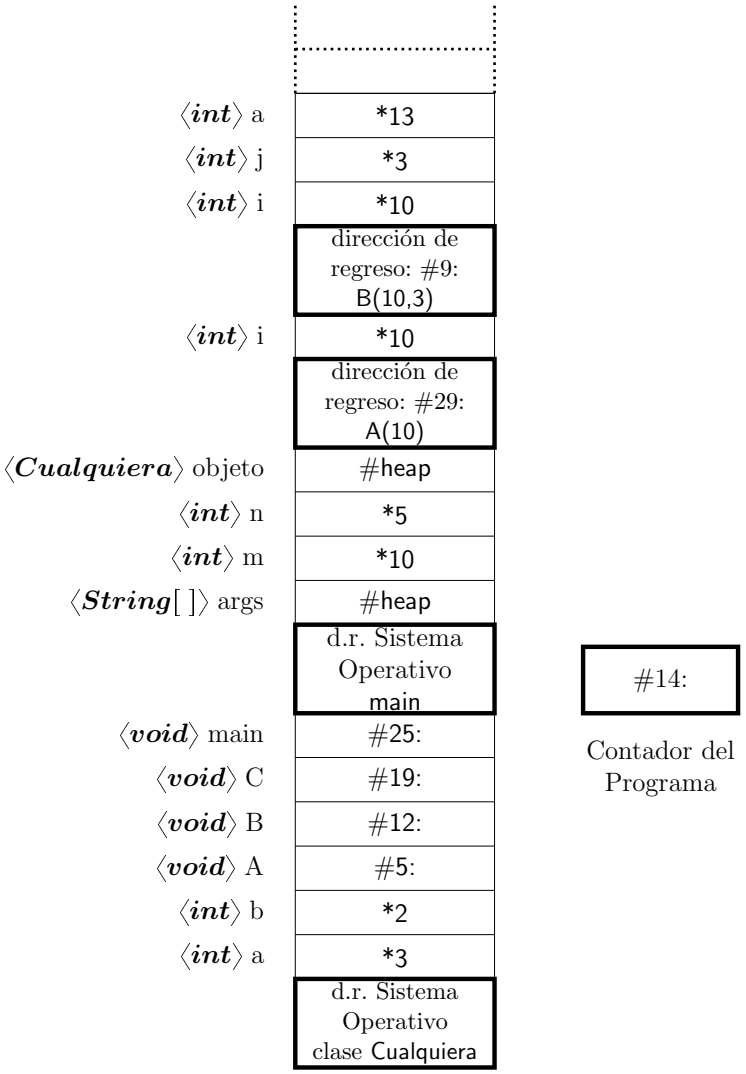


Figura 7.13 El stack antes de empezar a ejecutar C desde la línea #16:.

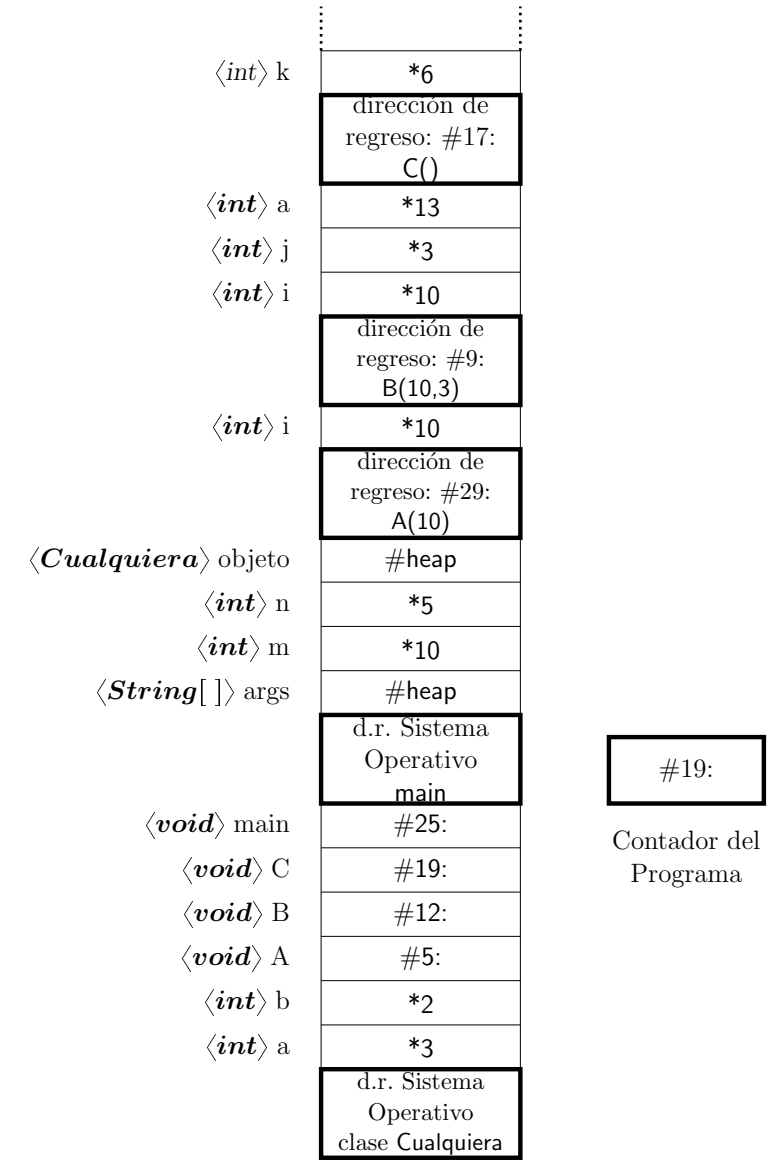


Figura 7.14 El stack al terminar de ejecutarse C().

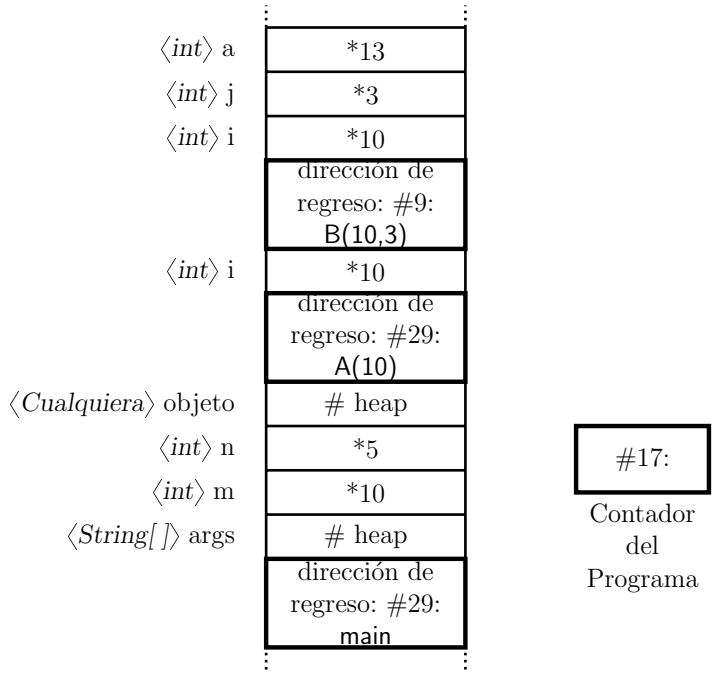
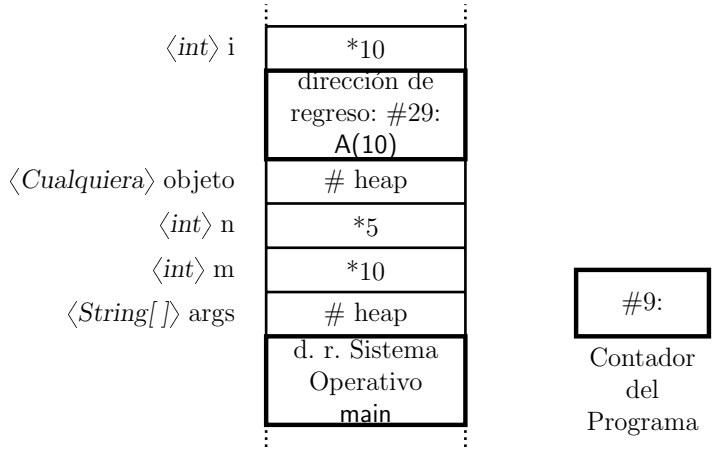
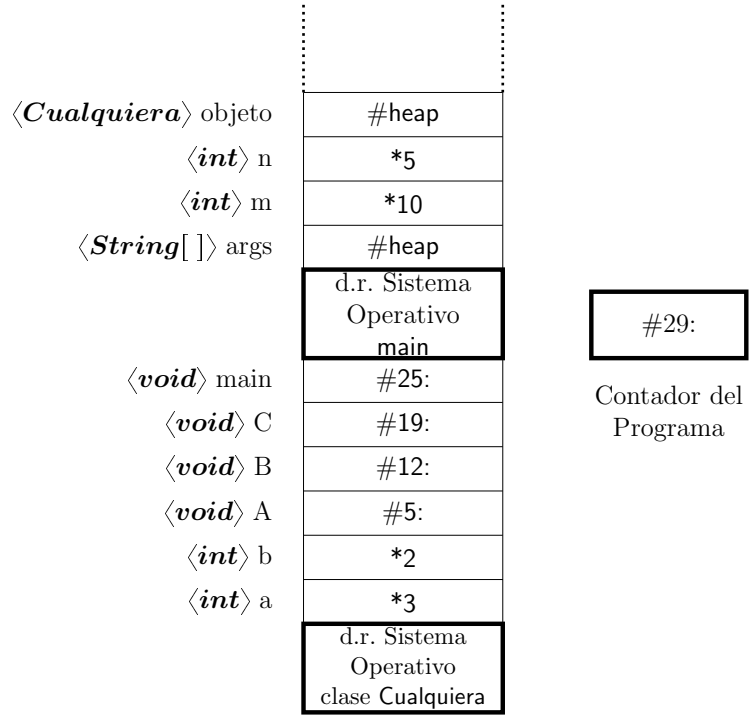


Figura 7.15 El stack al terminar la ejecución de B(10,3).



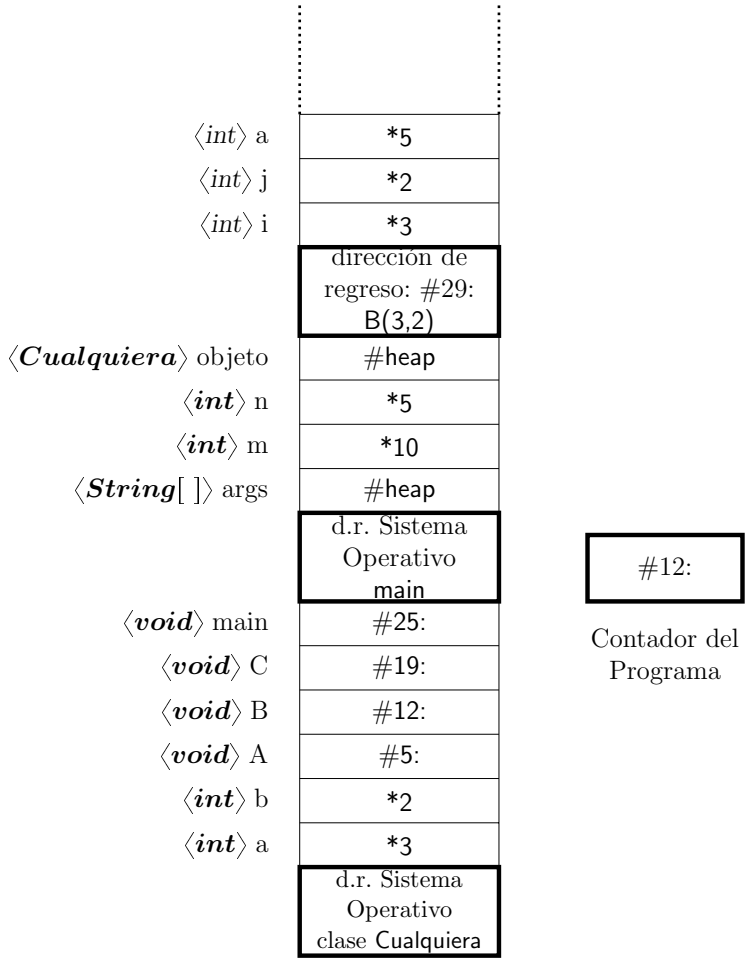
Se llega al final del método A(10), por lo que se desmonta el registro de activación de A(10), se copia la dirección de regreso de la marca al contador del programa y quita la marca del stack. Podemos observar el estado del stack en este momento en la figura 7.16.

Figura 7.16 El stack al terminar la ejecución de A(10).



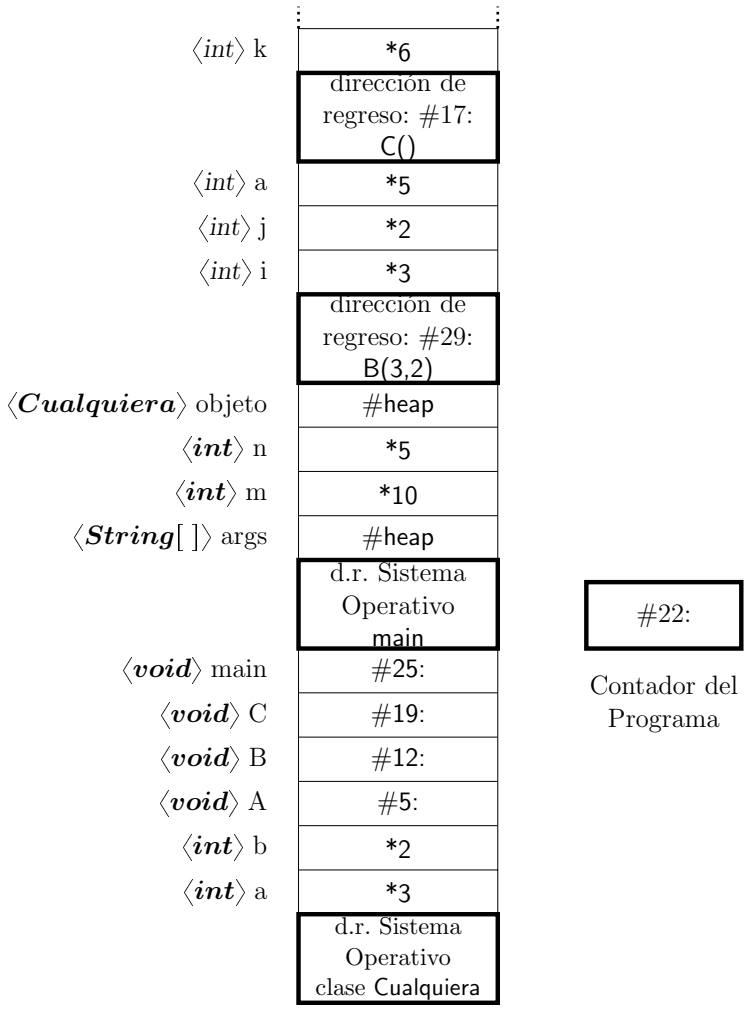
Al llegar la ejecución del programa a la línea 29: se encuentra con otra invocación a B(3,2), que son los campos de la clase. Se coloca la marca en el stack con dirección de regreso 30:, se actualiza el PC para que marque el inicio del método B y se monta al stack el registro de activación de B(3,2). Los resultados de estas acciones se muestran en la figura 7.17.

Figura 7.17 El stack antes de la ejecución de B(3,2).



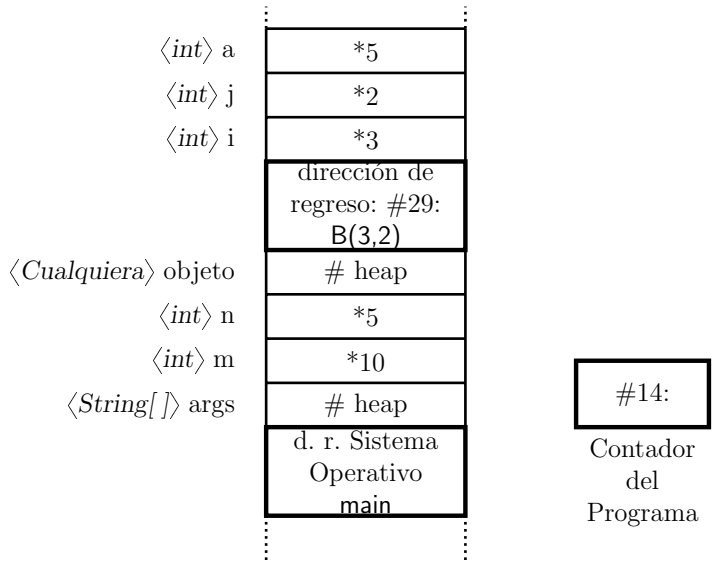
Al ejecutar al método B(3,2), en la línea 16: se invoca al método C(), por lo que el sistema hace lo conducente con el stack, quedando éste como se muestra en la figura 7.18.

Figura 7.18 El stack antes de la ejecución de C().



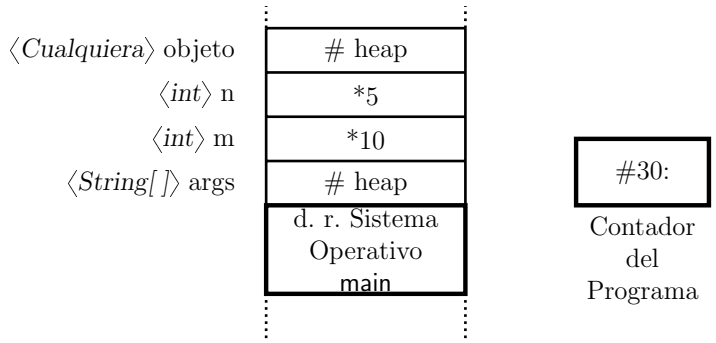
Termina de ejecutarse C() en la línea 18: y el stack regresa a verse como en la figura 7.17 en la página anterior, excepto que el PC vale ahora 17:. Esta situación se muestra en la figura 7.19.

Figura 7.19 El stack al terminar la ejecución deC().



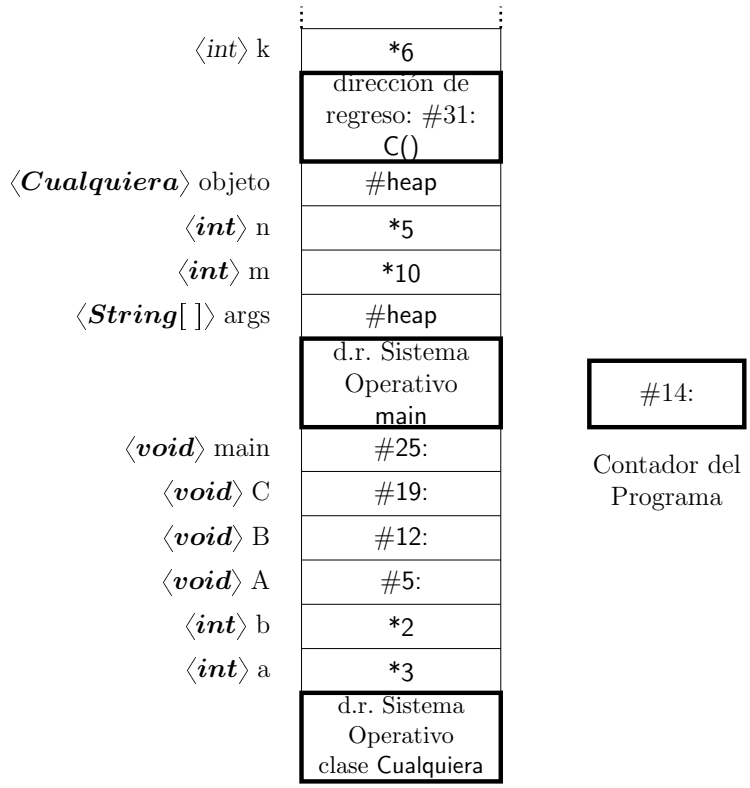
Al continuar la ejecución el programa, llega al final del método B(3,2) y sale de él, dejando el stack como se ve en la figura 7.20, con el PC apuntando a la dirección de código 30:.

Figura 7.20 El stack al terminar la ejecución de B(3,2).



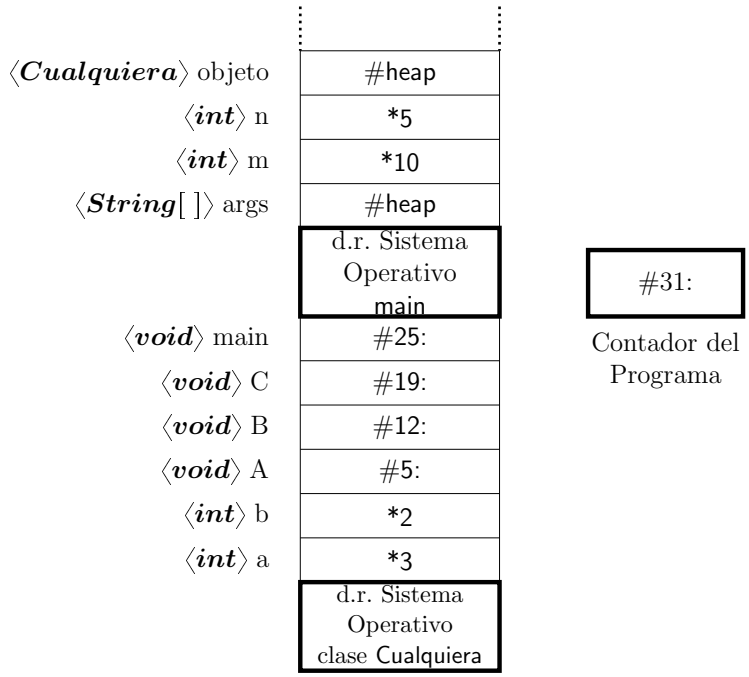
En la línea 30: nuevamente se hace una llamada al método C(), por lo que se marca el stack y se monta su registro de activación. El resultado se puede ver en la figura 7.21 en la página opuesta.

Figura 7.21 El stack antes de empezar la ejecución de C().



Al terminarse de ejecutar C() se desmonta su registro de activación del stack, se copia la dirección de regreso de la marca al PC y se quita la marca. El stack queda como se muestra en la figura 7.22, con el PC apuntando a la línea 31: del código.

Figura 7.22 El stack listo para iniciar la ejecución de main.



Como la línea 31: es la que termina main, se descarga del stack el registro de activación de este método, se copia al PC la dirección de regreso de la marca y se quita la marca. En ese momento termina la ejecución del programa, por lo que se libera el stack y el PC.

En todo momento durante la ejecución, el sistema puede utilizar lo que se encuentre en el bloque global, más aquello que se encuentre por encima de la última marca en el stack y hasta inmediatamente antes de la celda antes de la última marca que se puso. De esta manera, Cada método "crea" su propio ambiente de ejecución.

Resumiendo, el stack se utiliza para la administración de la memoria en ejecución. Cada vez que se invoca una rutina o método, se construye el registro de activación de la misma y se coloca en el stack. Cada vez que se sale de un método, se quita del stack el registro de activación de la rutina que está en el tope y la ejecución continúa en la dirección de regreso desde la que se invocó a esa instancia

del método.

Durante la ejecución de un programa, el sistema trabaja con dos variables, el tope del stack, que indica cuál es la siguiente celda en la que se va a colocar información, y el contador del programa, que indica cuál es la siguiente instrucción que se va a ejecutar. En ambos casos, decimos que las variables son apuntadores, pues el tope del stack apunta a una celda en el stack (contiene una dirección del stack) y el contador del programa apunta a una dirección de memoria del programa donde se encuentra almacenado el código del programa.

En el stack se le da lugar a:

- Todo lo declarado público en el paquete (o conjunto de programas).
- Todas las estructuras de datos de las clases.
- Apuntadores a todos los métodos miembros de clases.
- A los resultados que entregan las funciones.
- A los parámetros formales de cada método.
- A las variables locales de cada método conforme se van declarando.

Decimos que cada método tiene su ambiente propio de trabajo, en la medida en que, al cargarse su registro de activación en el stack, su entorno lo constituye ese registro de activación y el registro de activación global. En nuestros esquemas, las celdas intermedias entre la primera y última marca no son accesibles en ese momento de la ejecución. Esto nos da dos conceptos importantes en programación:

Rango de un identificador Se refiere a los puntos del programa desde donde el identificador puede ser referido. El rango está dado de manera estática por la estructura de bloques del programa. El compilador se encarga de que las referencias a los identificadores sean válidas.

Existencia de una variable Se refiere a los momentos, durante la ejecución, en que una variable conserva su valor. Una variable declarada existe mientras se encuentre en el stack. Deja de existir cuando se quita del stack el registro de activación que la contiene.

Como ya mencionamos antes, de existir identificadores duplicados, el compilador busca a la declaración más cercana en el stack, pero busca únicamente en los registros de activación vivos (despiertos), el global y el local, primero en el local. Por ello, al declarar una variable repitiendo un nombre global, se crea una instancia fresca y nueva, que no tiene relación alguna con la variable global original y que de hecho, oculta a la variable global original. Java permite ver datos

miembros de una clase que han sido ocultados por declaraciones locales utilizando el identificador de objeto this seguido del operador "." y a continuación el nombre del atributo. Debo insistir en que el bloque o registro de activación en el que se encuentra la variable debe ser visible desde el punto de ejecución y únicamente se aplica a variables que hayan sido ocultadas por una reutilización del nombre. Si la variable se encuentra en un registro de activación inaccesible, entonces el compilador emitirá un mensaje de error.

7.2 Recursividad

En matemáticas nos encontramos frecuentemente con definiciones o funciones recursivas. El ejemplo típico de este tipo de funciones es la definición del método factorial:

$$\boldsymbol{n!} \begin{cases} \mathbf{1} & \boldsymbol{si} \;\; \boldsymbol{n} = \mathbf{1} \\ \\ \boldsymbol{n} * (\boldsymbol{n} - \mathbf{1})\mathbf{!} & \boldsymbol{si} \;\; \boldsymbol{n} > \mathbf{1} \end{cases}$$

Si escribimos un método en Java que refleje esta definición, tendríamos lo siguiente:

```
long factorial(int n)  {
    if (n <= 0)   {
       return -1;
    }
    if (n > 1)    {
       return ( factorial( n-1) * n);
    }
    else   {
      return 1;
    }
}
```

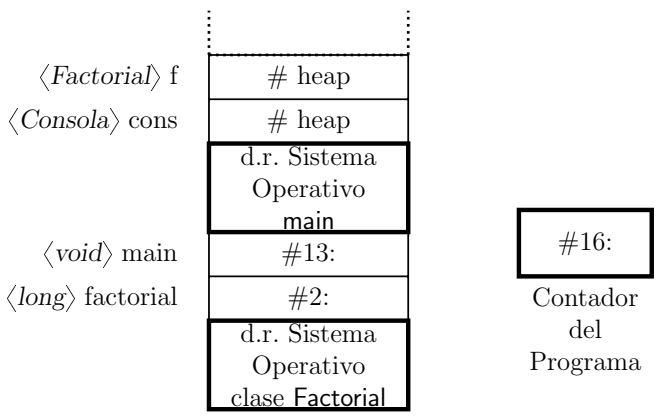
La ejecución del método es como sigue: en la invocación desde "fuera" (el método main o algún otro método), se llama con un argumento $\boldsymbol{n}$, que debe ser un valor entero. Si el entero es mayor que 1, procede el método a llamarse nuevamente a sí mismo, pero disminuyendo en 1 al argumento. Si el argumento vale 1, el método logra salir. Veamos la ejecución de este método dentro de una clase en el listado 7.2, para poder seguir su ejecución en el stack.

Código 7.2 La función factorial

```
1:    class Factorial    {
2:        long factorial(int n)  {
3:            if (n <= 0)  {
4:              return -1;
5:            }
6:            if (n > 1)  {
7:                return ( factorial( n-1)
8:                                * n);
9:            } else  {
10:               return 1;
11:           }
12:       }
13:       public static void main(String[] args)  {
14:           Consola cons = new Consola();
15:           Factorial f = new Factorial();
16:           cons.imprimeln("4! es " +
17:                           f.factorial(4));
18:       }
19:   }
```

El stack, una vez que se cargó el registro de activación de main, se muestra en la figura 7.23.

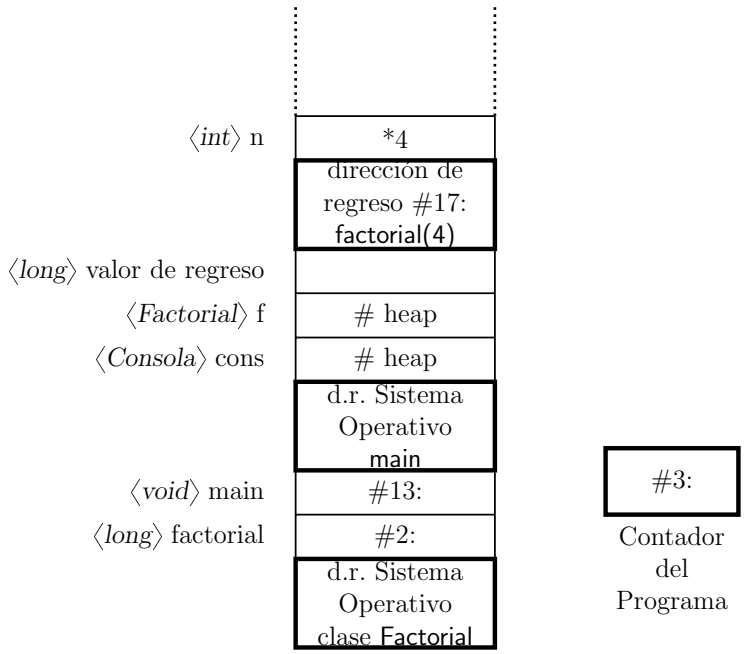
Figura 7.23 Estado del stack al iniciarse la ejecución de una clase.



Numeramos las líneas del programa para poder hacer referencia a ellas en

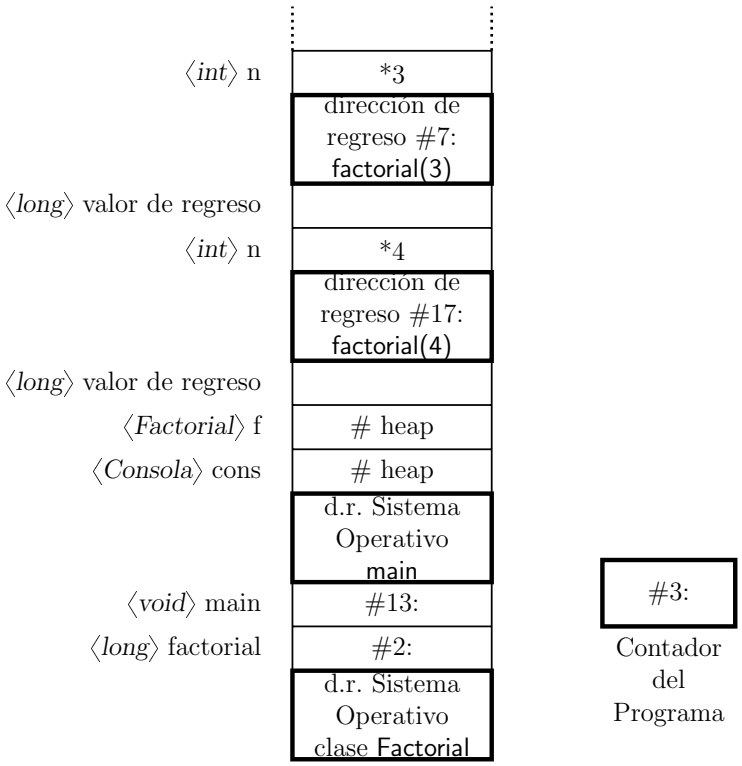
la ejecución, que empieza en la línea 13:. En las líneas 13: y 15: tenemos las declaraciones e inicializaciones de variables locales a main, y en las líneas 16: y 17: está el único enunciado realmente de ejecución de main. La primera llamada de factorial desde main deja el stack como se ve en la figura 7.24

Figura 7.24 Estado del stack al iniciarse la llamada de factorial desde main.



Se ejecuta la condicional de la línea 3:, pero como ***n*** es mayor que 1 no se hace nada. Después se evalúa la condicional de la línea 6:, y como es verdadera se ejecuta el enunciado en las líneas 7: y 8: que es una llamada recursiva a factorial. Vuelve a entrar a ejecutar factorial y el stack se ve como en la figura 7.25 en la siguiente página.

Figura 7.25 Estado del stack al iniciarse la llamada de factorial desde factorial.



Nuevamente se evalúa a falso la primera condición, y como se evalúa a verdadero la condición en la línea 6: volvemos a llamar a factorial desde la línea 7:, quedando el stack como se puede apreciar en la figura 7.26 en la página opuesta.

Como ***n*** sigue siendo mayor que 1, volvemos a llamar a factorial con 1 como argumento. El stack se ve como se muestra en la figura 7.27 en la página 228.

En esta llamada las condicionales de las líneas 3: y 6: ambas se evalúan a falso, por lo que se ejecuta el enunciado de la línea 10:, y se regresa el valor 1. Esto se traduce en colocar en el espacio reservado para ello cerca del tope del stack ese valor, quitar del tope del stack el registro de activación de la llamada de factorial(1) y continuar la ejecución en la línea 8: para hacer la multiplicación. El stack se ve como se muestra en la figura 7.28 en la página 229.

En este punto se puede terminar de ejecutar la invocación de factorial(2), por lo que nuevamente se hace la multiplicación y se coloca el resultado inmediatamente abajo de la última marca en el stack; se procede a desmontar el registro de activación de factorial(2).El stack se muestra en la figura 7.29 en la página 230.

Figura 7.26 Estado del stack al iniciarse la llamada de factorial desde factorial.

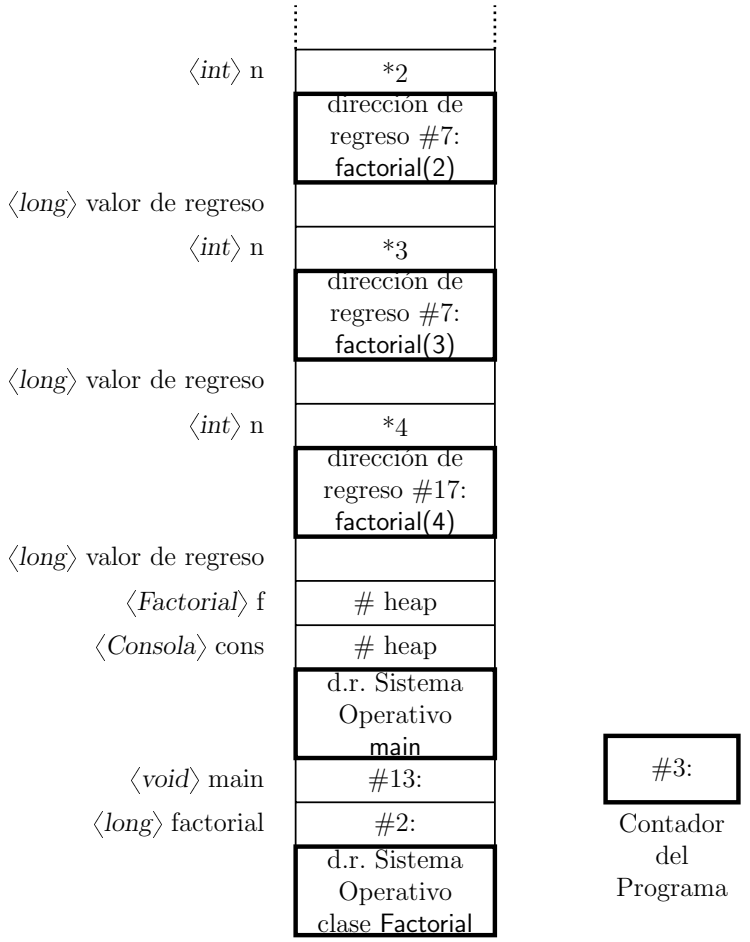


Figura 7.27 Estado del stack al iniciarse la llamada de factorial desde factorial.

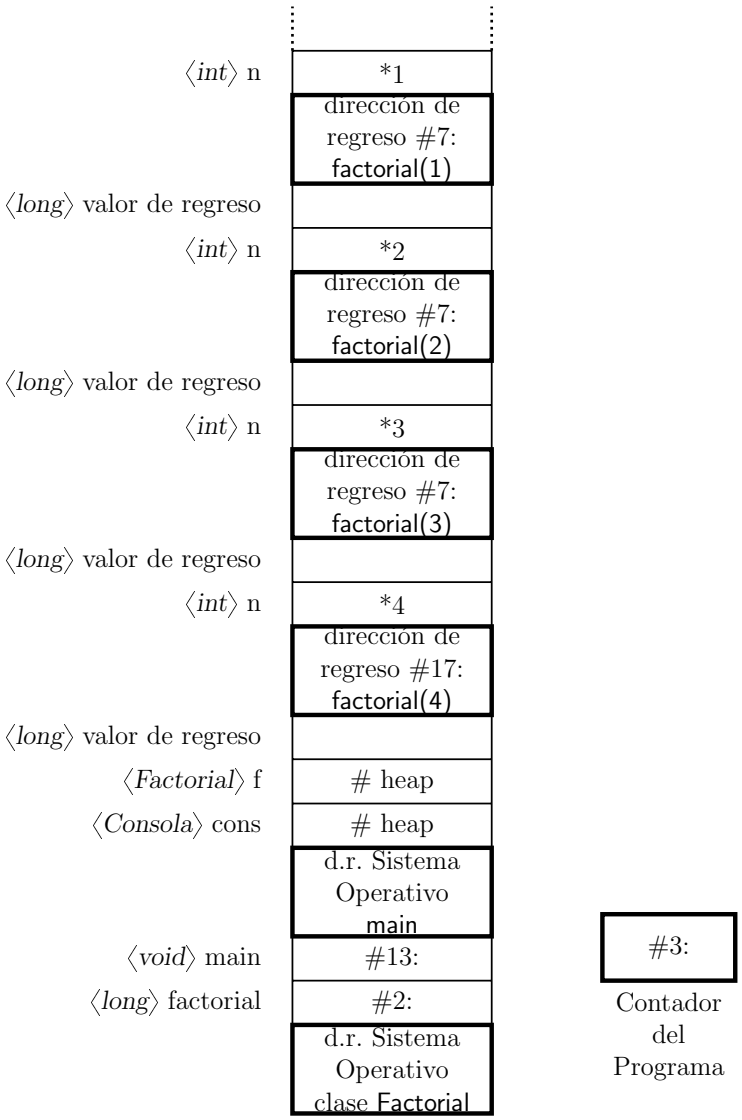
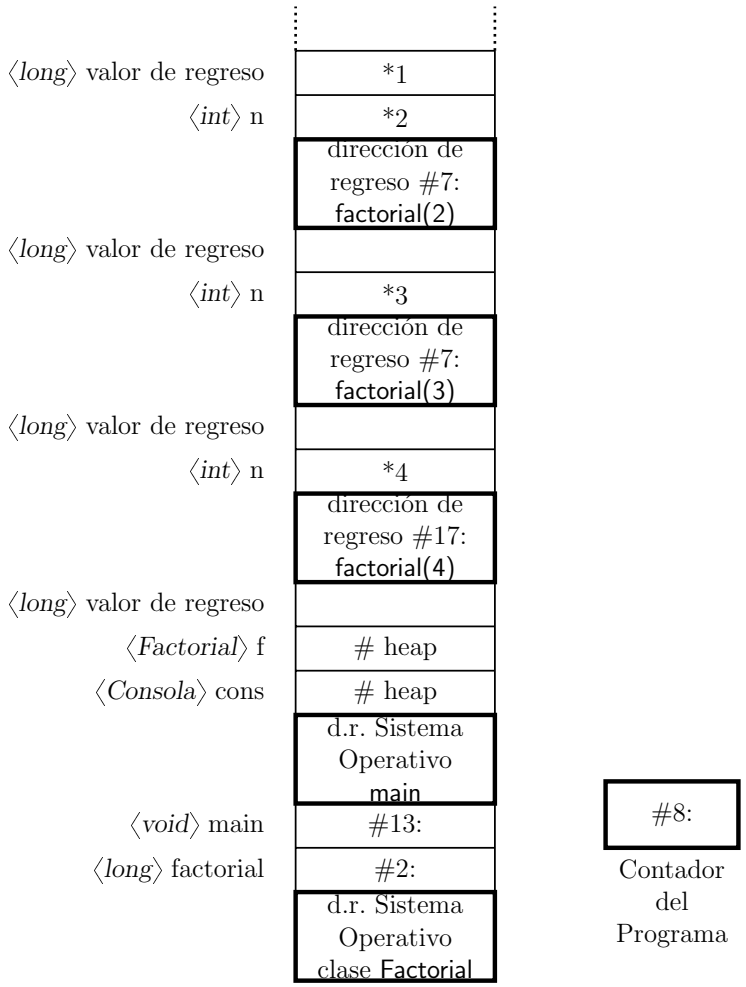


Figura 7.28 Estado del stack al terminarse la llamada de factorial(1).

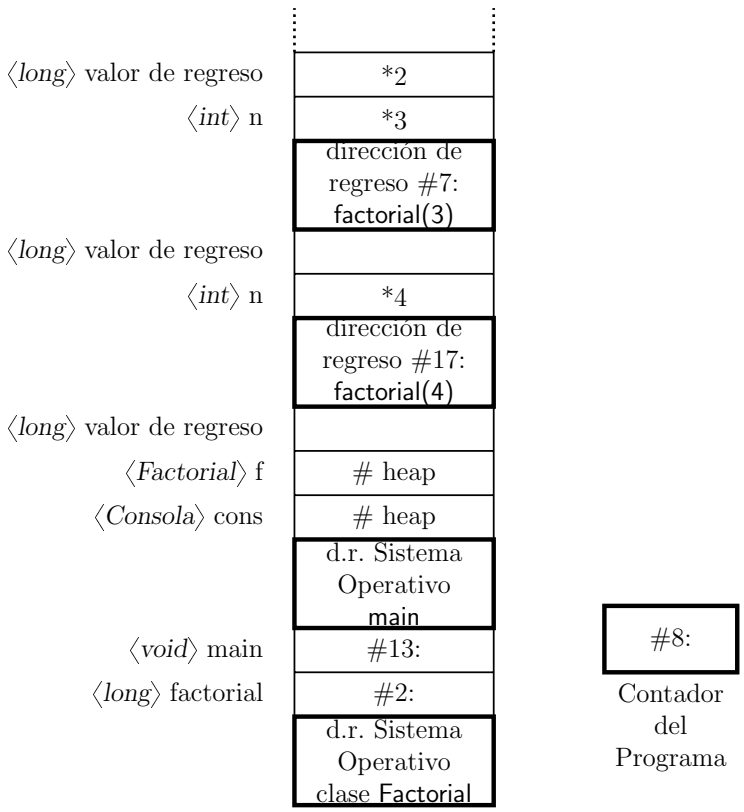


Se sale de la llamada de factorial(3), acomodando el resultado del producto del resultado entregado por factorial(2) y 3. El stack se puede ver en la figura 7.30 en la página 231.

Nuevamente, al regresar con el valor de factorial(3) en la línea 7:, lo multiplica por 4, usando para ello el valor que colocó la ejecución de factorial(3) en el stack.

Después de quitar el registro de activación y la marca de factorial(3) el stack se ve como se muestra en la figura 7.31 en la página opuesta.

Figura 7.29 Estado del stack al terminarse la llamada de factorial(2).



En este momento se procede a escribir el valor de factorial(4) que fue entregado y colocado en el stack. Se termina de ejecutar el programa, y de manera similar a métodos no recursivos, se libera el stack y el contador del programa.

Figura 7.30 Estado del stack al terminarse la llamada de factorial(3).

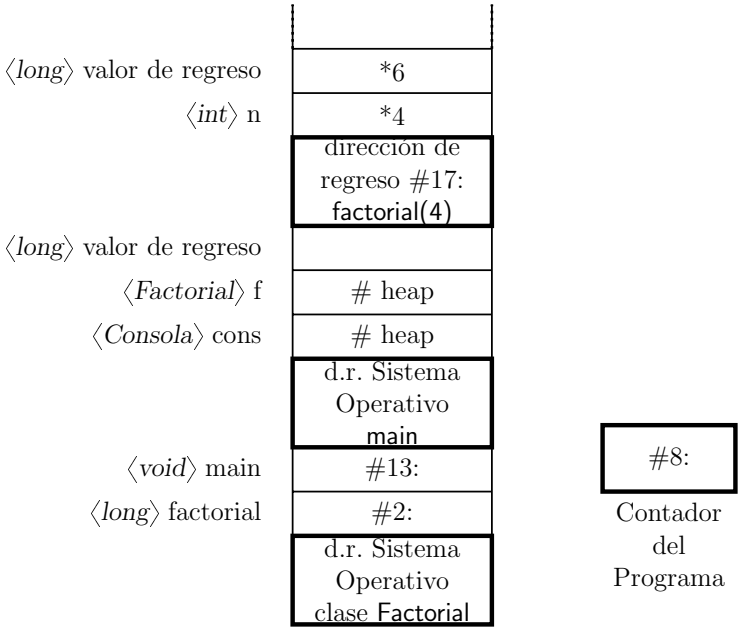
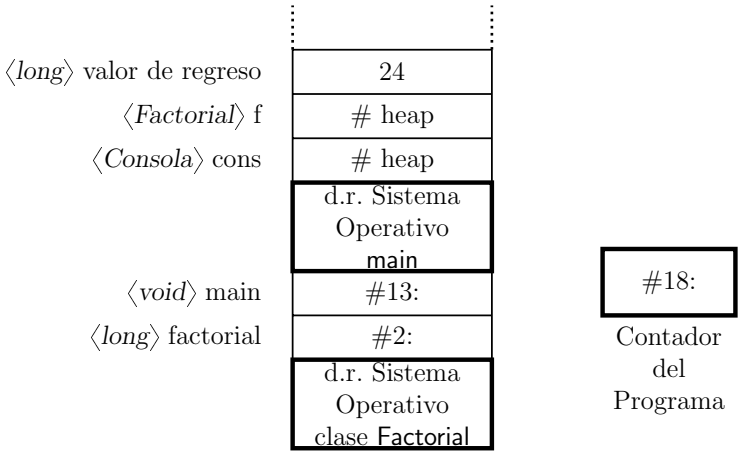


Figura 7.31 Estado del stack al terminarse la llamada de factorial desde main.



Debemos insistir en que usamos la función factorial para mostrar la situación en el stack porque es fácil de mostrar los cambios que va sufriendo el stack, no porque sea un buen ejemplo para la recursividad. De hecho, dado que ya vimos el trabajo escondido que tenemos con la recursividad, una forma más económica de calcular factorial es con una iteración simple, que no involucra manejo del stack. El método quedaría codificado como se ve en el listado lis:s7-3.

Código 7.3 Factorial calculado iterativamente

```
static public long factorial(int tope)  {
   long fact = 1;
   for ( int i = 2; i <= tope; i++)  {
       fact = fact * i;
   }
   return fact;
}
```

7.2.1. Las torres de Hanoi

Un ejemplo donde se entiende muy bien la utilidad de la recursividad es en el de las torres de Hanoi. El juego consiste de los siguiente:

> El juego consiste de una tabla con tres postes pegados perpendiculares a la tabla y $\boldsymbol{n}$ discos de radios distintos entre sí y con un orificio en el centro para poder ser colocados en los postes. Al empezar el juego se encuentran los $\boldsymbol{n}$ discos en un mismo poste, acomodados por tamaño decreciente, el más grande hasta abajo.
>
> El juego consiste de pasar los $\boldsymbol{n}$ discos a un segundo palo, moviendo disco por disco; está prohibido que quede un disco encima de otro que es menor que él.

En la figura 7.32 se muestra un ejemplo con 8 fichas.

Este juego tiene su origen en un monasterio tibetano, y consistía de 64 fichas. La leyenda decía que cuando se lograran mover las 64 fichas siguiendo las reglas el mundo se iba a terminar. El algoritmo para lograr mover las $\boldsymbol{n}$ fichas se muestra en la figura 7.33.

Figura 7.32 Juego de las torres de Hanoi

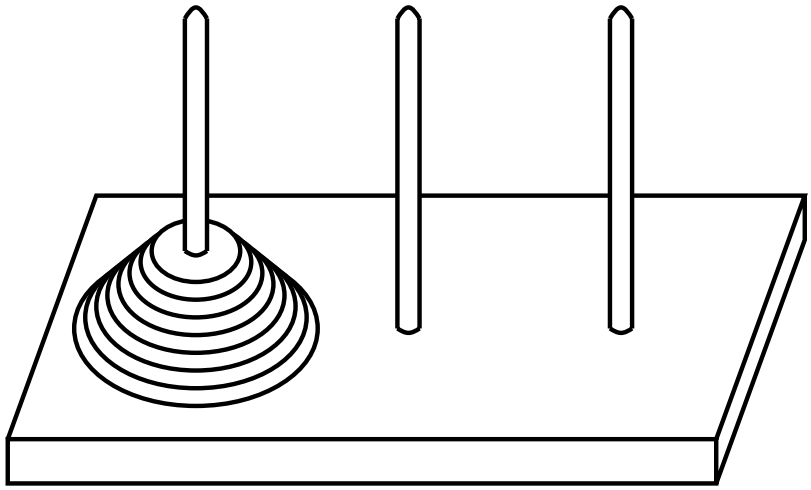


Figura 7.33 Estrategia recursiva para las torres de Hanoi

Mover $\boldsymbol{n}$ fichas del poste 1 al poste 2

- ¿$\boldsymbol{n} = \mathbf{2}$?
  - Mover 1 ficha del poste 1 al 3
  - Mover 1 ficha del poste 1 al 2
  - Mover 1 ficha del poste 3 al 2
- $\oplus$
- ¿$\boldsymbol{n} > \mathbf{2}$?
  - Mover $\boldsymbol{n} - \mathbf{1}$ fichas del poste 1 al 3
  - Mover 1 ficha del poste 1 al 2
  - Mover $\boldsymbol{n} - \mathbf{1}$ fichas del poste 3 al 2

Lo que me dice esta estrategia es que si sólo tenemos dos fichas las sabemos mover "a pie". Para el caso de que tenga más de dos fichas ($\boldsymbol{n} > \mathbf{2}$), suponemos que pudimos mover las $\boldsymbol{n} - \mathbf{1}$ fichas que están en el tope del poste al poste auxiliar, siguiendo las reglas del juego, después movimos una sola ficha al poste definitivo, y para terminar movimos las $\boldsymbol{n}-\mathbf{1}$ fichas del poste auxiliar al definitivo. Como en el caso del cálculo de factorial con recursividad, se entra al método

decrementando la n en 1 hasta que tengamos que mover una sola ficha; en cuanto la movemos, pasamos a trabajar con el resto de las fichas. Hay que aclarar que esto funciona porque se van intercambiando los postes 1, 2 y 3. El código (de manera esquemática) se puede ver en el listado 7.4.

Código 7.4 Métodos para las torres de Hanoi

```
/**
  * Realiza el movimiento de una ficha de un poste a otro.
  * @param poste1 El poste desde el que se mueven las fichas.
  * @param poste2 El poste al que se mueve la ficha.
  */
public void mueveUno(int poste1, int poste2) {
    /*  Escribe el número de poste correspondiente
     *  o dibuja el movimiento                      */
  System.out.println("Del " + poste1 + " al " + poste2);
}

/**
  * Mueve n fichas del poste1 al poste2, usando el
  * poste3 como poste de trabajo.
  * @param n el número de fichas a mover.
  * @param poste1 el poste desde el cual se mueven.
  * @param poste2 el poste destino.
  * @param poste3 el poste de trabajo.
  */
public void mueveN(int n, int poste1, int poste2, int poste3) {
   if (n == 2) {
       mueveUno(poste1, poste3);
       mueveUno(poste1, poste2);
       mueveUno(poste3, poste2);
   }
   else {
       mueveN(n-1,poste1, poste3, poste2);
       mueveUno(poste1, poste2);
       mueveN(n-1,poste3, poste2, poste1);
   }
}
```

Podemos hacer el ejercicio con 4 fichas, llamando al procedimiento con mueveN(4,1,2,3). Veamos en la 7.34 los anidamientos que se hacen. Debe ser claro que en el único método que realmente hace trabajo es mueveUno, ya que para $n > 2$ todo lo que se hace es una llamada recursiva. Ilustraremos en las figuras 7.35 a 7.42 cuál es el trabajo realizado en las llamadas a mueveUno.

Figura 7.34 Secuencia de llamadas en la torres de Hanoi

```
mueveN(4,1,2,3)
    ¿4 == 2?
        mueveN(3,1,3,2)
            ¿3 == 2?
                mueveN(2,1,2,3)
                    ¿2 == 2?
                        mueveUno(1,3)     /* 1 */
                        mueveUno(1,2)     /* 2 */
                        mueveUno(3,2)     /* 3 */
                mueveUno(1,3)             /* 4 */
                mueveN(2,2,3,1)
                    ¿2 == 2?
                        mueveUno(2,1)     /* 5 */
                        mueveUno(2,3)     /* 6 */
                        mueveUno(1,3)     /* 7 */
        mueveUno(1,2)                     /* 8 */
        mueveN(3,3,2,1)
            ¿3 == 2?
                mueveN(2,3,1,2)
                    ¿2 == 2?
                        mueveUno(3,2)     /* 9 */
                        mueveUno(3,1)     /* 10 */
                        mueveUno(2,1)     /* 11 */
                mueveUno(3,2)             /* 12 */
                mueveN(2,1,2,3)
                    ¿2 == 2?
                        mueveUno(1,3)     /* 13 */
                        mueveUno(1,2)     /* 14 */
                        mueveUno(3,2)     /* 15 */
```

Comprobemos que este algoritmo trabaja viendo una visualización con cuatro fichas. En cada figura mostraremos los movimientos que se hicieron mediante flechas desde el poste en el que estaba la ficha al poste en el que se colocó. Las reglas exigen que cada vez que se mueve una ficha, ésta sea la que se encuentra hasta arriba.

Figura 7.35 Situación de las fichas antes de la llamada

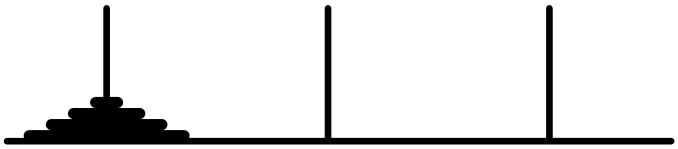


Figura 7.36 Movimientos /* 1 */ al /* 3 */

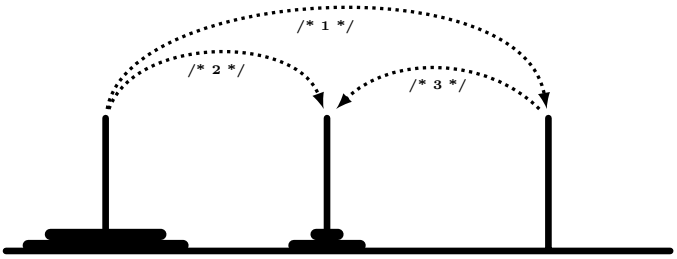


Figura 7.37 Movimiento /* 4 */

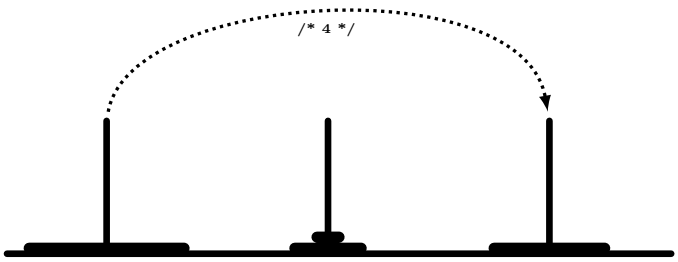


Figura 7.38 Movimientos /* 5 */ al /* 7 */

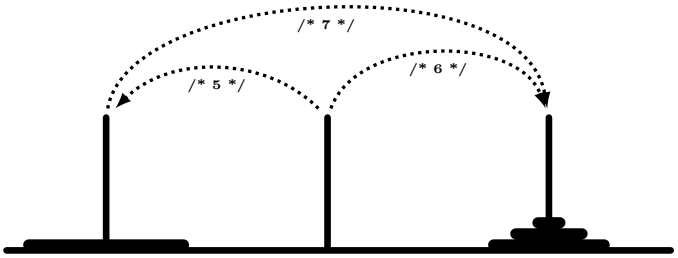


Figura 7.39 Movimiento /* 8 */

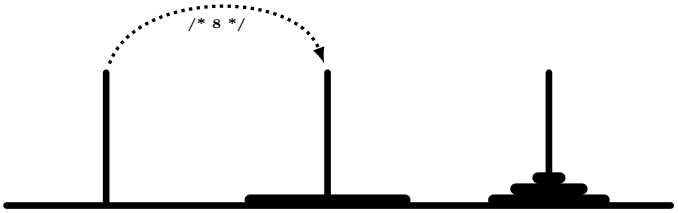


Figura 7.40 Movimientos /* 9 */ al /* 11 */

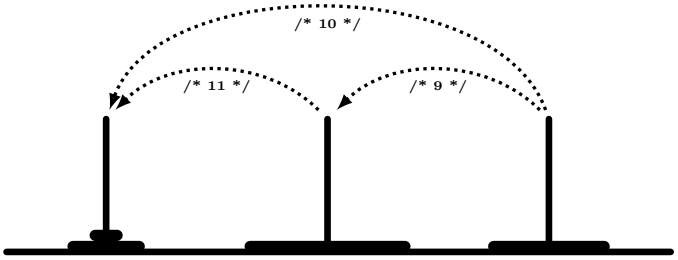


Figura 7.41 Movimiento /* 12 */

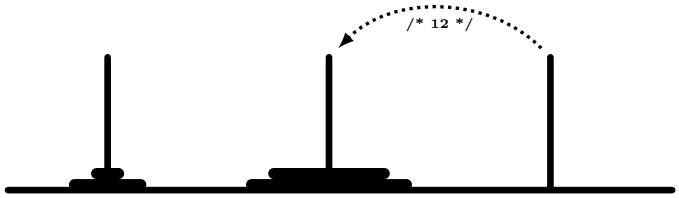
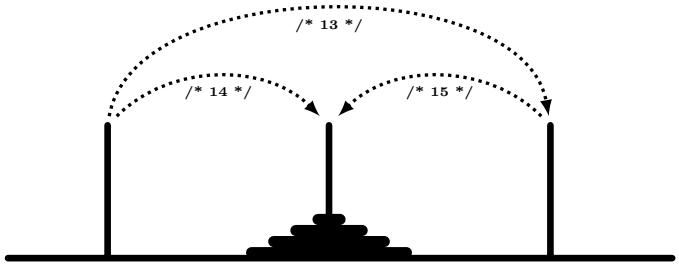


Figura 7.42 Movimientos /* 13 */ al /* 15 */



Como se puede ver del ejercicio con las torres de Hanoi, 4 fichas provocan 15 movimientos. Podríamos comprobar que 5 fichas generan 31 movimientos. Esto se debe a la recursividad, que se encuentra "escondida" en la simplicidad del algoritmo. Aunque definitivamente es más fácil expresarlo así, que ocupa aproximadamente 10 líneas, que dar las reglas con las que se mueven las fichas de dos en dos.

Entre otros ejemplos que ya no veremos por el momento, donde la solución recursiva es elegante y mucho más clara que la iterativa se encuentra el recorrido de árboles, las búsquedas binarias y algunos ordenamientos como el de mezcla y el de Quick.

Con esto damos por terminado una descripción somera sobre cómo se comporta la memoria durante la ejecución de un programa, en particular el stack de ejecución. Esta descripción no pretende ser exhaustiva, sino únicamente proporcionar una idea de cómo identifica la ejecución los puntos de entrada, de regreso y parámetros a una función.

8 Ordenamientos usando estructuras de datos

8.1 Base de datos en un arreglo

Habiendo ya visto arreglos, se nos ocurre que puede resultar más fácil guardar nuestras listas de cursos en un arreglo, en lugar de tenerlo en una lista ligada. Todo lo que tenemos que hacer es pensar en cuál es el tamaño máximo de un grupo y reservar ese número de localidades en un arreglo. La superclase para el registro con la información del estudiante queda casi exactamente igual al que utilizamos como EstudianteBasico, excepto que como ahora la relación de quién sigue a quién va a estar dada por la posición en el arreglo, no necesitamos ya la referencia al siguiente estudiante. Todos los métodos quedan exactamente igual, excepto que todo lo relacionado con el campo siguiente ya no aparece – ver listado 8.1 en la siguiente página.

Código 8.1 Superclase con la información básica de los estudiantes (**InfoEstudiante**)1/3

```
1: import icc1.interfaz.Consola;
2: /**
3:  * Base de datos, a base de listas de registros, que emula la lista
4:  * de un curso de licenciatura. Tiene las opciones normales de una
5:  * base de datos y funciona mediante un Menú.
6:  */
7: public class InfoEstudiante implements Estudiante  {
8:     protected String nombre,
9:         cuenta,
10:        carrera;
11:    /** Constantes simbólicas para identificar campo */
12:    static public final int NOMBRE = 1;
13:    static public final int CUENTA = 2;
14:    static public final int CARRERA = 3;\
15:    /** Constantes para editar mejor los campos.
16:     *  Registran el valor correspondiente de mayor
17:     *  tamaño encontrado para la clase.
18:     */
19:    static protected int maxTamN=0;
20:    static protected int maxTamCta = 0;
21:    static protected int maxTamCarr = 0;
22:    /** Para regresar la posición relativa dentro de la lista */
23:    protected int pos;
24:    /** Se usa para rellenar blancos.  */
25:    static protected final String blancos = replicate(" ",100);
26:
27:    /** Constructor sin parametros. */
28:    public InfoEstudiante() {
29:        nombre = carrera = cuenta = null;
30:    }
31:
32:    /** Constructor que construye el objeto con los datos. */
33:    public InfoEstudiante(String nmbre, String cnta,
34:                          String crrera) {
35:        nombre = nmbre.trim();
36:        cuenta = cnta.trim();
37:        carrera = crrera.trim();
38:        maxTamN = Math.max(nmbre.length(),maxTamN);
39:        maxTamCta = Math.max(cuenta.length(),maxTamCta);
40:        maxTamCarr = Math.max(carrera.length(), maxTamCarr);
41:    }
42:
43:    /** Regresa el valor del atributo nombre. */
44:    public String getNombre()  {
45:        return this.nombre;
46:    }
```

Código 8.1 Superclase con la información básica de los estudiantes (**InfoEstudiante**)2/3

```
47:     /**
48:      * Establece el valor del atributo nombre.
49:      * @param argNombre, cadena.
50:      */
51:     public void setNombre(String argNombre) {
52:         this.nombre = argNombre;
53:     }
54:
55:     /**
56:      * Regresa el valor del atributo cuenta.
57:      */
58:     public String getCuenta()  {
59:         return this.cuenta;
60:     }
61:
62:     /** Establece el valor del atributo cuenta. */
63:     public void setCuenta(String argCuenta) {
64:         this.cuenta = argCuenta;
65:     }
66:
67:     /** Regresa el valor del atributo carrera. */
68:     public String getCarrera()  {
69:         return this.carrera;
70:     }
71:
72:     /** Establece el valor del atributo carrera. */
73:     public void setCarrera(String argCarrera) {
74:         this.carrera = argCarrera;
75:     }
76:
77:     /**
78:      * Regresa la posición relativa de ese registro en
79:      * la lista.
80:      * @return La posición relativa.
81:      */
82:     public int getPos()  {
83:          return pos;
84:     }
85:
86:     /**
87:      * Registra la posición relativa del registro.
88:      * @param pos La posición relativa.
89:      */
90:      public setPos(int pos)  {
91:          this.pos = pos;
92:      }
```

Código 8.1 Superclase con la información básica de los estudiantes (**InfoEstudiante**)3/3

```
93:     /** Regresa el campo indicado del registro dado.
94:      * @param cual Selecciona campo en el registro.
95:      * @return el valor de ese campo en el registro.
96:      */
97:     public String getCampo(int cual) {
98:         String cadena;
99:         switch (cual) {
100:        case NOMBRE:
101:            cadena = getNombre().trim().toLowerCase();
102:            break;
103:        case CUENTA:
104:            cadena = getCuenta().trim().toLowerCase();
105:            break;
106:        case CARRERA:
107:            cadena = getCarrera().trim().toLowerCase();
108:            break;
109:        default:
110:            cadena = "Campo no existente";
111:            break;
112:        } // end of switch (cual)
113:        return cadena;
114:    }
115:
116:    /** Arma una cadena con el contenido del registro.
117:     * @return el contenido del registro.
118:     */
119:    public String armaRegistro() {
120:        return (nombre.trim() + blancos).substring(0,maxTamN + 1)
121:           + "\t"
122:           + (cuenta.trim() + blancos).substring(0,maxTamCta + 1)
123:           + "\t"
124:           + (carrera.trim() + blancos).substring(0,maxTamCarr + 1)
125:           + "\t";
126:    }
127:    /** Llena un registro con los valores de los argumentos.
128:     * @param nmbre Cadena con el nombre.
129:     * @param cnta  Cadena con el n'umero de cuenta.
130:     * @param crrera Cadena con la carrera.
131:     */
132:    public void setRegistro(String nmbre, String cnta,
133:                            String crrera) {
134:        nombre = nmbre.trim();
135:        cuenta = cnta.trim();
136:        carrera = crrera.trim();
137:    }
138: }
```

Como se puede observar en el listado 8.1 en la página 240, todo quedó prácticamente igual, excepto que quitamos todo lo relacionado con la referencia al siguiente. En su lugar agregamos un campo para que el registro "se identifique" a sí mismo en cuanto a la posición que ocupa en el arreglo, pos y los métodos correspondientes. Este campo se verá actualizado cuando se entregue la referencia del registro sin especificar su posición.

La clase así definida puede ser usada, por ejemplo, para cuando queramos una lista de estudiantes que tengan esta información básica incluida. Un posible ejemplo se muestra en el listado 8.2. Esta jerarquía se puede extender tanto como queramos. Si pensamos en estudiantes para listas usamos InfoEstudiante para heredar, agregando simplemente campos necesarios para mantener la lista, como en el caso EstudianteLista del Listado 8.2.

Código 8.2 Extendiendo la clase **InfoEstudiante** (**EstudianteLista**)

```
1: public class EstudianteLista extends InfoEstudiante {
2:   /** Referencia al siguiente de la lista */
3:     protected EstudianteLista siguiente;\
4:   /** Constructor sin paramteros. */
5:     public EstudianteLista() {
6:     }
7:   /** Constructor a partir de una clase que implemente a * Estudiante.
8:    */
9:     public EstudianteLista(Estudiante est) {
10:        nombre = est.getNombre();
11:        cuenta = est.getCuenta();
12:        carrera = est.getCarrera();
13:    }
14:  /** Constructor con los datos para los atributos.
15:   */
16:    public EstudianteLista(String nmbre, String cta, String carr) {
17:        super(nmbre, cta, carr);
18:    }
19:    /** Regresa el valor del atributo siguiente. */
20:    public EstudianteLista getSiguiente()  {
21:        return this.siguiente;
22:    }
23:    /** Establece el valor del atributo siguiente. */
24:    public void setSiguiente(EstudianteLista argSiguiente) {
25:        this.siguiente = argSiguiente;
26:    }
27: }
```

Si en cambio queremos estudiantes para arreglos, usamos directamente In-

foEstudiante. Por ejemplo, si pensamos en registros para estudiantes de un curso podemos usar la definición de EstudianteCalifs para armar un curso – ver Listado 8.3.

Código 8.3 Extendiendo la clase **InfoEstudiante** con calificaciones (**EstudianteCalifs**) 1/3

```
1: public class EstudianteCalifs extends InfoEstudiante {
2:     /** Para almacenar las calificaciones del estudiante */
3:     protected float[] califs;
4:     /** Valor por omisión para el numero de calificaciones. */
5:     protected final int MAX_CALIFS = 10;
6:     /** Constructor que establece el numero de calificaciones
7:      *  como el valor por omision.
8:      */
9:     public EstudianteCalifs() {
10:         califs = new float[MAX_CALIFS];
11:     }
12:     /** Constructor con valores para los atributos y numero de
13:      *  calificaciones.
14:      */
15:     public EstudianteCalifs(int cuantas, String nmbre, String cta,
16:                             String crrera) {
17:         super(nmbre, cta, crrera);
18:         califs = new float[cuantas];
19:     }
20:     /** Construye un objeto nuevo a partir de uno en la interfaz
21:      *  en la raiz de la estructura jerarquica de clases.
22:      */
23:     public EstudianteCalifs(Estudiante est, int numCalifs) {
24:         nombre = est.getNombre();
25:         carrera = est.getCarrera();
26:         cuenta = est.getCuenta();
27:         califs = new float[numCalifs];
28:     }
29:     /** Regresa el valor del atributo califs. */
30:     public float[] getCalifs()  {
31:         return this.califs;
32:     }
33:     /** Establece el valor del atributo califs. */
34:     public void setCalifs(float[] argCalifs) {
35:         this.califs = argCalifs;
36:     }
37:     /** Regresa el valor del atributo MAX_CALIFS. */
38:     public int getMAX_CALIFS()  {
39:         return this.MAX_CALIFS;
40:     }
```

Código 8.3 Extensión de la clase **InfoEstudiante** con calificaciones (**EstudianteCalifs**)2/3

```
41:     /** Coloca la calficacion en la posicion indicada.
42:      * @param calif La calificaci'on deseada.
43:      * @param donde La posici'on deseada.
44:      */
45:     public boolean setCalif(float calif, int donde) {
46:         /* Asegurarse que la posicion este en rangos */
47:         if (donde < 0 || donde >= califs.length) {
48:             return false;
49:         } // end of if (donde < 0 !! donde >= califs.length)
50:         /* Asegurarse que la calificacion este en rangos */
51:         calif = Math.min(10.00f, calif);
52:         calif = Math.max(0f, calif);
53:         /* Registrar la calificacion */
54:         califs[donde] = calif;
55:         return true;
56:     }
57:     /** Regresa la calficacion en la posicion indicada.
58:      * @param donde La posicion deseada.
59:      */
60:     public float getCalif(int donde) {
61:         /* Asegurarse que la posicion este en rangos */
62:         if (donde < 0 || donde >= califs.length) {
63:             return 0;
64:         } // end of if (donde < 0 !! donde >= califs.length)
65:         return califs[donde];
66:     }
67:    /** Regresa una cadena con el registro armado.
68:     */
69:     public String armaRegistro() {
70:         String linea = super.armaRegistro();
71:         linea += muestraCalifs(califs.length);
72:         return linea;
73:     }
74:     /** Obtiene el promedio de calificaciones.
75:      * @param el numero de calificaciones a considerar.
76:      * @return el promedio como un float.
77:      */
78:     public float calcPromedio(int cuantos) {
79:         float suma = 0.0f;
80:         int maxI = Math.min(califs.length, cuantos);
81:         for (int i = 0; i < maxI; i++) {
82:             suma += califs[i];
83:         } // end of for (int i = 0; ...
84:         return suma / maxI;
85:     }
```

Código 8.3 Extensión de la clase **InfoEstudiante** con calificaciones (**EstudianteCalifs**)3/3

```
86:     /** Regresa una cadena con las calificaciones bien organizadas.
87:      * @param el numero de califiaciones a considerar.
88:      * @return Una cadena con las calificiones organizadas.
89:      */
90:     public String muestraCalifs(int cuantas) {
91:         int maxI = Math.max(califs.length, cuantas);
92:         String armada = "";
93:         for (int i = 0; i < maxI; i++) {
94:             int parteEntera = Math.round(califs[i] - 0.5f);
95:             int parteFracc = Math.round(califs[i] * 100) %100;
96:             String edita;
97:             if (parteEntera > 9) {
98:                 edita = parteEntera + ".";
99:
100:            } // end of else
101:            else {
102:                if (parteEntera == 0) {
103:                    edita = "0.";
104:                } // end of if (califs[i] == 0)
105:                else {
106:                    edita = "␣" + parteEntera + ".";
107:                } // end of else
108:
109:            } // end of else
110:            if (parteFracc > 9) {
111:                edita += parteFracc;
112:            } // end of if (parteFracc > 9)
113:            else {
114:                if (parteFracc == 0) {
115:                    edita += "00";
116:                } // end of if (parteFracc == 0)
117:                else {
118:                    edita += "0" + parteFracc;
119:                } // end of else
120:            } // end of else
121:            armada += ((i %5 == 0 && i > 0)
122:                       ? "\n"
123:                       + replicate("␣",maxTamN +1) + "\t"
124:                       + replicate("␣", maxTamCta + 1) +"\t"
125:                       + replicate("␣", maxTamCarr + 1) +"\t\t"
126:                       : "\t")
127:                    + edita;
128:        } // end of for (int i = 0; i < maxI; i++)
129:        return armada;
130:    }
131: }
```

Queremos cambiar nuestra estructura de datos de una lista a un arreglo. Estas dos estructuras de datos tienen semejanzas y diferencias. Veamos primero las semejanzas:

- En ambas estructuras existe una noción de orden entre los elementos de la estructura: podemos determinar cuál elemento va antes y cuál después. Decimos entonces que ambas estructuras son *lineales* porque podemos "formar" a los elementos en una línea. En el caso de los arreglos el orden está dado por el índice, y en el caso de las listas está dado por la posición relativa entre los elementos.
- Todos los elementos de una lista o de un arreglo son del mismo tipo. Decimos entonces que ambas estructuras son *homogéneas*.

En cuanto a las diferencias, mencionamos las siguientes:

- Las listas pueden cambiar de tamaño durante la ejecución, mientras que los arreglos, una vez definido su tamaño, éste ya no puede cambiar. Las listas son estructuras *dinámicas* mientras que los arreglos son estructuras *estáticas*.
- El acceso al elemento de una lista se lleva a cabo recorriendo cada uno de los elementos que están antes que el que buscamos; esto es, es un acceso *secuencial*; el acceso a un elemento de un arreglo es mediante un índice, esto es acceso *directo*.

Dependiendo de qué tipo de datos tengamos podremos elegir entre listas o arreglos para nuestras estructuras de datos. Esta decisión deberá estar, de alguna manera, justificada.

Para construir la clase que maneja el arreglo, lo primero es que en lugar de una cabeza de lista deberemos tener el arreglo, definido de un cierto tamaño, que corresponderá al máximo número de elementos que esperamos. Nuestros algoritmos son exactamente igual, excepto que interpretamos de distinta manera "toma el siguiente", o "colócate al principio". En el caso de que los registros estén en un arreglo "colócate al principio" se interpreta como "inicializa un índice en 0"; y "toma el siguiente" se interpreta como "incrementa en uno a la variable empleada como índice". Con esta representación es fácil obtener el anterior, ya que únicamente se decrementa el índice en 1, si es que existe anterior. Además, en todo momento tenemos que tener cuidado de no tratar de tomar elementos más allá del fin del arreglo. Veamos cómo queda con estos cambios.[1] Dado un arreglo es fácil saber cuál es el tamaño del arreglo, mediante el atributo `length`, pero no es igual de fácil saber el número de elementos que se encuentran *realmente* en el arreglo. Por lo tanto, es conveniente ir contando los registros que se agregan y los

[1] Se recomienda referirse a los diagramas de Warnier-Orr donde se dieron los algoritmos en su momento.

que se quitan, para que en todo momento se tenga claro el número de elementos que tenemos en un arreglo. En el listado 8.4 podemos ver lo relacionado con el cambio de estructura de datos de una lista a un arreglo.

Código 8.4 Base de datos implementada en un arreglo (**CursoEnVector**)1/2

```
1: import icc1.interfaz.Consola;
2: public class CursoEnVector implements Curso {
3:     /** En este curso se van a registrar NUM_CALIFS
4:       * calificaciones. */
5:      private  int NUM_CALIFS = 10;
6:     /** Numero por omision de registros. */
7:      private final int MAXREG = 10;
8:
9:     /** El arreglo que almacena a la lista. */
10:     private EstudianteVector[] lista;
11:    /** Numero de grupo. */
12:     private String grupo;
13:    /** Numero de estudiantes inscritos. */
14:     private int numRegs = 0;
15:
16:    /** Constructor que establece el numero de grupo y cuantas
17:     *  calificaciones.
18:     */
19:     public CursoEnVector( String gr, int numCalifs) {
20:         NUM_CALIFS = numCalifs;
21:         grupo = gr;
22:         numRegs = 0;
23:         lista = new EstudianteVector[MAXREG];
24:     }
25:
26:    /** Constructor que establece el numero de grupo. cuantas
27:     *  calificaciones y cual es el maximo de estudiantes.
28:     */
29:     public CursoEnVector( String gr, int cuantos, int numCalifs) {
30:         NUM_CALIFS = numCalifs;
31:         grupo = gr;
32:         numRegs = 0;
33:         lista = new EstudianteVector[cuantos];
34:     }
35:
36:    /** Regresa el valor del atributo numRegs. */
37:     public int getNumRegs() {
38:         return numRegs;
39:     }
```

Código 8.4 Base de datos implementada en un arreglo (**CursoEnVector**)2/2

```
40:     /** Constructor que establece el numero de grupo y una lista
41:      *  inicial de estudiantres inscritos.
42:      */
43:     public CursoEnVector(Estudiante[] iniciales, String gr) {
44:         grupo = gr;
45:         if (iniciales.length > MAXREG) {
46:             lista = new EstudianteVector[Math.max(2
47:                         * iniciales.length, 2 * MAXREG)];
48:         } // end of if (iniciales.length)
49:         else {
50:             lista = new EstudianteVector[MAXREG];
51:         } // end of else
52:         for (int i = 0; i < iniciales.length; i++) {
53:             lista[i] = new EstudianteVector(iniciales[i],
54:                                                 NUM_CALIFS);
55:             numRegs ++;
56:         } // end of for (int i = 0; i < iniciales.length; i++)
57:     }
```

Hay algunas operaciones básicas que vamos a necesitar al trabajar con arreglos. Por ejemplo, para agregar a un elemento en medio de los elementos del arreglo (o al principio) necesitamos recorrer a la derecha a todos los elementos que se encuentren a partir de la posición que queremos que ocupe el nuevo elemento. Esto lo tendremos que hacer si queremos agregar a los elementos y mantenerlos en orden conforme los vamos agregando.

Similarmente, si queremos eliminar a alguno de los registros del arreglo, tenemos que recorrer a los que estén más allá del espacio que se desocupa para que se ocupe el lugar desocupado. En ambos casos tenemos que recorrer a los elementos uno por uno y deberemos tener mucho cuidado en el orden en que recorramos a los elementos. Al recorrer a la derecha deberemos recorrer desde el final del arreglo hacia la primera posición que se desea mover. Si no se hace en este orden se tendrá como resultado el valor del primer registro que se desea mover copiado a todos los registros a su derecha. El método para recorrer hacia la derecha se encuentra en el listado 8.5 en la siguiente página. El método nos tiene que regresar si pudo o no pudo recorrer a los elementos. En el caso de que no haya suficiente lugar a la derecha, nos responderá falso, y nos responderá verdadero si es que pudo recorrer.

Si deseamos regresar un lugar a la izquierda, el procedimiento es similar, excepto que tenemos que mover desde el primero hacia el último, para no acabar con una repetición de lo mismo.

Código 8.5 Corrimiento de registros hacia la derecha e izquierda (**CursoEnVector**)

```
58:     /** Recorrer  registros a partir del que está en la
59:      * posición desde cuantos lugares a la derecha.
60:      * @param int desde Posición del primer registro a recorrer.
61:      * @param int cuantos Número de "espacios" a recorrer.
62:      * @returns boolean Si pudo o no hacerlo.
63:      */
64:     /** Hace lugar parta insertar a un elemento en el lugar que
65:      *  le corresponde. */
66:     private boolean recorre(int desde, int cuantos) {
67:         if (numRegs + cuantos >= lista.length)
68:             return false;
69:         for (int i = numRegs - 1; i >= desde; i--) {
70:             lista[i + cuantos] = lista[i];
71:             lista[i + cuantos].setPos(i + cuantos);
72:             lista[i] = null;
73:         } // end of for (int i = numRegs - 1; i >= desde; i--)
74:         return true;
75:     }
76:     /** Se recorren registros a la izquierda para ocupar un espacio
77:      * @param int desde la primera posición que se recorre hacia la
78:      * izquierda
79:      */
80:     private boolean regresa(int desde, int cuantos) {
81:         if (((desde - cuantos) < 0) || (desde > numRegs -1)) {
82:             return false;
83:         } // end of if ((desde - cuantos) < 0)
84:
85:         for (int i = desde + 1; i < numRegs - 1; i++) {
86:             lista[i-1] = lista[i];
87:             lista[i-1].setPos(i-1);
88:         } // end of for (int i = 0; i < numRegs - 1; i++)
89:         numRegs -= cuantos;
90:         return true;
91:     }
```

El método que regresaba la referencia de la lista – del primer elemento de la lista – ahora debe regresar la referencia al arreglo completo. Queda como se muestra en el listado 8.6 en la página opuesta, junto con los métodos que ponen y regresan el número de grupo, que no cambian.

Para saber el número de registros en el arreglo ya no nos sirve "revisarlo" y ver cuántas referencias distintas de null tiene. Por ejemplo, si el arreglo en vez de objetos tiene números, y un valor válido para los números es el 0, no habría forma de distinguir entre un lugar ocupado por un 0 o un lugar que no estuviera ocupado. Por ello es conveniente ir contando los registros que se van agregando e ir descon-

tando los que se quitan, con los registros activos ocupando posiciones consecutivas

Código 8.6 Métodos de acceso y manipulación (**CursoEnVector**)

```
92:    /** Regresa la lista de estudiantes.
93:     */
94:     public Estudiante[] getLista()  {
95:       return lista;
96:     }
97:    /** Proporciona el número de grupo.
98:     */
99:     public String getGrupo()  {
100:      return grupo;
101:    }
102:   /** Modifica el número de grupo.
103:    */
104:    public void setGrupo(String gr){
105:      grupo = gr;
106:    }
```

en el arreglo. Por ello, ya no se calcula el número de elementos en el arreglo, sino que simplemente se regresa este valor. El método se encuentra en el listado 8.7.

Código 8.7 Método de acceso al número de registros (**CursoEnVector**)

```
107:   /** Regresa el número de registros en la lista.
108:    */
109:    public int  getNumRegs()  {
110:       return numRegs;
111:    }
```

Tenemos tres maneras de agregar registros al arreglo. La primera de ellas, la más fácil, es agregando al final de los registros, ya que esto no implica mover a nadie, sino simplemente verificar cuál es el siguiente lugar a ocupar. De manera similar a que cuando el número de elementos en un arreglo es $\boldsymbol{n}$ los índices van del 0 al $\boldsymbol{n-1}$, si numRegs vale $\boldsymbol{k}$ quiere decir que los $\boldsymbol{k}$ registros ocupan las posiciones 0 a $\boldsymbol{k-1}$, por lo que la siguiente posición a ocupar es, precisamente, $\boldsymbol{k}$. En general, numRegs contiene la siguiente posición a ocuparse. Por lo que si se agrega al final, lo único que hay que hacer es ocupar el lugar marcado por numRegs, incrementando a este último.

Si vamos a agregar los registros siempre al principio del arreglo, lo que tenemos que hacer es recorrer todos los registros un lugar a la derecha para "desocupar"

el primer lugar y colocar ahí el registro.

Por último, si se desea mantener ordenados los registros con un orden lexicográfico, primero tenemos que localizar el lugar que le toca. Una vez hecho esto se recorren todos los registros a partir de ahí un lugar a la derecha, y se coloca en el lugar desocupado al nuevo registro.

En los tres casos, antes de agregar algún registro deberemos verificar que todavía hay lugar en el arreglo, ya que el arreglo tiene una capacidad fija dada en el momento en que se crea. Como no siempre vamos a poder agregar registros (algo que no sucedía cuando teníamos una lista), todos estos métodos tienen que decirnos de regreso si pudieron o no.

Para localizar el lugar que le toca a un registro nuevo lo vamos comparando con los registros en el arreglo, hasta que encontremos el primero "mayor" que él. Como el orden está dado por el nombre, que es una cadena, tenemos que comparar cadenas. El método compareTo de la clase String nos sirve para saber la relación entre dos cadenas, de la siguiente forma:

$$\text{s1.compareTo(String s2)} \begin{cases} \mathbf{-1} & \text{Si s1} < \text{s2} \\ \mathbf{0} & \text{Si s1} == \text{s2} \\ \mathbf{1} & \text{Si s1} > \text{s2} \end{cases}$$

La programación de estos tres métodos se puede apreciar en el listado 8.8.

Código 8.8 Agregando registros a la base de datos (**CursoEnVector**)1/2

```
112:    /**
113:     * Agrega un registro al final de la lista.
114:     *
115:     * @param InfoEstudiante nuevo El registro a agregar.
116:     */
117:    public boolean agregaEstFinal(Estudiante nuevo)    {
118:        int actual;
119:        if (numRegs >= lista.length)
120:            return false;
121:        lista[numRegs++] = nuevo;
122:        return true;
123:    }
```

Código 8.8 Agregando registros a la base de datos (**CursoEnVector**)2/2

```
124:     /**
125:      * Agrega un registro al principio de la lista.
126:      *
127:      * @param InfoEstudiante nuevo A quien se va a agregar
128:      */
129:     public boolean agregaEst(InfoEstudiante nuevo)    {
130:        // Recorre a todos los estudiantes un lugar a la derecha
131:         if (! recorre(0,1))  {
132:             return false;
133:         }
134:         lista[0] = nuevo;
135:         numRegs++;
136:         return true;
137:     }
138:     /**
139:      * Agrega un registro donde la toca para mantener la lista
140:      * ordenada.
141:      *
142:      * @param InfoEstudiante nuevo A quien se va a agregar
143:      * @returns boolean Si pudo o no agregar
144:      */
145:     public boolean agregaEstOrden(InfoEstudiante nuevo)    {
146:         if (numRegs == lista.length)  {
147:              return false;
148:         }
149:         int actual = 0;
150:         while (actual < numRegs && lista[actual].daNombre()
151:                 .compareTo(nuevo.daNombre()) <= 0)
152:              actual++;
153:         if (actual == numRegs) { // Entra al final
154:             lista[actual] = nuevo;
155:             numRegs++;
156:             return true;
157:         }
158:         if (!recorre(actual,1))    {
159:            return false;
160:         }
161:         lista[actual] = nuevo;
162:         numRegs++;
163:         return true;
164:     }
```

Cuando deseamos agregar un registro de tal manera de mantener la lista en orden, debemos, como ya dijimos, encontrarle el lugar que le toca, entre un registro lexicográficamente menor o igual a él y el primero mayor que él. Esto lo hacemos

en las líneas 149: – 152: del listado 8.8 en la página 252. Nos colocamos al principio del vector, poniendo el índice que vamos a usar para recorrerlo en 0 – línea 149:. A continuación, mientras no se nos acaben los registros del arreglo y estemos viendo registros lexicográficamente menores al que buscamos – condicionales en líneas 150: y 151: – incrementamos el índice, esto es, pasamos al siguiente.

Podemos salir de la iteración porque se deje de cumplir cualquiera de las dos condiciones: que ya no haya elementos en el arreglo o que ya estemos entre uno menor o igual y uno mayor. En el primer caso habremos salido porque el índice llegó al número de registros almacenados – actual == numRegs – en cuyo caso simplemente colocamos al nuevo registro en el primer lugar sin ocupar del arreglo. No hay peligro en esto pues al entrar al método verificamos que todavía hubiera lugares disponibles.

En el caso de que haya encontrado un lugar entre dos elementos del arreglo, tenemos que recorrer a todos los que son mayores que él para hacer lugar. Esto se hace en la línea 158:, donde de paso preguntamos si lo pudimos hacer – seguramente sí porque ya habíamos verificado que hubiera lugar. Una vez recorridos los elementos del arreglo, colocamos el nuevo elemento en el lugar que se desocupó gracias al corrimiento, y avisamos que todo estuvo bien – líneas 161: y 162: – no sin antes incrementar el contador de registros.

Código 8.9 Quitando a un estudiante de la base de datos **(CursoEnVector)**

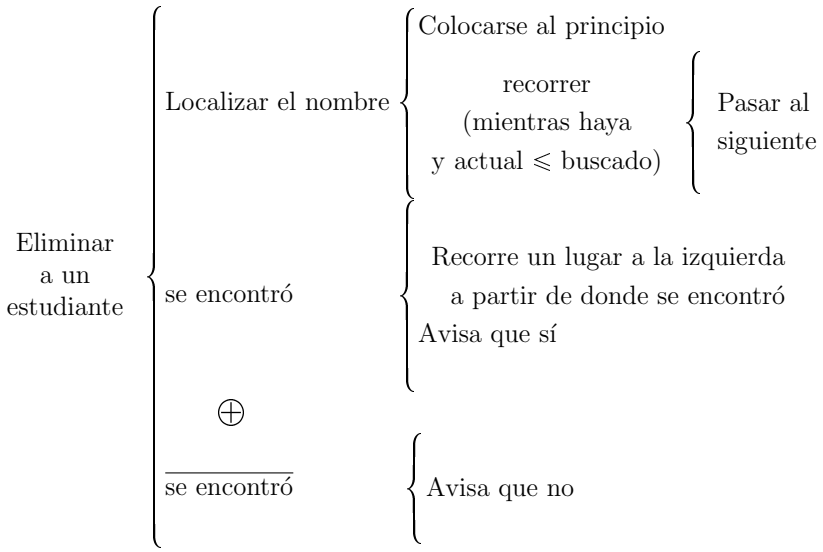
```
165:     /** Quita el registro solicitado (por nombre) de la lista.
166:      * @param nombre El estudiante que se desea eliminar.
167:      */
168:     public boolean quitaEst(String nombre)     {
169:         int actual;
170:         nombre=nombre.trim().toLowerCase();
171:         if (numRegs == 0)
172:             return false; // Está vacía la lista: no se pudo
173:         actual = 0;       // El primero de la lista
174:         while (actual < numRegs &&
175:                         !(lista[actual].daNombre().toLowerCase().
176:                         equals(nombre.toLowerCase()))) {
177:               actual++;
178:         }
179:         if (actual == numRegs)
180:             return false;     // No se encontró
181:         regresa(actual,1);
182:         return true;
183:     }
```

También el método que quita a un estudiante va a cambiar. El método hace

lo siguiente: localiza al registro que contenga al nombre completo, que llega como parámetro – líneas 174: a 178: del listado 8.9 en la página opuesta.

Una vez que encontró el registro en el que está ese nombre, se procede a "desaparecerlo", recorriendo a los registros que están después que él un lugar a la izquierda, encimándose en el registro que se está quitando; esto se hace con la llamada a regresa(actual,1) en la línea 181: del listado 8.9 en la página opuesta. Al terminar de recorrer a los registros hacia la izquierda, se decrementa el contador de registros numRegs. La ubicación de este decremento en el método regresa – línea 85: del listado 8.5 en la página 250 – se justifica bajo la óptica de que numRegs indica la posición del primer registro vacío; otra opción hubiera sido poner el decremento en quitaEst, que estaría justificado bajo la óptica de que numRegs cuenta el número de estudiantes en la base de datos. Como numRegs juega ambos papeles la respuesta a dónde poner el decremento no es única. El diagrama de Warnier correspondiente se encuentra en la Figura 8.1.

Figura 8.1 Algoritmo para eliminar a un estudiante



El método que busca una subcadena en alguno de los campos en el arreglo cambia la forma en que nos colocamos al principio: colocarse al principio ahora implica poner al índice que vamos a usar para recorrerlo en 0 – línea 191: del

listado 8.10 – mientras que tomar el siguiente quiere decir incrementar en 1 el índice que se está usando para recorrer el arreglo – línea 194: del listado 8.10.

Código 8.10 Búsqueda de una subcadena en algún campo del arreglo (**CursoEnVector**)

```
184:     /** Busca al registro que contenga a la subcadena.
185:      * @param int cual Cual es el campo que se va a comparar.
186:      * @param String subcad La cadena que se está buscando.
187:      * @returns int El registro deseado o -1.                */
188:     public InfoEstudiante buscaSubcad(int cual, String subcad) {
189:         int actual;
190:         subcad = subcad.trim().toLowerCase();
191:         actual = 0;
192:         while (actual < numRegs && (lista[actual].daCampo(cual).
193:                     indexOf(subcad.toLowerCase())) == -1)
194:             actual++;
195:         if (actual < numRegs)
196:             return lista[actual];
197:         else
198:             return null;
199:     }
```

Otra diferencia es que cuando en un arreglo preguntamos si ya se nos acabaron los elementos, lo que hacemos es preguntar si el índice ya alcanzó al número de registros – línea 195: – y no si la referencia es nula. Por lo tanto, recorremos el arreglo mientras nuestro índice sea menor que el número de registros – el índice sea válido – y no tengamos enfrente – en la posición actual del arreglo – a quien estamos buscando.

Una vez recorrido el arreglo deberemos averiguar si encontramos o no la subcadena. Si el índice llegó a ser el número de registros, entonces no lo encontró. Si no llegó, el entero contenido en el índice corresponde a la posición de la subcadena encontrada. El método procede, entonces, a regresar el registro que contiene a la subcadena.

Una pregunta natural es ¿por qué no regresamos simplemente el índice en el que se encuentra el registro? La respuesta es muy sencilla. El índice tiene sentido como mecanismo de acceso al arreglo. Sin embargo, el arreglo es un dato privado de VectorCurso, por lo que desde CursoMenu, y desde cualquier otra clase, no se tiene acceso a él. Entonces, el conocer una posición en el arreglo, desde fuera de VectorCurso, no sólo no nos sirve, sino que va contra los principios del encapsulamiento, en el que los datos son privados en un 99 % de los casos (para hacer un dato público deberá estar sumamente justificado). Adicionalmente, la clase que maneja el menú queda prácticamente idéntica a como estaba para el manejo de

las listas, y esto es algo deseable. De esa manera podemos decir que la clase MenuVector no tiene que saber cómo están implementadas las estructuras de datos o los métodos de la clase VectorLista, sino simplemente saber usarlos y saber que le tiene que pasar como parámetro y qué espera como resultado. Excepto por los métodos que agregan y quitan estudiantes, que los volvimos booleanos para que informen si pudieron o no, todos los demás métodos mantienen la firma que tenían en la implementación con listas ligadas. Vale la pena decir que podríamos modificar los métodos de las listas ligadas a que también contestaran si pudieron o no, excepto que en el caso de las listas ligadas siempre podrían.

Código 8.11 Listar todos los registros de la base de datos (**CursoEnVector**)

```
200:     /**
201:      * Lista todos los registros del Curso.
202:      *
203:      * @param Consola cons dónde escribir.
204:      */
205:     public void listaTodos(Consola cons)    {
206:         int actual;
207:         for (actual = 0; actual < numRegs; actual++)  {
208:             cons.imprimeln(lista[actual].daRegistro());
209:         }
210:         if (actual == 0)  {
211:             cons.imprimeln("No hay registros en la base de datos");
212:         }
213:     }
```

Nos falta revisar nada más dos métodos: el que lista todo el contenido de la base de datos y el que lista solamente los que cazan con cierto criterio. Para el primer método nuevamente se aplica la transformación de que colocarse al principio de la lista implica poner al índice que se va a usar para recorrerla en 0 – línea 207: en el listado 8.11. Nuevamente nos movemos por los registros incrementando el índice en 1, y verificamos al salir de la iteración si encontramos lo que buscábamos o no.

En el caso del método que lista a los que cazan con cierto criterio – listado 8.12 en la siguiente página – nuevamente se recorre el arreglo de la manera que ya vimos, excepto que cada uno que contiene a la subcadena es listado. Para saber si se listó o no a alguno, se cuentan los que se van listando – línea 223: en el listado 8.12 en la siguiente página. Si no se encontró ningún registro que satisficiera las condiciones dadas, se da un mensaje de error manifestándolo.

Código 8.12 Listando los que cumplan con algún criterio (**CursoEnVector**)

```
214:     /**
215:      * Imprime los registros que  cazan con un cierto patrón.
216:      *
217:      * @param Consola cons Dispositivo en el que se va a escribir.
218:      * @param int cual  Con cuál campo se desea comparar.
219:      * @param String subcad Con el que queremos que cace.
220:      */
221:     public void losQueCazanCon(Consola cons, int cual,
222:                                            String subcad)  {
223:         int i = 0;
224:         subcad = subcad.toLowerCase();
225:         int actual;
226:
227:         /** Recorremos buscando el regsitro */
228:         for (actual = 0; actual < numRegs; actual++)  {
229:             if (lista[actual].daCampo(cual).indexOf(subcad) !=
230:                     -1)  {
231:                 i++;
232:                 cons.imprimeln(lista[actual].daRegistro());
233:             }
234:         }
235:
236:         /**  Si no se encontró ningún registro  */
237:         if (i == 0)  {
238:             cons.imprimeln("No se encontró ningún registro " +
239:                                 "que  cazara");
240:         }
241:     }
```

8.2 Mantenimiento del orden con listas ligadas

Tenemos ya una clase que maneja a la base de datos en una lista ligada (ListaCurso). Podemos modificar levemente ese programa para beneficiarnos de la herencia y hacer que Estudiante herede de la clase InfoEstudiante, y de esa manera reutilizar directamente el código que ya tenemos para InfoEstudiante. Todo lo que tenemos que hacer es agregarle los campos que InfoEstudiante no tiene y los métodos de acceso y manipulación para esos campos. La programación de la

clase utilizando herencia se puede observar en el listado 8.13.

Código 8.13 Definición de la clase **Estudiante** para los registros (**Estudiante**) 1/3

```
1: import icc1.interfaz.Consola;
2: /**
3:  * Base de datos, a base de listas de registros, que emula la lista
4:  *  de un curso de licenciatura. Tiene las opciones normales de una
5:  * base de datos y funciona mediante un Menú
6:  */
7: class Estudiante extends InfoEstudiante    {
8:     protected Estudiante siguiente;
9:     protected String clave;
10:    public static final int CLAVE = 4;
11:    /** Constructor sin parámetros. */
12:    public Estudiante()   {
13:        super();
14:        clave = null;
15:        siguiente = null;
16:    }
17:    /**
18:     * Constructor a partir de datos de un estudiante.
19:     * Los campos vienen separados entre sí por comas, mientras
20:     * que los registros vienen separados entre sí por punto
21:     * y coma.
22:     * @param String, String, String, String los valores para
23:     *              cada uno de los campos que se van a llenar.
24:     * @return Estudiante una referencia a una lista
25:     */
26:    public Estudiante(String nmbre, String cnta, String clve,
27:                      String crrera)  {
28:        super(nmbre, cnta, crrera);
29:        clave = clve.trim();
30:        siguiente = null;
31:    }
32:    /**
33:     * Regresa el contenido del campo clave.
34:     */
35:    public String getClave()  {
36:        return clave;
37:    }
38:    /**
39:     * Actualiza el campo clave con el valor que pasa como
40:     * parámetro.
41:     */
42:    public void setClave(String clve)  {
43:       clave = clve;
44:    }
```

Código 8.13 Definición de la clase **Estudiante** para los registros (**Estudiante**)2/3

```
45:     /**
46:      * Regresa el campo que corresponde al siguiente registro
47:      * en la lista
48:      */
49:     public Estudiante getSiguiente()  {
50:         return siguiente;
51:     }
52:     /**
53:      * Regresa el campo seleccionado del registro dado.
54:      * @param int Estudiante  El número del campo y el registro.
55:      * @returns String La cadena solicitada
56:      */
57:     public String getCampo(int cual)  {
58:         String cadena;
59:         switch(cual) {
60:           case InfoEstudiante.NOMBRE:
61:               cadena = getNombre().trim().toLowerCase();
62:               break;
63:           case InfoEstudiante.CUENTA:
64:               cadena = getCuenta().trim().toLowerCase();
65:               break;
66:           case Estudiante.CARRERA:
67:               cadena = getCarrera().trim().toLowerCase();
68:               break;
69:           case Estudiante.CLAVE:
70:               cadena = getClave().trim().toLowerCase();
71:               break;
72:           default:
73:               cadena = "Campo␣no␣existente";
74:           }
75:           return cadena;
76:      }
77:     /**
78:      * Actualiza el campo siguiente con la referencia que se
79:      * la pasa.
80:      * @param sig La referencia a colocar.
81:      */
82:     public void setSiguiente(Estudiante sig)  {
83:        siguiente = sig;      \
84:     }
85:     /**
86:      * Arma una cadena con el contenido de todo el registro.
87:      * @return Una cadena con el registro deseado.
88:      */
89:     public String getRegistro()  {
90:         return super.getRegistro()+"\t"+clave.trim();
91:     }
```

Código 8.13 Definición de la clase **Estudiante** para los registros (**Estudiante**)3/3

```
92:     /**
93:      * Actualiza todo el registro de un jalón.
94:      * @param  String  el nombre,
95:      *                 String cuenta
96:      *                 String carrera
97:      *                 String clave.
98:      */
99:     public void setRegistro(String nmbre, String cnta,
100:                             String clve, String crrera)  {
101:          super.setRegistro(nmbre, cnta, crrera);
102:          clave = clve.trim();
103:    }
104: }
```

Hay que notar que lo que programamos como de acceso privado cuando no tomábamos en consideración la herencia, ahora se convierte en acceso protegido, para poder extender estas clases.

Algunos de los métodos que enunciamos en esta clase son, simplemente, métodos nuevos para los campos nuevos. Tal es el caso de los que tienen que ver con clave y siguiente. El método getCampo se redefine en esta clase, ya que ahora tiene que considerar más posibilidades. También el método getRegistro es una redefinición, aunque usa a la definición de la superclase para que haga lo que correspondía a la superclase.

Los constructores también son interesantes. Cada uno de los constructores, tanto el que tiene parámetros como el que no, llaman al correspondiente constructor de la superclase, para que inicialice los campos que tiene en común con la superclase.

La palabra super se está utilizando de dos maneras distintas. Una de ellas, en el constructor, estamos llamando al constructor de la superclase usando una notación con argumentos. En cambio, en el método getRegistro se usa igual que cualquier otro objeto, con la notación punto. En este segundo caso nos referimos al "super-objeto" de this, a aquél definido por la superclase.

8.2.1. Revisita de la clase ListaCurso

Vimos en el capítulo anterior prácticamente todos los métodos relativos al manejo de listas. Nos faltó únicamente el método que agrega registros a la base de datos, manteniendo el orden. Como en la parte anterior seguiremos teniendo al

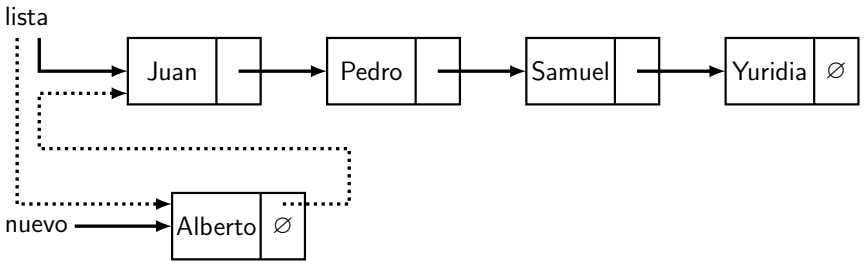
nombre como la llave (*key*) de nuestros registros, y es el que va a definir el orden. Igual podríamos tener el número de cuenta o la carrera.

Para agregar un registro tenemos que distinguir entre tres situaciones distintas:

(a) La lista está vacía, en cuyo caso le toca ser la cabeza de la lista.
(b) Ya hay registros en la lista pero todos van después que el que se está agregando.
(c) Le toca entre dos registros que ya están en la lista, o bien es el último (ambos casos se tratan igual).

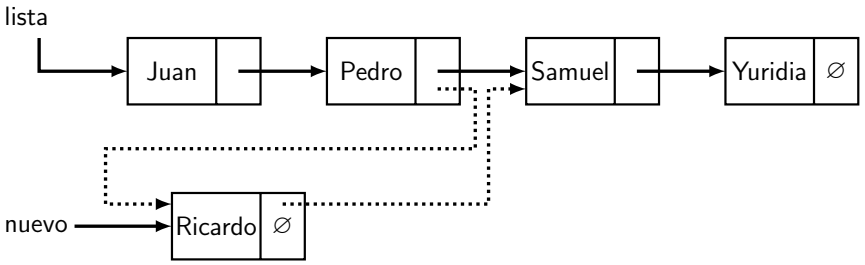
El primer caso es sencillo, ya que simplemente "inauguramos" la lista con el registro que se nos da. El segundo caso no es igual al tercero porque el nuevo registro tiene que quedar en la cabeza de la lista – ver figura 8.2, donde las flechas punteadas son las referencias que deben de quedar. Por lo que la referencia que queda antes es la de la cabeza de la lista y la que queda después es la que antes era la primera.

Figura 8.2 Agregando al principio de la lista



En el caso de que no sea el primero que se agrega, y que no le toque al principio de la lista, se debe recorrer la lista hasta que a su izquierda haya uno menor y a su derecha uno mayor. Esto se logra, simplemente recorriendo la lista y parando cuando se encuentre el final de la lista, o bien el primero que va a tener una llave mayor que el nuevo. Una vez que se encontró el lugar, se modifican las referencias al siguiente para que quede insertado en la lista – ver figura 8.3 en la página opuesta.

Figura 8.3 Agregando en medio de la lista



El diagrama de Warnier que describe este proceso se encuentra en la figura 8.4 en la siguiente página y la programación del método se puede ver en el listado 8.14.

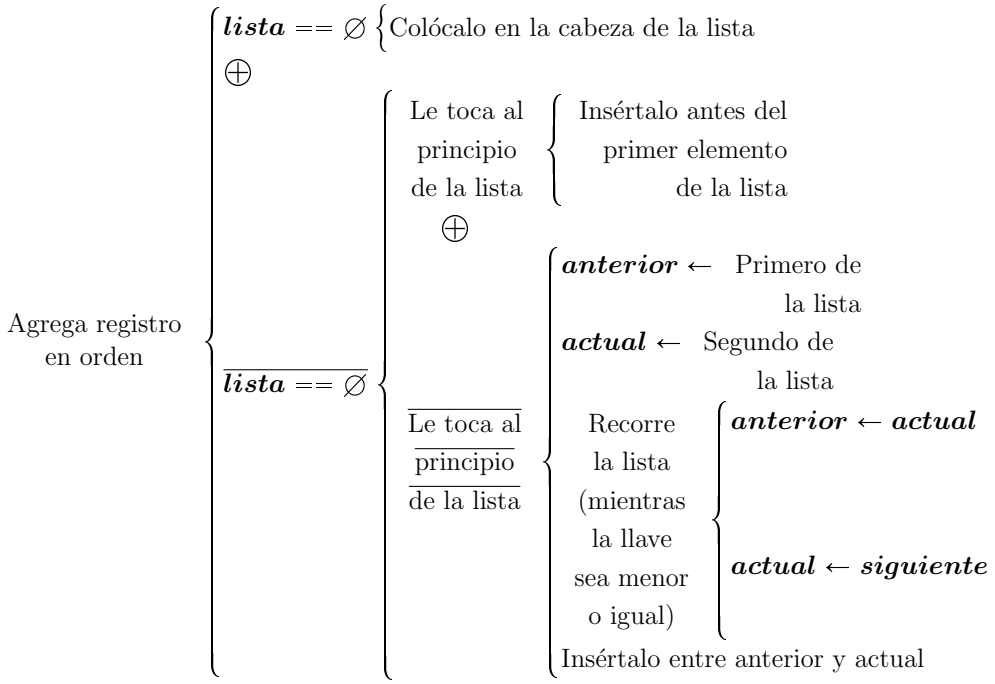
Código 8.14 Agregar un registro manteniendo el orden (**ListaCurso**)1/2

```
105:    /** Agrega un estudiante en orden en la lista.
106:     * @param Estudiante nuevo El registro a agregar
107:     * @returns boolean Si pudo o no hacerlo
108:     */
109:    public boolean agregaEstOrden(Estudiante nuevo)    {
110:        String scompara = nuevo.daNombre().trim().toLowerCase();
111:        if (lista == null)        // Es el primero que se mete
112:            lista = nuevo;
113:            numRegs++;
114:            return true;
115:        }
116:        if (lista.daNombre().trim().toLowerCase()
117:                                .compareTo(scompara) > 0)  {
118:         // Le toca el primer lugar de la lista, pero
119:         // no es el único
120:            nuevo.ponSiguiente(lista);
121:            lista = nuev;
122:            numRegs++;
123:            return true;
124:        }
125:        Estudiante actual = lista.daSiguiente(),
126:        anterior = lista;
127:        while ( actual != null && actual.daNombre().trim()
128:                  .toLowerCase().compareTo(scompara) <= 0) {
129:            anterior = actual;
130:            actual = actual.daSiguiente();
131:        }
```

Código 8.14 Agregar un registro manteniendo el orden (**ListaCurso**)2/2

```
132:                    // Si actual == null le toca al final de la lista
133:            nuevo.ponSiguiente(actual);
134:            anterior.ponSiguiente(nuevo);
135:            numRegs++;
136:            return true;
137:        }
```

Figura 8.4 Agregando un registro en orden



Debemos insistir en que se debe tener cuidado el orden en que se cambian las referencias. Las referencias que vamos a modificar son la de nuevo y la siguiente en anterior, que es la misma que tenemos almacenada en actual. Por ello, el orden para cambiar las referencias podría haber sido

```
138:            anterior.ponSiguiente(nuevo);
139:            nuevo.ponSiguiente(actual);
```

Lo que debe quedar claro es que una vez modificado anterior.siguiente, esta referencia ya no se puede usar para colocarla en nuevo.siguiente.

8.3 *Ordenamiento usando árboles

Mantener una lista de cadenas ordenada alfabéticamente es costoso si la lista la tenemos ordenada en un arreglo o en una lista ligada. El costo se expresa en términos de operaciones que debemos realizar para llevar a cabo una inserción, remoción, modificación, etc. de la lista. Por ejemplo, cuando tenemos la lista en un arreglo, agregar a alguien quiere decir:

- Encontrarle lugar
- Recorrer a todos los que están después de él un lugar, para hacerle lugar al nuevo.
- Copiar el nuevo a ese lugar.

Similarmente, para quitar a alguien de la lista, debemos recorrer a todos los que se encuentren después de él para mantener a las cadenas en espacio contiguo.

Para el caso de listas ligadas, la remoción e inserción de registros es un poco más económica, pero para encontrar a alguien en promedio tenemos que hacer $\boldsymbol{n/2}$ comparaciones, donde $\boldsymbol{n}$ es el tamaño de la lista.

Deseamos mantener la lista ordenada, pero queremos bajar los tiempos de localización de un registro, asumiendo que ésta es la operación más frecuente que deseamos hacer. Un método bastante eficiente de almacenar información que se debe mantener ordenada (con respecto a algún criterio) es el de utilizar una estructura de datos llamada árbol binario.

Empecemos por definir lo que es un árbol, para pasar después a ver, específicamente, a los árboles binarios. Un árbol es una estructura de datos dinámica, no lineal, homogénea. Está compuesta por nodos y los nodos están relacionados entre sí de la siguiente manera:

- Cada nodo tiene a un único nodo apuntando a él. A éste se le llama el nodo *padre*.
- Cada nodo apunta a cero o más nodos. A estos nodos se les llama los *hijos*.
- A los nodos que no tienen hijos se les llama *hojas*.
- Existe un único nodo privilegiado, llamado la *raíz* del árbol, que no tiene padre: corresponde al punto de entrada de la estructura.
- A cada nodo le corresponde un *nivel* en el árbol, y se calcula sumándole 1 al nivel de su padre. La raíz tiene nivel 1.

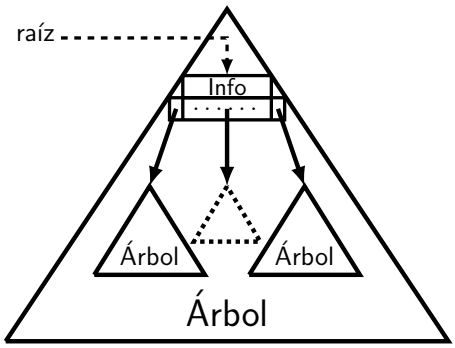
- La *profundidad* del árbol es el máximo de los niveles de sus nodos.
- Cada nodo del árbol va a tener un campo para la información que deseamos organizar, y $\boldsymbol{n}$ referencias, donde $\boldsymbol{n}$ es el número máximo de hijos que puede tener un nodo.

Un árbol es *n-ario* si el máximo número de hijos que puede tener es $\boldsymbol{n}$. Es *binario*, si el máximo número de hijos es 2; *terciario* para tres hijos, y así sucesivamente. Podemos representar árboles con un número arbitrario de hijos para cada nodo, pero dejaremos ese tema para cursos posteriores. También podemos definir los árboles recursivamente:

$$\text{Un } \textit{árbol n-ario} \text{ es } \begin{cases} \text{Un nodo que no tiene hijos} \\ \bigoplus \\ \text{Un nodo que tiene como hijos a } \boldsymbol{n} \text{ árboles} \end{cases}$$

A esta definición le corresponde el esquema en la figura 8.5.

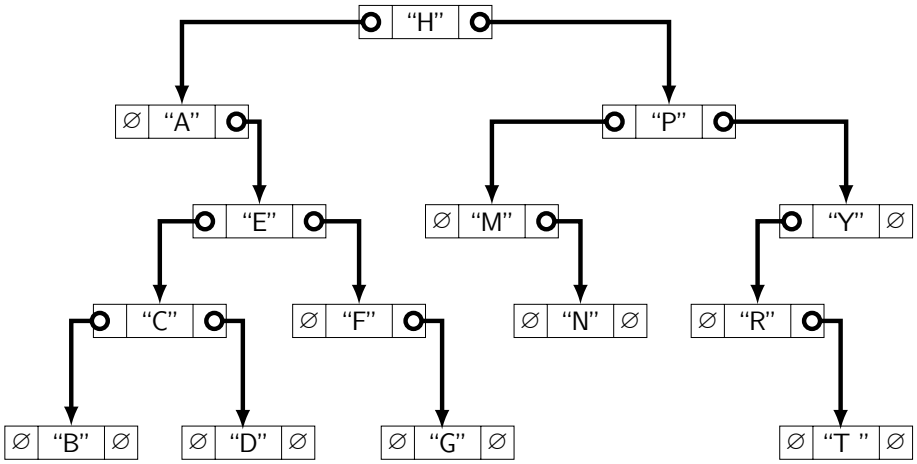
Figura 8.5 Definición recursiva de un árbol



En estos momentos revisaremos únicamente a los árboles binarios, por ser éstos un mecanismo ideal para organizar a un conjunto de datos de manera ordenada. Un árbol binario, entonces, es un árbol donde el máximo número de hijos para cada nodo es dos. Si lo utilizamos para organizar cadenas, podemos pensar que dado un árbol, cada nodo contiene una cierta cadena. Todas las cadenas que se encuentran en el subárbol izquierdo son menores a la cadena que se encuentra en la raíz. Todas las cadenas que se encuentran en el subárbol derecho, son mayores

a la que se encuentra en la raíz. Un árbol binario bien organizado se muestra en la figura 8.6.

Figura 8.6 Árbol binario bien organizado



Cuando todos los nodos, excepto por las hojas del último nivel, tienen exactamente el mismo número de hijos, decimos que el árbol está *completo*. Si la profundidad del subárbol izquierdo es la misma que la del subárbol derecho, decimos que el árbol está *equilibrado*[2]. El árbol del esquema anterior no está ni completo ni equilibrado.

Para el caso que nos ocupa, la clase correspondientes a cada Estudiante vuelve a extender a la clase InfoEstudiante, agregando las referencias para el subárbol izquierdo y derecho, y los métodos de acceso y manipulación de estos campos. La programación se puede ver en el listado 8.15 en la siguiente página.

Como se ve de la declaración del registro ArbolEstudiante, tenemos una estructura recursiva, donde un registro de estudiante es la información, con dos referencias a registros de estudiantes.

[2]En inglés, *balanced*

Código 8.15 Clase ArbolEstudiante para cada registro o nodo (**ArbolEstudiante**)

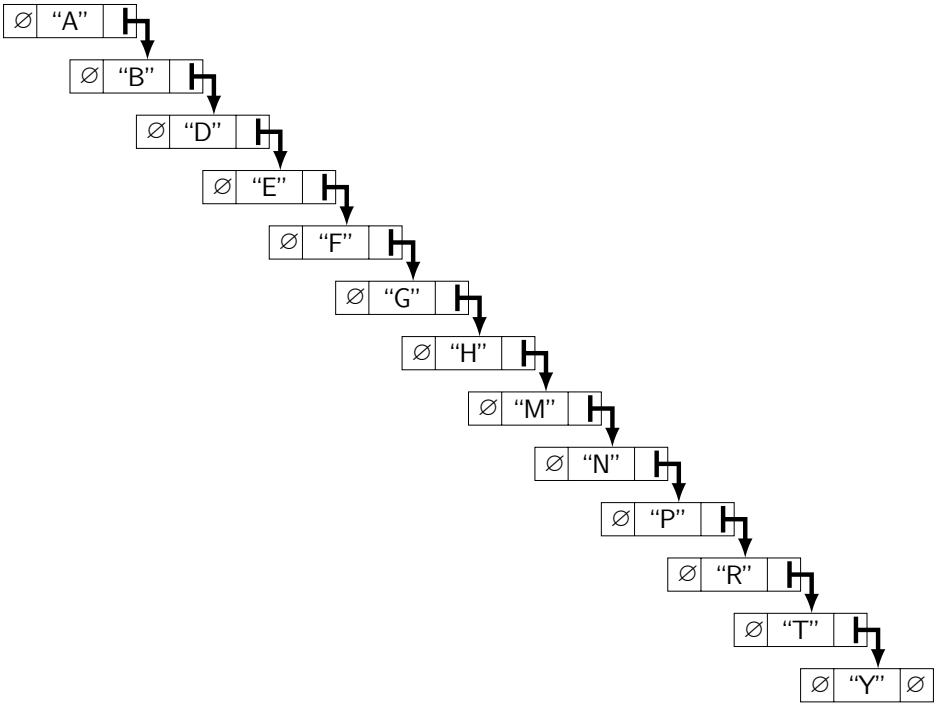
```
1: class ArbolEstudiante extends InfoEstudiante    {
2:     /* Referencias a subárbol izquierdo y derecho *\
3:     protected ArbolEstudiante izqrdo,
4:     /**
5:      * Constructor con la información como argumentos.
6:      */
7:     public ArbolEstudiante(String nmbre, String cnta,
8:                            String crrera)   {
9:         super(nmbre, cnta, crrera);
10:        izqrdo = dercho = null;
11:    }
12:    /**
13:     * Proporciona la referencia del subárbol izquierdo.
14:     */
15:    public ArbolEstudiante getIzqrdo()  {
16:        return izqrdo;
17:    }
18:    /**
19:     * Proporciona la referencia al subárbol derecho.
20:     */
21:    public ArbolEstudiante getDercho()  {
22:        return dercho;
23:    }
24:    /*
25:     * Actualiza el subárbol izquierdo.
26:     */
27:    public void setIzqrdo(ArbolEstudiante nuevo)  {
28:        izqrdo = nuevo;
29:    }
30:    /**
31:     * Actualiza el subárbol derecho.
32:     */
33:    public void setDercho(ArbolEstudiante nuevo){
34:        dercho = nuevo;
35:    }
36: }
```

8.3.1. Construcción de árboles binarios para ordenamientos

Los árboles de ordenamiento se van construyendo conforme se van insertando los nodos. Cuando se inicia la inserción y el árbol está vacío, el primer dato que llega se coloca en la raíz. A partir de ese momento, si el dato nuevo es menor al

de la raíz, va a quedar en el subárbol izquierdo, mientras que si no es menor va a quedar en el subárbol derecho. Esto quiere decir que si dos registros tienen la misma llave (en este caso, nombres iguales pero que el resto de la información difiere) el segundo que llegue va a quedar "a la derecha" del primero que llegó (en el subárbol derecho del nodo que contiene al que llegó primero). En el diagrama del árbol binario que dimos arriba, y si consideramos que la llave es la letra que aparece, un posible orden de llegada de los registros pudo ser "H", "P", "Y", "R", "M", "A", "E", "C", "F", "D", "T", "B", "N". No cualquier orden de llegada produce el mismo árbol. En este ejemplo, si los registros van llegando ya ordenados de menor a mayor, el árbol que se obtiene es el que se puede ver en la figura 8.7.

Figura 8.7 Árbol que se forma si los registros vienen ordenados



Esto es lo que conocemos como un *árbol degenerado*, pues degeneró en una lista. Esta es una situación poco afortunada, pues cuando utilizamos un árbol para ordenar, queremos que los datos lleguen con un cierto desorden, que nos garantiza

un árbol más equilibrado. Dados $\boldsymbol{n}$ registros en un árbol, la menor profundidad se alcanza cuando el árbol está equilibrado, mientras que la mayor profundidad se alcanza cuando tenemos un árbol degenerado como el que acabamos de mostrar. Hay varios resultados importantes respecto a árboles binarios que se reflejan en la eficiencia de nuestras búsquedas. En un árbol equilibrado con $\boldsymbol{n}$ nodos, una hoja está, en promedio, a distancia $\mathbf{log_2}\, \boldsymbol{n}$; mientras que la distancia de la cabeza de una lista a alguno de sus elementos es en promedio de $\boldsymbol{n}/\mathbf{2}$.

Una forma de buscar que el árbol quede equilibrado es mediante el uso de una raíz que contenga un dato fantasma, y que este dato fantasma resulte ser un buen "dato de en medio" para el árbol. Por ejemplo, dado que la "M" se encuentra a la mitad del alfabeto, un buen dato fantasma para la raíz pudiera ser un registro con M's en él. De esa manera, al menos la mitad de los datos quedaría a su izquierda y la otra mitad a su derecha. En general, asumimos que los datos vienen desordenados, y en este caso tenemos una distribución adecuada para armar el árbol.

8.3.2. La clase ArbolOrden

Queremos reutilizar en la medida de lo posible las clases que tenemos tanto para el manejo de la estructura de datos como del menú que invoca a los métodos de aquélla. Para ello mantendremos las firmas de los métodos públicos de la clase ListaCurso, excepto que en todo lugar donde aparece un objeto de la clase Estudiante nosotros vamos a usar uno de la clase ArbolEstudiante. Hay dos métodos en esta clase que no vamos a utilizar – porque la única manera en que vamos a meter registros es manteniendo el orden – y que son agregaEst y agregaEstFinal. Lo que vamos a hacer con estos dos métodos es que regresen el valor false ya que no va a ser cierto que agreguemos así.

Otro aspecto importante es que como la estructura de datos es recursiva, lo ideal es manejarla con métodos recursivos. Pero como la firma de cada uno de los métodos públicos es fija, lo que haremos en cada uno de los métodos públicos es desde allí llamar por primera vez al método recursivo que va a hacer el trabajo, y el método recursivo, que va a ser privado, es el que se va a encargar de todo.

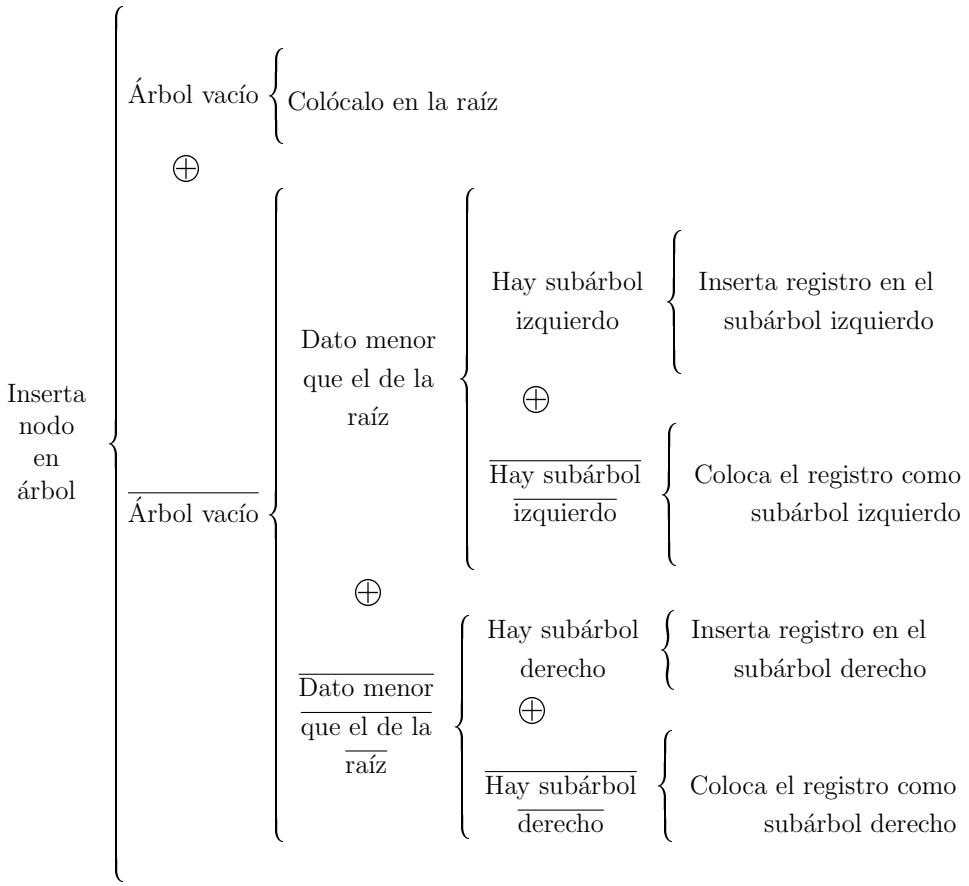
8.3.3. Inserción

El algoritmo que nos permite insertar un registro en el árbol es como se muestra en el diagrama de la figura 8.8 en la página opuesta. Queda programado, de

acuerdo al algoritmo anterior, como se puede apreciar en el listado 8.16 en la siguiente página.

Del algoritmo en la figura 8.8 podemos ver que únicamente podemos agregar un registro cuando le encontramos lugar como una hoja. Lo que tenemos que hacer es ir bajando por la izquierda o la derecha del árbol, dependiendo del resultado de comparar al registro que se desea agregar con el que se encuentra en la raíz del subárbol en turno. El algoritmo se traduce bastante directamente a código.

Figura 8.8 Agregar un registro manteniendo el orden



Código 8.16 Agregar un registro en un árbol binario ordenado (**ArbolOrden**)

```
1: public class ArbolOrden  {
2:     /**  La raíz del árbol */
3:     private ArbolEstudiante raiz;
4:     private String grupo;
5:     private int numRegs;
6:     /** Agrega un registro al final de la lista
7:      * @param ArbolEstudiante nuevo El registro a agregar.
8:      */
9:     public boolean agregaEstFinal(ArbolEstudiante nuevo)  {
10:         return false;
11:     }
12:     /** Agrega un registro al principio de la lista
13:      * @param InfoEstudiante nuevo A quien se va a agregar
14:      */
15:     public boolean agregaEst(ArbolEstudiante nuevo)  {
16:         return false;
17:     }
18:     /** Agrega un registro donde la toca para mantener la lista
19:      * ordenada
20:      * @param ArbolEstudiante nuevo A quien se va a agregar
21:      * @returns boolean Si pudo o no agregar
22:      */
23:     public boolean agregaEstOrden(ArbolEstudiante nuevo)  {
24:        if (raiz == null)  {//Es el primero
25:            raiz = nuevo;
26:            return true;
27:        }
28:        return meteEnArbol(raiz, nuevo);
29:     }
30:     /** Tengo la seguridad de llegar acá con raiz != null */
31:     private boolean meteEnArbol(ArbolEstudiante raiz,
32:                                 ArbolEstudiante nuevo)  {
33:         int compara = nuevo.daNombre().trim().toLowerCase()
34:                       .compareTo(raiz.daNombre().trim().toLowerCase());
35:         if (compara < 0 && raiz.getIzqrdo() == null)    {
36:             raiz.ponIzqrdo(nuevo);
37:             return true;
38:         }
39:         if (compara >= 0 && raiz.getDercho() == null)   {
40:             raiz.setDercho(nuevo);
41:             return true;
42:         }
43:         if (compara < 0)
44:             return meteEnArbol(raiz.getIzqrdo(), nuevo);
45:         else
46:             return meteEnArbol(raiz.getDercho(), nuevo);
47:     }
```

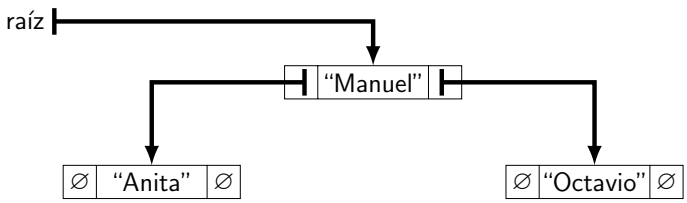
8.3.4. Listar toda la base de datos

Tenemos tres posibles recorridos para listar la información en un árbol binario ordenado, dependiendo de cuál sea el orden que sigamos para ir listando el contenido de cada registro:

Recorrido en *preorden*
Recorrido *simétrico*
Recorrido en *postorden*

El siguiente algoritmo que vamos a revisar es el que se encarga de mostrar o recorrer la lista. Si nuestra lista únicamente tuviera tres registros, y el árbol estuviera equilibrado, se vería como en la figura 8.9 (el orden de llegada pudo haber sido primero "Anita" y después "Octavio", pero si hubiera llegado primero "Anita" u "Octavio" el árbol no se hubiera armado equilibrado).

Figura 8.9 Ejemplo simple para recorridos de árboles



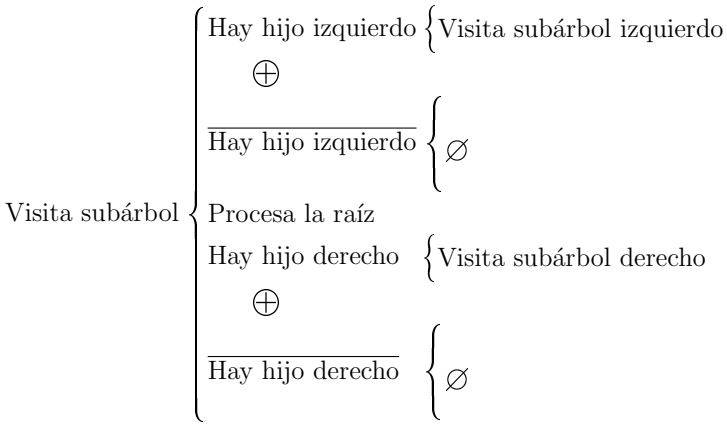
Como podemos ver en este esquema, para listar el contenido de los nodos en orden tenemos que recorrer el árbol de la siguiente manera:

Muestra el hijo izquierdo
Muestra la raíz
Muestra el hijo derecho

y este orden de recorrido nos entrega la lista en el orden adecuado. A este recorrido se le conoce como *inorden* o *simétrico*, pues la raíz se visita entre los dos hijos. Tenemos otros órdenes de recorrido posibles. Por ejemplo, si visitáramos primero a la raíz, después al hijo izquierdo y después al hijo derecho, tendríamos lo que se conoce como preorden. Y si primero visitáramos a los dos hijos y por último a la raíz, tendríamos lo que se conoce como postorden o notación polaca (este nombre se utiliza porque los árboles son muy útiles no nada más para ordenamientos, sino que se utilizan para denotar la precedencia en operaciones aritméticas). Dado que

los árboles son estructuras recursivas, el hijo izquierdo es, a su vez, un árbol (lo mismo el hijo derecho). Por ello, si pensamos en el caso más general, en que el hijo izquierdo pueda ser un árbol tan complicado como sea y el hijo derecho lo mismo, nuestro algoritmo para recorrer un árbol binario arbitrario quedaría como se muestra en el esquema de la figura 8.10.

Figura 8.10 Recorrido simétrico de un árbol



En nuestro caso, el proceso del nodo raíz de ese subárbol en particular consiste en escribirlo. El método público que muestra toda la lista queda con la firma como la tenía en las otras dos versiones que hicimos, y programamos un método privado que se encargue ya propiamente del recorrido recursivo, cuya firma será

```
private void listaArbol(Consola cons, ArbolEstudiante raiz)
```

Debemos recordar que éste es un método privado de la clase ArbolOrden, por lo que siempre que lo invoquemos tendrá implícito un objeto de esta clase como primer argumento. Los métodos listaTodos y listaArbol quedan programados como se muestra en los listados 8.17 en la página opuesta y 8.18 en la página opuesta respectivamente.

Código 8.17 Listado de la base de datos completa (**ArbolOrden**)

```
208:     /** Lista todos los registros del Curso.
209:      * @param Consola cons dónde escribir.
210:      */
211:     public void listaTodos(Consola cons)    {
212:         if (raiz == null)   {
213:             cons.imprimeln("No hay registros en la base de datos");
214:             return;
215:         }
216:         listaArbol(cons, raiz);
217:     }
```

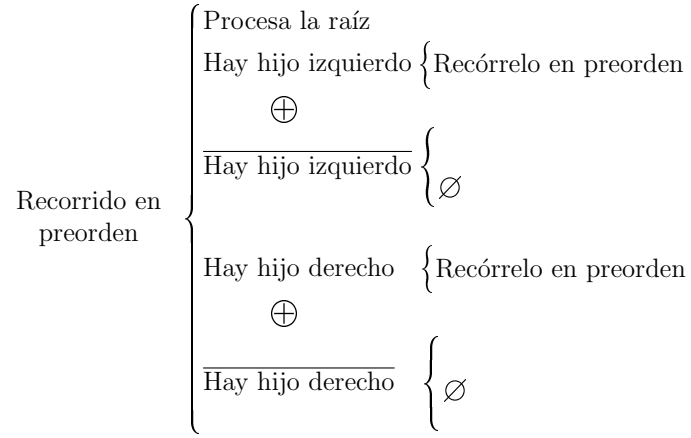
Código 8.18 Recorrido simétrico del árbol (**ArbolOrden**)

```
218:     /** Recorrido simétrico recursivo de un árbol.
219:      * @param Consola cons Donde escribir.
220:      *                 ArbolEstudiante raiz La raíz del subárbol.
221:      */
222:     private void listaArbol(Consola cons, ArbolEstudiante raiz)    {
223:         if (raiz == null)
224:             return;
225:         listaArbol(cons, raiz.getIzqrdo());
226:         cons.imprimeln(raiz.daRegistro());
227:         listaArbol(cons,raiz.getDercho());
228:     }
```

8.3.5. Conservación de la aleatoriedad de la entrada

Quisiéramos guardar el contenido del árbol en disco, para la próxima vez empezar a partir de lo que ya tenemos. Si lo guardamos en orden, cuando lo volvamos a cargar va a producir un árbol degenerado, pues ya vimos que lo peor que nos puede pasar cuando estamos cargando un árbol binario es que los datos vengan en orden (ya sea ascendente o descendente). Por ello, tal vez sería más conveniente recorrer el árbol de alguna otra manera. Se nos ocurre que si lo recorremos en preorden, vamos a producir una distribución adecuada para cuando volvamos a cargar el directorio. Esta distribución va a ser equivalente a la original, por lo que estamos introduciendo un cierto factor aleatorio. El algoritmo es igual de sencillo que el que recorre en orden simétrico y se muestra en la figura 8.11 en la siguiente página.

Figura 8.11 Recorrido en preorden de un árbol

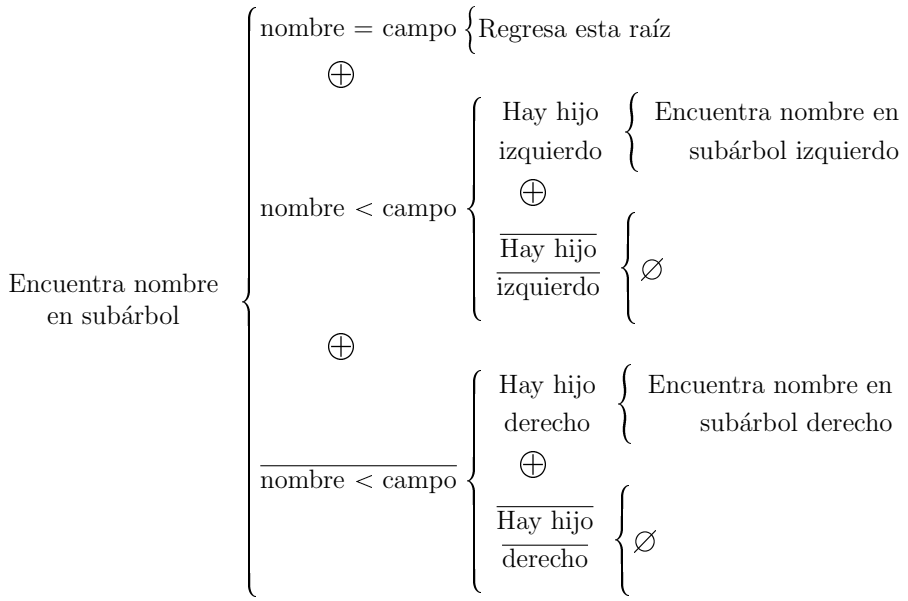


8.3.6. Búsquedas

Para encontrar un registro dado en el árbol, debemos realizar algo similar a cuando lo insertamos. Compara con la llave de la raíz. Si son iguales, ya lo encontramos. Si el que busco es menor, recursivamente busco en el subárbol izquierdo. Si el que busco es mayor, recursivamente busco en el árbol derecho. Decido que no está en el árbol, si cuando tengo que bajar por algún hijo, éste no existe. La única diferencia entre el algoritmo que inserta y el que busca, es la reacción cuando debemos bajar por algún hijo (rama) que no existe. En el caso de la inserción tenemos que crear al hijo para poder acomodar ahí la información. En el caso de la búsqueda, nos damos por vencidos y respondemos que la información que se busca no está en el árbol. El algoritmo se muestra en la figura 8.12 en la página opuesta.

Si no se encuentra al estudiante, el método deberá regresar una referencia nula. La programación de este método se puede apreciar en el listado 8.19 en la página opuesta.

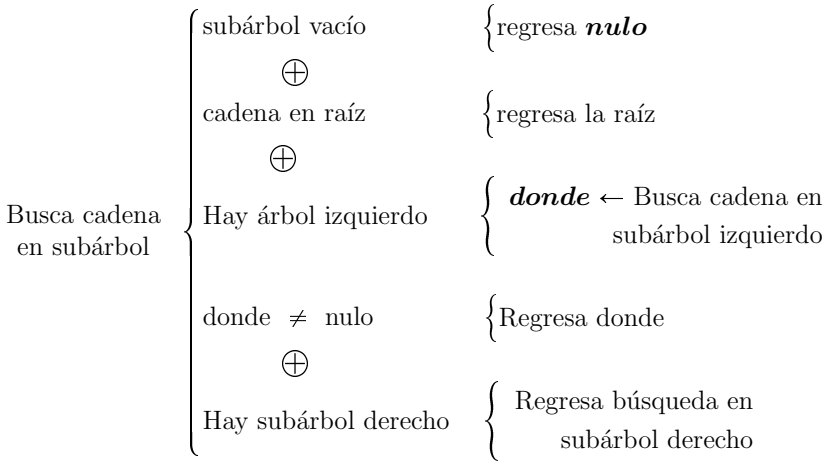
Figura 8.12 Búsqueda en un árbol ordenado



Código 8.19 Búsqueda del registro con el nombre dado (**ArbolOrden**)

```
229:    /** Localiza el nodo del árbol en el que está este nombre.
230:     * @param ArbolEstudiante raiz La raíz del subárbol en el que va
231:     * a buscar.
232:     * @param String nombre El nombre que está buscando.
233:     * @returns ArbolEstudiante La referencia de donde se encuentra
234:     * este registro, o null si no lo encontró.
235:     */
236:    private ArbolEstudiante buscaDonde(ArbolEstudiante raiz,
237:                                       String nombre)   {
238:        if (raiz == null)     return null;
239:        int compara = nombre.trim().toLowerCase().
240:                compareTo(raiz.daNombre().trim().toLowerCase());
241:        if (compara == 0)     return raiz;
242:        if (compara < 0)
243:            return buscaDonde(raiz.getIzqrdo(), nombre);
244:        else
245:            return buscaDonde(raiz.getDercho(), nombre);
246:    }
```

Figura 8.13 Búsqueda de una subcadena



El otro tipo de búsqueda que tenemos en nuestro programa es el de localizar a algún registro que contenga a la subcadena pasada como parámetro, en el campo pasado como parámetro. Para dar satisfacción a esta solicitud debemos recorrer el árbol hasta encontrar la subcadena en alguno de los registros. El problema es que como se trata de una subcadena no nos ayuda el orden en el árbol. El algoritmo para esta búsqueda se muestra en la figura 8.13 mientras que la programación se muestra en el listado 8.20.

Código 8.20 Búsqueda de subcadena en determinado campo (**ArbolOrden**)1/2

```
177:     /**
178:      * Busca al registro que contenga a la subcadena.
179:      *
180:      * @param int cual Cual es el campo que se va a comparar.
181:      * @param String subcad La cadena que se está buscando.
182:      * @returns int El registro deseado o -1.
183:      */
184:     public ArbolEstudiante buscaSubcad(int cual, String subcad) {
185:         subcad = subcad.trim().toLowerCase();
186:         return preOrden(cual, subcad, raiz);
187:     }
```

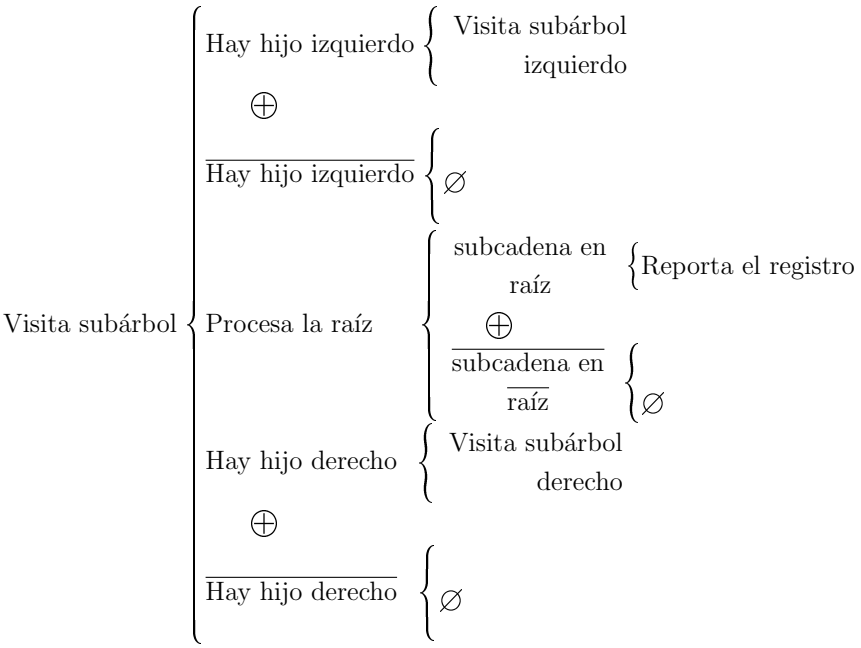
Código 8.20 Búsqueda de subcadena en determinado campo (**ArbolOrden**) 2/2

```
177:     /** Recorre el árbol en preorden hasta encontrar una subcadena.
178:      * @param int cual Campo en el cual buscar.
179:      * @param String subcad Cadena a localizar.
180:      * @param ArbolEstudiante raiz Raíz del árbol donde va a buscar.
181:      * @returns ArbolEstudiante El primer registro que cumpla
182:      */
183:     private ArbolEstudiante preOrden(int cual, String subcad,
184:                                      ArbolEstudiante raiz) {
185:         if (raiz == null) {
186:             return null;
187:         }
188:         String elCampo = raiz.daCampo(cual).trim().toLowerCase();
189:         if (elCampo.indexOf(subcad) != -1) {
190:             return raiz;
191:         }
192:         ArbolEstudiante donde = null;
193:         if (raiz.getIzqrdo() != null) {
194:             donde = preOrden(cual, subcad, raiz.getIzqrdo());
195:         }
196:         if (donde != null) {
197:             return donde;
198:         }
199:         if (raiz.getDercho() != null) {
200:             return preOrden(cual, subcad, raiz.getDercho());
201:         }else {
202:             return null;
203:         }
204:     }
```

8.3.7. Listado condicionado de registros

Otro método importante en nuestro sistema es el listado de aquellos registros que contengan, en el campo elegido, una subcadena. En este caso, al igual que en el listado de todos los registros, deberemos recorrer todo el árbol en orden simétrico, de tal manera que al llegar a cada nodo, si es que el nodo contiene a la subcadena en el lugar adecuado, lo listemos, y si no no. El algoritmo para este método se parece mucho al que lista a todos los registros, excepto que al visitar la raíz, antes de imprimir, verifica si cumple la condición. El algoritmo se muestra en la figura 8.14 en la siguiente página. La programación de este algoritmo se puede ver en el listado 8.21 en la siguiente página.

Figura 8.14 Selección de registros que cumplen una condición



Código 8.21 Listado de registros que contienen a una subcadena (**ArbolOrden**) 1/2

```
227:     /** Imprime los registros que cazan con un cierto patrón.
228:      * @param Consola cons Dispositivo en el que se va a escribir.
229:      * @param int cual  Con cuál campo se desea comparar.
230:      * @param String subcad Con el que queremos que cace.
231:      */
232:     public void losQueCazanCon(Consola cons, int cual,
233:                                            String subcad)    {
234:         subcad = subcad.trim().toLowerCase();
235:         int cuantos = 0;
236:         if (raiz == null)
237:             cons.imprimeln("No hay registros en la base de"
238:                            + "datos");
239:         else
240:             cuantos = buscaCazan(cons, cual, subcad, raiz);
241:         if (cuantos == 0)
242:             cons.imprimeln("No hay registros que cacen.");
243:     }
```

Código 8.21 Listado de registros que contienen a subcadena (**ArbolOrden**) 2/2

```
244:     /** Recorre el árbol verificando si cazan.
245:      * @param Consola cons Medio para escribir.
246:      *                int cual. Campo en el que se va a buscar.
247:      *                String subcad Cadena a buscar.
248:      *                ArbolEstudiante raiz Raíz del árbol a recorrer.
249:      *  @returns int El total de registros que cazaron */
250:     private int buscaCazan(Consola cons, int cual, String subcad,
251:                                        ArbolEstudiante raiz)    {
252:         int este = 0;
253:         if (raiz == null)
254:             return 0;
255:         este = buscaCazan(cons, cual, subcad, raiz.getIzqrdo());
256:         if (raiz.daCampo(cual).trim().toLowerCase().indexOf(subcad)
257:                                != -1)   {
258:             cons.imprimeln(raiz.daRegistro());
259:             este++;
260:         }
261:         este += buscaCazan(cons, cual, subcad, raiz.getDercho());
262:         return este;
263:     }
```

8.3.8. Eliminación de nodos

Hemos llegado a la parte más complicada de manejar en un árbol binario, que es la eliminación de alguno de los nodos del árbol. Si el nodo es una hoja, realmente no hay ningún problema: simplemente hay que regresar al padre del nodo y nulificar el apuntador a esa hoja.

Una manera de localizar al padre de un nodo es el volver a realizar la búsqueda desde la raíz, pero recordando en todo momento al nodo anterior que se visitó. Otra manera es la de poner a cada nodo a "apuntar" hacia su padre, lo que se puede hacer trivialmente en el momento de insertar a un nodo, pues al insertarlo tenemos un apuntado al padre (de quien lo vamos a colgar) y un apuntador al hijo (a quien vamos a colgar). Optamos por la primera solución, para no modificar ya el registro de cada estudiante. La programación del método que localiza al padre se da en el listado 8.22 en la siguiente página. El algoritmo es similar al que se dio para la búsqueda de una subcadena, por lo que ya no lo mostramos.

Código 8.22 Localización del padre de un nodo (**ArbolOrden**)

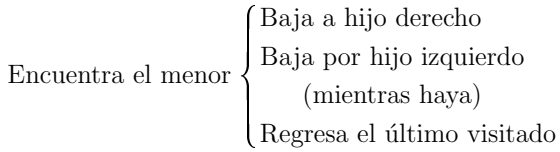
```
147:     /**
148:      * Busca al padre de un nodo, recorriendo el árbol desde la
149:      * raíz.
150:      *
151:      * @param ArbolEstudiante raiz Raíz del subárbol
152:      *        ArbolEstudiante deQuien A quien se le busca el
153:      *                padre
154:      * @returns ArbolEstudiante La referencia del padre
155:      */
156:     private ArbolEstudiante buscaPadre(ArbolEstudiante raiz,
157:                                        ArbolEstudiante deQuien)    {
158:         ArbolEstudiante padre = null;
159:         if (raiz.getIzqrdo() == deQuien)
160:             return raiz;
161:         if (raiz.getDercho() == deQuien)
162:             return raiz;
163:         if (raiz.getIzqrdo() != null)
164:             padre = buscaPadre(raiz.getIzqrdo(), deQuien);
165:         if (padre == null && raiz.getDercho() != null)
166:             return buscaPadre(raiz.getDercho(), deQuien);
167:         else
168:             return padre;
169:     }
```

Como dijimos, una vez que se tiene al padre de una hoja, todo lo que hay que hacer es identificar si el nodo es hijo izquierdo o derecho y poner el apuntador correspondiente en null.

Resuelta la eliminación de una hoja, pasemos a ver la parte más complicada, que es la eliminación de un nodo intermedio. Veamos, por ejemplo, el árbol de la figura 8.6 en la página 267 y supongamos que deseamos eliminar el nodo etiquetado con "E". ¿Cómo reacomodamos el árbol de tal manera que se conserve el orden correcto? La respuesta es que tenemos que intercambiar a ese nodo por el nodo menor de su subárbol derecho. ¿Por qué? Si colocamos al nodo menor del subárbol derecho en lugar del que deseamos eliminar, se sigue cumpliendo que todos los que estén en el subárbol izquierdo son menores que él, mientras que todos los que estén en el subárbol derecho son mayores o iguales que él:

Para localizar el elemento menor del subárbol derecho simplemente bajamos a la raíz del subárbol derecho y de ahí en adelante seguimos bajando por las ramas izquierdas hasta que ya no haya ramas izquierdas.El algoritmo para encontrar el elemento menor de un subárbol se encuentra en la figura 8.15 en la página opuesta. La programación de este método se muestra en el listado 8.23.

Figura 8.15 Algoritmo para encontrar el menor de un subárbol



Código 8.23 Localiza al menor del subárbol derecho **(ArbolOrden)**

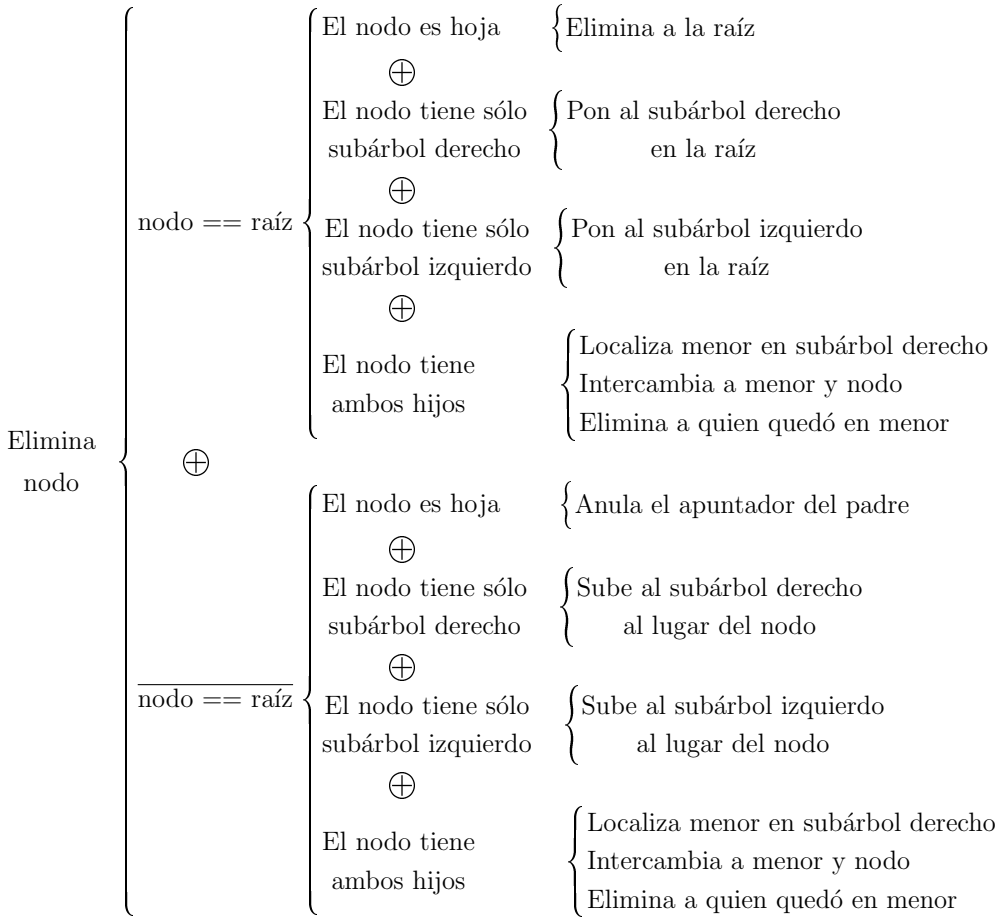
```
138:     /**
139:      * Localiza al menor de un subárbol.
140:      *
141:      * @param ArbolEstudiante raiz La raíz del subárbol
142:      * @returns ArbolEstudiante La referencia del menor
143:      */
144:     private ArbolEstudiante daMenor(ArbolEstudiante raiz) {
145:         ArbolEstudiante menor = raiz;
146:         while (menor.getIzqrdo() != null)
147:             menor = menor.getIzqrdo();
148:         return menor;
149:     }
```

Una vez localizado el menor del subárbol derecho, lo intercambiamos con el nodo que queremos eliminar, en este caso "E". Con este intercambio, logramos mover a "E" hacia una hoja, y entonces ya podemos eliminarlo fácilmente. Cabe aclarar que para intercambiar dos nodos preservando su estructura, basta con intercambiar el campo correspondiente a la información, dejando las referencias sin tocar. Pero puede muy fácil suceder que el menor de un subárbol no sea forzosamente una hoja. Entonces, lo que debemos hacer con él es, nuevamente intercambiarlo con el menor de su subárbol derecho.

Pudiera presentarse el caso de que el nodo no tuviera subárbol izquierdo. En ese caso, debemos "subir" el subárbol derecho al lugar que ocupa ahora el nodo que deseamos eliminar. Por ejemplo, si en el árbol anterior original deseáramos eliminar al nodo que contiene una "A" (que no tiene subárbol izquierdo), simplemente "subimos" al subárbol etiquetado con "D" y todas las relaciones entre los nodos se mantienen. En realidad, si el nodo tiene sólo a uno de los hijos (sea el izquierdo o el derecho) el nodo se elimina fácilmente subiendo al subárbol que sí tiene a su lugar.

Otra situación a considerar es cuando se desea eliminar a la raíz del árbol. En ese caso no vamos a tener padre, aún cuando el único nodo en el árbol sea éste. A la raíz se le debe tratar de manera un poco diferente. El algoritmo para eliminar a un nodo se muestra en la figura 8.3.8. La programación del método se puede ver en el listado 8.24 en la página opuesta.

Figura 8.16 Eliminación de un nodo en un árbol



Código 8.24 Eliminación de un nodo en el árbol (**ArbolOrden**) 1/3

```
57:     /**
58:      * Define si un nodo del árbol es hoja */
59:     private boolean esHoja(ArbolEstudiante nodo)  {
60:         if (nodo.getIzqrdo() == null && nodo.getDercho() == null)
61:            return true;
62:         return false;
63:     }
64:     /**
65:      * Quita el registro solicitado (por nombre) de la lista
66:      * @param String nombre El nombre del registro */
67:     public boolean quitaEst(String nombre)    {
68:         ArbolEstudiante donde = buscaDonde(raiz, nombre);
69:         if (donde == null)
70:             return false;
71:         if ( donde == raiz && esHoja(raiz))  {
72:             raiz = null;
73:             return true;
74:         }
75:         if (donde == raiz.getIzqrdo()
76:                 && esHoja(raiz.getIzqrdo()))  {
77:             raiz.ponIzqrdo(null);
78:             return true;
79:         }
80:         if (donde == raiz.getDercho()
81:              && esHoja(raiz.getDercho()))  {
82:             raiz.ponIzqrdo(null);
83:             return true;
84:         }
85:         return borraA(donde);
86:     }
87:     private boolean borraA(ArbolEstudiante donde)    {
88:         ArbolEstudiante conQuien;
89:         if (donde == raiz.getIzqrdo()
90:                         && esHoja(raiz.getIzqrdo()))  {
91:             raiz.ponIzqrdo(null);
92:             return true;
93:         }
94:         if (donde == raiz.getDercho()
95:                && esHoja(raiz.getDercho()))  {
96:             raiz.ponIzqrdo(null);
97:             return true;
98:         }
99:         ArbolEstudiante padre;
100:        if (donde == raiz)
101:            padre = null;
102:        else
103:            padre = buscaPadre(raiz, donde);
```

Código 8.24 Eliminación de un nodo en el árbol (**ArbolOrden**)2/3

```
104:         if (donde.getIzqrdo() == null && donde.getDercho() != null
105:                                 && padre == null) {
106:                     // Es la raíz y tiene nada más subárbol derecho}
107:             raiz = donde.getDercho();
108:             return true;
109:         }
110:         if (donde.getIzqrdo() != null && donde.getDercho() == null
111:                                 && padre == null) {
112:                                // Sólo hay subárbol izquierdo
113:             raiz = donde.getIzqrdo();
114:             return true;
115:         }
116:
117:         if (padre == null)   {
118:                   // Tiene a los dos subárboles
119:             conQuien = daMenor(donde.getDercho());
120:             donde.ponNombre(conQuien.daNombre());
121:             donde.ponCuenta(conQuien.daCuenta());
122:             donde.ponCarrera(conQuien.daCarrera());
123:             return borraA(conQuien);
124:         }
125:
126:         if (donde.getIzqrdo()==null
127:                   && donde.getDercho() == null)   {
128:                             // Es  hoja
129:             if (padre.getIzqrdo()==donde)   {
130:                 padre.ponIzqrdo(null);
131:                 return true;
132:             }
133:             if (padre.getDercho() == donde)    {
134:                 padre.setDercho(null);
135:                 return true;
136:             }
137:         }
138:         if (donde.getIzqrdo() == null)  {
139:                   // Sólo tiene subárbol derecho
140:             if (padre.getIzqrdo() == donde)    {
141:                 padre.ponIzqrdo(donde.getDercho());
142:                 return true;
143:             }
144:             if (padre.getDercho() == donde)    {
145:                 padre.setDercho(donde.getDercho());
146:                 return true;
147:             }
148:         }
```

Código 8.24 Eliminación de un nodo en el árbol (**ArbolOrden**) 3/3

```
149:         if (donde.getDercho() == null && padre != null)     {
150:             if (padre.getIzqrdo() == donde)     {
151:                 padre.ponIzqrdo(donde.getIzqrdo());
152:                 return true;
153:             }
154:             if (padre.getDercho() == donde)     {
155:                 padre.setDercho(donde.getIzqrdo());
156:                 return true;
157:             }
158:         }
159:         // Ambos subárboles existen
160:         conQuien = daMenor(donde.getDercho());
161:         donde.ponNombre(conQuien.daNombre());
162:         donde.ponCuenta(conQuien.daCuenta());
163:         donde.ponCarrera(conQuien.daCarrera());
164:         return borraA(conQuien);
165:     } // borraA
```

Por último, el mecanismo para modificar algún registro no puede ser, simplemente, modificar la información, pues pudiera ser que la llave cambiara y el registro quedara fuera de orden. La estrategia que vamos a utilizar para modificar la información de un registro es primero borrarlo y luego reinsertarlo, para evitar que se modifique la llave y se desacomode el árbol.

8.3.9. La clase MenuOrden

Utilizamos lo que tenemos programado en MenuLista para copiarlo a MenuOrden. Sustituimos la llamada a agregaEst por una llamada a agregaEstOrden para que los vaya acomodando bien en el árbol. Asimismo, todos los parámetros o valores de retorno que antes eran de la clase Estudiante ahora pasan a ser de la clase ArbolEstudiante.

Adicionalmente, en el método main cuando se declaran y crean los objetos miCurso y miMenu, se cambian las clases a que ahora sean de las clases ArbolOrden y MenuArbol respectivamente. Como tuvimos mucho cuidado en mantener la interfaz tal cual, con todos los métodos conservando su firma, no hay necesidad de realizar ningún otro cambio.

Con esto damos por terminado cómo mantener ordenada una lista de acuerdo a cierta llave, que corresponde a uno de los campos. Hay muchos algoritmos para ordenar una lista, pero no es material de estos temas, por lo que no lo revisaremos acá.

Manejo de errores en ejecución

9

9.1 Tipos de errores

Todos hemos padecido en algún momento errores de ejecución. Java tiene mecanismos muy poderosos para detectar errores de ejecución de todo tipo, a los que llama excepciones. Por ejemplo, si se trata de usar una referencia nula, Java terminará (*abortará*) el programa con un mensaje de error. Generalmente el mensaje tiene la forma:

```
Exception in thread "main" java.lang.NullPointerException
        at MenuLista.main(MenuLista.java:151)
```

En el primer renglón del mensaje Java nos dice el tipo de excepción que causó que el programa "abortara", que en este caso es NullPointerException, y a partir del segundo renglón aparece la "historia" de la ejecución del programa, esto es, los registros de activación montados en el stack de ejecución en el momento en que sucede el error. Hay muchos errores a los que Java va a reaccionar de esta manera, como por ejemplo usar como índice a un arreglo un entero menor que cero o mayor o igual al tamaño declarado (ArrayIndexOutOfBoundsException) o una división entre 0 (ArithmeticException).

La manera como responde la máquina virtual de Java cuando sucede un error de ejecución es que *lanza* lo que se conoce como una **excepción**. Una excepción es un objeto de alguna clase que hereda de la clase Throwable y que contiene información respecto a dónde se produjo el error y qué tipo de error es. Dependiendo de la clase a la que pertenece el objeto, fue el tipo de error que hubo. También el programador puede crear sus propias excepciones, como un mecanismo más de control del flujo del programa.

Dos clases importantes extienden a la clase Throwable, Error y Exception. El programador puede crear sus propias excepciones extendiendo a la clase Exception[1].

9.1.1. Excepciones en tiempo de ejecución (RuntimeException)

Las excepciones de tiempo de ejecución son las siguientes:

ArithmeticException Es lanzada cuando el programa intenta hacer una operación aritmética inválida, como la división de un entero entre 0[2]. Podemos ver un ejemplo en el listado 9.1.

Código 9.1 Ejemplo de una excepción aritmética

```
1:  import java.io.*;
2:
3:  class AritmExc  {
4:      private static int calcula(int k)   {
5:          return ((int)1000 / k);
6:      }
7:
8:      public static void main(String[] args)   {
9:          int k = 0;
10:         System.out.println("El valor de k es " +
11:                                 calcula(0));
12:     }
13: }
```

En este pequeño programa, el sistema lanzará una ArithmeticException si al invocar al método calcula se le pasa el valor de 0. Java no tiene manera,

[1]Es conveniente que aquellas clases que heredan a la clase Exception su nombre termine con "Exception" y el resto del nombre sirva para describir el tipo de error que está detectando.

[2]Java contempla el valor infinity, por lo que una división entre 0 cuyo resultado va a ser real no lanza esta excepción.

durante la compilación, de evitar este error aritmético. Lo que produce la ejecución de este programa se puede ver en la figura 9.1.

Figura 9.1 Ejecución de AritmExc

```
elisa@lambda ...ICC1/progs/excepciones % java AritmExc
Exception in thread "main" java.lang.ArithmeticException: / by zero
          at AritmExc.calcula(AritmExc.java:5)
          at AritmExc.main(AritmExc.java:11)
elisa@lambda ...ICC1/progs/excepciones %
```

ArrayStoreException Es lanzada cuando el programa intenta guardar un objeto de tipo erróneo en un arreglo. Veamos el ejemplo en el listado 9.2.

Código 9.2 Ejemplo de una excepción de la subclase **ArrayStoreException**

```
1:  class ArraySE  {
2:      private static void guardaObjeto(Object[] a, int i,
3:                                       Object objeto)   {
4:          a[i] = objeto;
5:      }
6:
7:      public static void main(String[] args)  {
8:          Integer[] intArray = new Integer[5];
9:          guardaObjeto(intArray, 0, new String("abc"));
10:     }
11: }
```

En la línea 4:, que es donde se guarda a un objeto en un arreglo, la asignación es perfectamente compatible. Java no puede detectar en el momento de compilación que va a tener un error al llamar a este método, ya que los argumentos son de un tipo que extiende a Object, y por lo tanto permitidos. La ejecución de este programa produce la salida que se muestra en la figura 9.2.

Figura 9.2 Ejecución de **ArraySE**

```
elisa@lambda ...ICC1/progs/excepciones % java ArraySE
Exception in thread "main" java.lang.ArrayStoreException
        at ArraySE.guardaObjeto(ArraySE.java:4)
        at ArraySE.main(ArraySE.java:9)
elisa@lambda ...ICC1/progs/excepciones %
```

ClassCastException Es lanzada cuando el programa intenta aplicar un *cast* de una clase a un objeto de otra clase, y la conversión no está permitida. El programa en el listado 9.3, al ejecutarse, lanza esta excepción.

Código 9.3 Programa que lanza una excepción **ClassCastException**

```
1:  class ClassCE  {
2:      private static void guardaObjeto(Integer[] a, int i,
3:                                       Object objeto)  {
4:          a[i] = (Integer)objeto;
5:      }
6:
7:      public static void main(String[] args)  {
8:          Integer[] intArray = new Integer[5];
9:          guardaObjeto(intArray, 0, new String("abc"));
10:     }
11: }
```

Nuevamente, el programa está sintácticamente correcto ya que en un arreglo de objetos de la clase Integer podemos guardar un objeto al que le apliquemos esa envoltura (línea 4:). Sin embargo, al pasarle como parámetro una cadena, que es sintácticamente compatible con un objeto, el programa no puede hacer la coerción, por lo que lanza una excepción. La ejecución del programa ClassCE la podemos ver en la figura 9.3.

Figura 9.3 Ejecución del programa **ClassCE**

```
elisa@lambda ...ICC1/progs/excepciones % java ClassCE
Exception in thread "main" java.lang.ClassCastException: java.lang.String
        at ClassCE.guardaObjeto(ClassCE.java:4)
        at ClassCE.main(ClassCE.java:9)
elisa@lambda ...ICC1/progs/excepciones %
```

IllegalArgumentException Esta excepción es lanzada cuando métodos de Java o definidos por el usuario detecta que un argumento no es como se esperaba. Por ejemplo, si se desea sacar la raíz cuadrada de un número negativo.

IllegalMonitorStateException Tiene que ver con sincronización de procesos. Lo veremos más adelante.

IndexOutOfBoundsException Es lanzado cuando se intenta usar un elemento de un arreglo que no existe, porque el índice dado no está en el rango dado para los índices del arreglo.

NegativeArraySizeException Se lanza si es que se intenta crear un arreglo con un número negativo de elementos.

NullPointerException Se lanza si se trata de usar al objeto referido por una referencia nula.

SecurityException Se lanza cuando el proceso intenta ejecutar un método que no se puede ejecutar por razones de seguridad. Tiene que ver con el sistema de seguridad (SecurityManager).

java.util.EmptyStackException Es lanzada cuando un programa, que utiliza la clase Stack de Java intenta tener acceso a un elemento del stack cuando el stack está vacío.

java.util.NoSuchElementException Error relacionado con *enumeraciones*.

Hay dos aspectos íntimamente relacionados con las excepciones: el primero de ellos corresponde a las condiciones bajo las cuales una excepción es disparada o lanzada (*thrown*), mientras que el otro aspecto tiene que ver con la manera que tendrá la aplicación de reaccionar cuando una excepción es lanzada. Dependiendo del tipo de excepción que fue lanzada, la aplicación debe reaccionar de manera adecuada frente a ella y actuar en consecuencia. Algunas veces esta respuesta consistirá de alguna forma en que la aplicación se recupere y siga adelante; otras veces simplemente le permitirá a la aplicación "morir dignamente". A la reacción frente a una excepción se conoce como *cachar* la excepción, para poder hacer algo con ella.

En general, si algún método de alguna clase que estemos utilizando avisa que puede lanzar una excepción, la aplicación que use esta clase deberá prepararse para detectar si se lanzó o no la excepción, y si se lanzó, reaccionar de alguna manera frente a ella. A los enunciados que comprenden la reacción frente a la excepción – que cachan la excepción – es a lo que se conoce como el *manejador de la excepción*.

Se recomienda enfáticamente que las excepciones de tiempo de ejecución no sean cachadas (RuntimeException). Este tipo de excepción indica errores de diseño en la aplicación, por lo que si se decide atraparlas esto se deberá hacer únicamente en una etapa de depuración del programa, para obtener un poco más de información sobre la causa de la excepción.

Sabemos que un método es susceptible de lanzar una excepción si en el encabezado del método, a continuación del paréntesis que cierra la lista de parámetros, aparece el enunciado

throws *lista de clases de Excepciones*

y en el cuerpo del método, o de alguno de los métodos invocados dentro de ese cuerpo, deberá aparecer un enunciado

throw new *constructor de una excepción*

En el caso de las excepciones en tiempo de ejecución – RuntimeException – los métodos que pudieran incurrir en algún error de ese tipo no tienen que avisar que pudieran lanzar este tipo de excepciones. Asimismo, las excepciones serán lanzadas, en su momento, implícitamente o por el usuario, pero sin necesidad de avisar que pudieran ocurrir.

Cuando algún enunciado o método detecta una excepción, la máquina virtual de Java[3] busca en su entorno inmediato al manejador de la excepción. Si lo encuentra, lo ejecuta, y ya sea que procese la excepción o que simplemente la propague (la vuelva a lanzar). Si en el entorno inmediato no se encuentra al manejador de la excepción, la JVM sigue buscando hacia "afuera" en la cadena de llamadas del método en cuestión a ver si encuentra a algún manejador de la excepción. Si no lo encuentra en la aplicación se lo pasa a la JVM, que procede a suspender el programa con el mensaje de error correspondiente, y una historia de la cadena de llamadas hasta el punto donde la excepción fue lanzada.

Si en el método en el que se lanza una excepción que no sea de tiempo de ejecución no hay un manejador para la excepción, el método deberá incluir el enunciado **throws** ⟨*excepción*⟩ en su encabezado.

9.2 La clase Exception

La clase Exception es una clase muy sencilla que tiene, realmente, muy pocos métodos. De hecho, únicamente tiene dos constructores que se pueden usar en aquellas clases que hereden a Exception. La clase Throwable, superclase de Exception, es la que cuenta con algunos métodos más que se pueden invocar desde cualquier excepción, ya sea ésta de Java o del programador. Veamos primero los dos constructores de Exception.

public Exception() Es el constructor por omisión. La única información que proporciona es el nombre de la excepción.

public Exception(String msg) Construye el objeto pasándole una cadena, que proporciona información adicional a la del nombre de la clase.

[3] en adelante JVM.

La clase Throwable nos va a permitir un manejo más preciso de las excepciones. Tiene varios tipos de métodos.

Constructores:

public Throwable() Es el constructor por omisión.

public Throwable(String msg) Da la oportunidad de construir el objeto con información que se transmite en la cadena.

Métodos del stack de ejecución:

public native Throwable fillInStackTrace() Coloca en el objeto una copia de la cadena dinámica del stack de ejecución en ese momento.

public void printStackTrace() Escribe en la pantalla el contenido del stack de ejecución, respecto a la cadena de llamadas que llevaron al método que está abortando.

public void printStackTrace(PrintStream s) Lo mismo que el anterior, pero da la opción de escribir a un archivo en disco.

Métodos de acceso a campos:

public String getMessage() Devuelve la cadena con que la excepción fue creada. SI fue creada con el constructor por omisión devuelve null.

Método descriptivo:

public String toString() Regresa en una cadena el nombre de la clase a la que pertenece y la cadena con la que fue creada, en su caso.

Como la clase Exception extiende a la clase Throwable, cualquier excepción que nosotros declaremos que extienda a Exception contará con los métodos de Throwable.

Las excepciones de tiempo de ejecución siempre van a imprimir la cadena producida por toString() y el stack de ejecución producido por printStackTrace(). De esa manera tenemos información muy puntual de cuál fue el tipo de error y dónde exactamente se presentó.

9.3 Cómo detectar y cachar una excepción

Como ya mencionamos, una aplicación común y corriente puede lanzar distintas excepciones de tiempo de ejecución. Estas excepciones no tienen que ser vigiladas y detectadas, sino que pasan a la JVM, quien suspende la ejecución del programa.

Sin embargo, también pueden presentarse excepciones que no son de tiempo de ejecución y que sí deben ser vigiladas y manejadas de alguna manera. Entre ellas podemos mencionar las que son lanzadas en relación con procesos de entrada y salida (IOException); cuando durante la ejecución de una aplicación no se encuentra una clase que se debe cargar (ClassNotFoundException); cuando se pretende hacer una copia de un objeto que no es copiable (CloneNotSupportedException); cuando se da alguna interrupción en un proceso que se está ejecutando (InterruptedException); y algunas más. Además, el usuario puede declarar sus propias clases de excepciones, que pueden extender a una clase particular de excepciones o la clase genérica Exception.

La detección y manejo de una excepción se lleva a cabo en un bloque que toma la forma que se muestra en la figura 9.4.

Figura 9.4 Detección y manejo de excepciones

Sintaxis:

try {
⟨*enunciados donde puede ser lanzada la excepción*⟩
} (**catch (** ⟨*tipo de excepción*⟩ ⟨*identif*⟩ **)** {
⟨*Enunciados que reaccionan frente a la excepción*⟩
})+

Semántica:

Se pueden agrupar tantos enunciados como se desee en la cláusula try, por lo que al final de ella se puede estar reaccionando a distintas excepciones. Se elige uno y solo un manejador de excepciones, utilizando la primera excepción que califique con ser del tipo especificado en las cláusulas catch.

La parte que aparece a continuación de cada catch corresponde al manejador

de cada clase de excepción que se pudiera presentar en el cuerpo del try. El tipo de excepción puede ser Exception, en cuyo caso cualquier tipo de excepción va a ser cachada y manejada dentro de ese bloque. En general, una excepción que extiende a otra puede ser cachada por cualquier manejador para cualquiera de sus superclases. Una vez que se lista el manejador para alguna superclase, no tiene sentido listar manejadores para las subclases.

La manera como se ejecuta un bloque try en el que se pudiera lanzar una excepción es la siguiente:

- La JVM entra a ejecutar el bloque correspondiente.
- Como cualquier bloque en Java, todas las declaraciones que se hagan dentro del bloque son visibles únicamente dentro del bloque.
- Se ejecuta el bloque enunciado por enunciado.
- En el momento en que se presenta una excepción, la ejecución del bloque se interrumpe y la ejecución prosigue buscando a un manejador de la excepción.
- La ejecución verifica los manejadores, uno por uno y en orden, hasta que encuentre el primero que pueda manejar la excepción. Si ninguno de los manejadores puede manejar la excepción que se presentó, sale de la manera que indicamos hasta que encuentra un manejador, o bien la JVM aborta el programa.
- Si encuentra un manejador adecuado, se ejecuta el bloque correspondiente al manejador.
- Si no se vuelve a lanzar otra excepción, la ejecución continúa en el enunciado que sigue al último manejador.

El programa que se encuentra en el listado 9.4 cacha las excepciones posibles en el bloque del try mediante una cláusula catch para la superclase Exception. Una vez dentro del manejador, averigua cuál fue realmente la excepción que se disparó, la reporta y sigue adelante con la ejecución del programa.

Código 9.4 Manejo de una excepción a través de la superclase (**CatchExc**)1/2

```
1:  import java.io.*;
2:
3: class CatchExc     {
4:     public static void main(String[] args)    {
5:         int k = 0;
6:         int c;
7:         int i = 3;
8:         int[] enteros = new int[3];
```

Código 9.4 Manejo de una excepción a través de la superclase (**CatchExc**)2/2

```
 9:          try  {
10:              System.out.println("Dame␣un␣entero␣para␣trabajar.");
11:              while((c = System.in.read()) >= '0'  && c <= '9' )
12:                  k = k * 10 + (c - ((int)'0'));
13:              System.out.println("Leí␣el␣valor␣" + k);
14:              c = 1000 / k;
15:              System.out.println("El␣valor␣final␣de␣c␣es␣" + c);
16:              enteros[i] = c;
17:          }
18:          catch(Exception e)   {
19:              System.out.println("La␣excepción␣es␣de␣la␣clase␣:\n\t" +
20:                                 e.getClass());
21:          }
22:          System.out.println("A␣punto␣de␣salir␣normalmente␣" +
23:                             "␣del␣método␣main");
24:      }
25: }
```

Este programa cacha cualquier excepción que se presente en el bloque try y la maneja, permitiendo que el programa termine normalmente. Dos ejecuciones consecutivas del mismo se pueden ver en la figura 9.5.

Figura 9.5 Excepciones de tiempo de ejecución cachadas con una superclase

```
elisa@lambda ...ICC1/progs/excepciones % java CatchExc
Dame un entero para trabajar.
0
Lei el valor 0
La excepción es de la clase :class java.lang.ArithmeticException
A punto de salir normalmente del método main
elisa@lambda ...ICC1/progs/excepciones % java CatchExc
Dame un entero para trabajar.
17
Leí el valor 17
El valor final de c es 58
La excepción es de la clase :class java.lang.ArrayIndex
OutOfBoundsException
A punto de salir normalmente del método main
```

Si queremos que el programa se interrumpa aún después de lanzada una excepción, podemos "relanzarla" en el manejador. En este caso, el método en el que esto se hace debe avisar que se va a lanzar una excepción, ya que habrá una salida brusca del mismo. En el listado 9.5 modificamos la clase CatchExc – CatchExc1 – para que relance la excepción y termine la ejecución. En la figura 9.6 en la siguiente página vemos la ejecución de esta nueva clase.

Código 9.5 La excepción es cachada y relanzada

```
1: import java.io.*;
2:
3: class CatchExc1  {
4:     public static void main(String[] args)
5:                                     throws Exception  {
6:         int k = 0;
7:         int c;
8:         int i = 3;
9:         int[] enteros = new int[3];
10:
11:        try{
12:            System.out.println("Dame␣un␣entero␣para␣trabajar.");
13:            while((c = System.in.read()) >= '0'  && c <= '9' )
14:                k = k * 10 + (c - ((int)'0'));
15:            System.out.println("Leí␣el␣valor␣"+ k);
16:            c = 1000 / k;
17:            System.out.println("El␣valor␣final␣de␣c␣es␣"+ c);
18:            enteros[i] = c;
19:        }
20:        catch(Exception e)   {
21:            Class en = e.getClass();
22:            System.out.println("La␣excepción␣es␣de␣la␣clase␣:" + en);
23:            throw ((Exception)en.newInstance());
24:        }
25:        System.out.println("A␣punto␣de␣salir␣normalmente␣" +
26:                           "del␣método␣main");
27:    }
28: }
```

Como la clase a la que pertenece la excepción se obtiene con el método getClass() de la clase Class, esto se hace durante ejecución. Si bien se maneja la excepción mediante la cláusula catch de la línea 20:, se vuelve a relanzar, por lo que la línea 25: del programa, que se encuentra a continuación del bloque de la excepción, ya no se ejecuta.

Figura 9.6 Ejecución con relanzamiento de la excepción

```
elisa@lambda ...ICC1/progs/excepciones % java CatchExc1
Dame un entero para trabajar.
0
Leí el valor 0
La excepción es de la clase :class java.lang.Arithmetic
Exception
Exception in thread "main" java.lang.ArithmeticException
       at java.lang.Class.newInstance0(Native Method)
       at java.lang.Class.newInstance(Class.java:237)
       at CatchExc1.main(CatchExc1.java:27)
```

Un método, ya sea de las bibliotecas de clases de Java o del programador, puede lanzar una excepción, y ésta puede ser, nuevamente, una excepción creada por el programador o de las clases de Java. Por ejemplo, si deseamos que no haya divisiones por 0 aún entre reales, podríamos tener las clases que se muestran en los listados 9.6 y 9.7.

Código 9.6 Creación de excepciones propias

```
1: // Archivo:  DivPorCeroException.java
2: public class DivPorCeroException extends ArithmeticException  {
3:     public DivPorCeroException()        {
4:         super();
5:     }
6:     public DivPorCeroException(String msg)   {
7:         super(msg);
8:     }
9: }
```

Código 9.7 Detección de excepciones propias (**DivPorCeroUso**)1/2

```
// Archivo: DivPorCeroUso.java
public class DivPorCeroUso    {
    public static float sqrt(float x)
             throws DivPorCeroException        {
        if (x < 0)
            throw new DivPorCeroException(
              "¡Se␣pide␣raíz␣de␣número␣negativo!");
        else
            return ((float)Math.sqrt(x));
    }
```

Código 9.7 Detección de excepciones propias (**DivPorCeroUso**)2/2

```
    public static void main(String[] args)        {
        // Acá leeríamos un valor para r
        float r = ((float)Math.random()*10000);
        float otro = (float)(Math.random()+.5);
        System.out.println("Otro: "+ otro);
        int signo = (otro > 1 ? 0 : 1);
        if (signo == 0) // Para forzar la excepción
            r = -r;     // Tener r negativo
        // Acá terminamos de obtener un valor para r
        System.out.println("La raíz cuadrada de "+ r + " es "
                                + sqrt(r));
    }
}
```

Como la excepción que construimos, DivPorCeroException, extiende a ArithmeticException y esta última no debe ser vigilada, la invocación a sqrt no tiene que hacerse dentro de un bloque try. Asimismo, el método sqrt no tiene por qué avisar que va a lanzar una excepción.

Sin embargo, si la excepción hereda directamente a Exception o a cualquiera que no sea RuntimeException, el método **no puede dejar de avisar** que va a lanzar una excepción: al compilar a la clase DivPorCeroUso, asumiendo que DivPorCeroException extiende a Exception, da un error que dice

```
javac DivPorCeroUso.java
DivPorCeroUso.java:7: unreported exception DivPorCeroException;
must be caught or declared to be thrown
            throw new DivPorCeroException("¡Se pide raíz de número
negativo!");
            ^
1 error

Compilation exited abnormally with code 1 at Thu Oct 25 17:52:44
javac DivPorCeroUso.java
```

Una vez corregido esto, la llamada al método sqrt tiene que aparecer dentro de un bloque try que se encargue de detectar la excepción que lanza el método. Estos cambios se pueden ver en los listados 9.8 en la siguiente página y 9.9 en la siguiente página.

Código 9.8 Declaración de excepción propia

```
1:    // Archivo: DivPorCeroException.java}
2:    public class DivPorCeroException extends Exception    {
3:        public DivPorCeroException()        {
4:            super();
5:        }
6:        public DivPorCeroException(String msg)        {
7:            super(msg);
8:        }
9:    }
10:  // Archivo: DivPorCeroUso.java
```

Código 9.9 Clase que usa la excepción creada

```
1:        public class DivPorCeroUso        {
2:            public static float sqrt(float x)
3:                                    throws DivPorCeroException  {
4:                if (x < 0)
5:                    throw new DivPorCeroException(
6:                            "¡Se pide raíz de número negativo!");
7:                else
8:                    return ((float)Math.sqrt(x));
9:            }
10:           public static void main(String[] args)  {
11:               // Acá leeríamos un valor para r
12:               float r = ((float)Math.random()*10000);
13:               float otro = (float)(Math.random()+.5);
14:               System.out.println("Otro: "+ otro);
15:               int signo = (otro > 1 ? 0 : 1);
16:               if (signo == 0) // Para forzar la excepción
17:                   r = -r;     // Trabajamos con r negativo
18:               // Acá terminamos de obtener un valor para r
19:               try  {
20:                   System.out.println("La raíz cuadrada de "+ r
21:                                      + " es " + sqrt(r));
22:               }
23:               catch(DivPorCeroException e)   {
24:                   System.out.println(e.getMessage());
25:                   System.exit(-1);
26:               }
27:           }
28:       }
```

Como la excepción se maneja totalmente dentro del método en cuestión, que

en este caso es main, la excepción no se propaga hacia afuera de main, por lo que el método no tiene que avisar que pudiera lanzar una excepción.

Como ya mencionamos antes, las excepciones se pueden lanzar en cualquier momento: sin simplemente un enunciado más. Por supuesto que un uso racional de ellas nos indica que las deberemos asociar a situaciones no comunes o críticas, pero esto último tiene que ver con la semántica de las excepciones, no con la sintaxis.

Tal vez el ejemplo del listado 9.9 en la página opuesta no muestre lo útil que pueden ser las excepciones, porque redefinen de alguna manera una excepción que la JVM lanzaría de todos modos. Pero supongamos que estamos tratando de armar una agenda telefónica, donde cada individuo puede aparecer únicamente una vez, aunque tenga más de un teléfono. Nuestros métodos de entrada, al tratar de meter un nombre, detecta que ese nombre con la dirección ya está registrado. En términos generales, esto no constituye un error para la JVM, pero si para el contexto de nuestra aplicación. Una manera elegante de manejarlo es a través de excepciones, como se muestra en los listados 9.10 a 9.12 en la siguiente página.

Código 9.10 Excepciones del programador (I)

```
1:  // Archivo: RegDuplicadoException.java
2:  public class RegDuplicadoException extends Exception {
3:      public RegDuplicadoException() {
4:          super();
5:      }
6:      public RegDuplicadoException(String msg) {
7:          super(msg);
8:      }
9:  }
```

Código 9.11 Uso de excepciones del programador (I)

```
1:  // Archivo: RegNoEncontradoException.java
2:  public class RegNoEncontradoException extends Exception {
3:      public RegNoEncontradoException() {
4:          super();
5:      }
6:      public RegNoEncontradoException(String msg) {
7:          super(msg);
8:      }
9:  }
```

Código 9.12 Excepciones del programador y su uso (II) (**BaseDeDatos**)1/2

```
1:    // Archivo: BaseDeDatos.java
2:    public class BaseDeDatos    {
3:        int numRegs;
4:        ...
5:        public void agrega(Registro reg)
6:                  throws RegDuplicadoException    {
7:            ...
8:          if (actual.equals(reg))
9:              throw new RegDuplicadoException(reg.nombre);
10:           ...
11:       }
12:       ...
13:       public void elimina(Registro reg)
14:                   throws RegNoEncontradoException  {
15:           ...
16:            if (actual == null)
17:                throw new RegNoEncontradoException(reg.nombre);
18:            ...
19:       }
20:       ...
21:       public static void main(String[] args)   {
22:           ...
23:           while(opcion != 0)   {
24:               try   {
25:                   switch(opcion) {
26:                     case 1: agrega(reg);
27:                             reportaAgregado();
28:                             break;
29:                     case 2: elimina(reg);
30:                             reportaEliminado();
31:                             break;
32:                      ...
33:                       ...
34:                   } // switch
35:               }//try
36:               catch ( RegDuplicadoException e)  {
37:                  // produce un manejo adecuado de la repetición
38:                  System.out.println("El registro de: "
39:                         + e.getMessage()
40:                         + "ya existe, por lo que no se agregó");
41:               } // RegDuplicadoException
```

Código 9.12 Excepciones del programador y su uso (II) (**BaseDeDatos**)2/2

```
42:                 catch ( RegNoEncontradoException e) {
43:                     // Produce un manejo adecuado al no encontrar al
44:                     // nombre
45:                     System.out.println("El registro de: "
46:                         + e.getMessage()
47:                         + "no se encontró, por lo que no se eliminó");
48:                 } // RegNoEncontradoException
49:             } //  while
50:             ...
51:         } // main
52:     } // class
```

El listado 9.12 en la página opuesta hace uso del hecho de que cuando en un bloque se presenta una excepción, la ejecución salta a buscar el manejador de la excepción y deja de ejecutar todo lo que esté entre el punto donde se lanzó la excepción y el manejador seleccionado. Como tanto agrega como elimina lanzan excepciones, su invocación tiene que estar dentro de un bloque try – que va de la línea 24: a la línea 35:. Si es que estos métodos lanzan la excepción, ya sea en la línea 26: o 29:, ya no se ejecuta la línea 27: en el primer caso y la línea 30: en el segundo. Por lo tanto se está dando un control adecuado del flujo del programa, utilizando para ello excepciones.

Otra característica que tiene este segmento de aplicación es que como el bloque try está dentro de una iteración, y si es que se hubiere lanzado una excepción en alguno de los métodos invocados, una vez que se llegó al final del bloque try y habiéndose o no ejecutado alguno de los manejadores de excepciones asociados al bloque try, la ejecución regresa a verificar la condición del ciclo, logrando de hecho que el programa no termine por causa de las excepciones. Esta forma de hacer las cosas es muy común. Supongamos que le pedimos al usuario que teclee un número entero y se equivoca. Lo más sensato es volverle a pedir el dato al usuario para trabajar con datos adecuados, en lugar de abortar el programa.

9.4 Las clases que extienden a Exception

Hasta ahora, cuando hemos declarado clases que extienden a Exception no hemos agregado ninguna funcionalidad a las mismas. No solo eso, sino que úni-

camente podemos usar sus constructores por omisión, los que no tienen ningún parámetro.

El constructor por omisión siempre nos va a informar del tipo de excepción que fue lanzado (la clase a la que pertenece), por lo que el constructor de Exception que tiene como parámetro una cadena no siempre resulta muy útil. Sin embargo, podemos definir una clase tan compleja como queramos, con los parámetros que queramos en los constructores. Únicamente hay que recordar que si definimos alguno de los constructores con parámetros, automáticamente perdemos acceso al constructor por omisión. Claro que siempre podemos invocar a super() en los constructores definidos en las subclases.

Podemos, en las clases de excepciones creadas por el usuario, tener más métodos o información que la que nos provee la clase Exception o su superclase Throwable. Por ejemplo, la clase RegNoEncontradoException que diseñamos para la base de datos pudiera proporcionar más información al usuario que simplemente el mensaje de que no encontró al registro solicitado; podría proporcionar los registros inmediato anterior e inmediato posterior al usuario. En ese caso, debería poder armar estos dos registros. Para ello, podríamos agregar a la clase dos campos, uno para cada registro, y dos métodos, el que localiza al elemento inmediato anterior en la lista y el que localiza al inmediato posterior. En los listados 9.13 y 9.14 en la página opuesta podemos ver un bosquejo de cómo se lograría esto.

Código 9.13 Definición de Excepciones propias (**RegNoEncontradoException**)1/2

```
1: public class RegNoEncontradoException extends Exception  {
2:     private Registro regAnt, regPost;
3:     public RegNoEncontradoException()  {
4:       super();
5:     }
6:
7:     public RegNoEncontradoException(String msg)  {
8:         super(msg);
9:     }
10:
11:    public RegNoEncontradoException(Registro anterior,
12:                                    Registro actual)  {
13:        regAnt = buscaAnterior(padre);
14:        regPost = buscaPosterior(actual);
15:    }
16:
17:    private Registro buscaAnterior(Registro anterior)  {
18:        ...
19:    }
```

Código 9.13 Definición de Excepciones propias (**RegNoEncontradoException**)2/2

```
20:     private Registro buscaPosterior(Actual) {
21:         ...
22:     }
23:
24:     public Registro daRegAnt() {
25:         return regAnt;
26:     }
27:
28:     public Registro daRegPost() {
29:         return regPost;
30:     }
31: }
```

Código 9.14 Definición de excepciones propias (ejemplo) (**Ejemplo**)

```
1: public class Ejemplo {
2:     public static void main(String[] args) {
3:         ...
4:         try {
5:             if(actual.sig == null) {
6:                 throw new
7:                     RegNoEncontradoException(actual.getAnterior(),
8:                                              actual);
9:             } // fin de if
10:            ...
11:        } // fin de try
12:        catch (RegNoEncontradoException e) {
13:            System.out.println("El␣registro␣debió␣encontrarse␣"
14:                               +"entre\n");
15:            System.out.println("***" + e.daRegAnt().daNombre()
16:                               + "***\n" + "␣␣␣␣␣␣y\n");
17:            System.out.println("***"
18:                               + e.daRegPost().daNombre()
19:                               + "***");
20:        } // fin de catch
21:        ...
22:    } // main
23: } // class
```

En las líneas 6: a 8: tenemos un constructor adicional declarado para nuestra clase que extiende a Exception. Como se puede ver en las líneas 13: a 17:, el manejador de la excepción hace uso de los campos y métodos declarados en la

excepción, y que son llenados por el constructor, para proveer más información al usuario de la situación presente en el momento de la excepción. Haciéndolo de esta manera, en el momento de lanzar la excepción se puede invocar un constructor que recoja toda la información posible del contexto en el que es lanzada, para reportar después en el manejador.

Únicamente hay que recordar que todas aquellas variables que sean declaradas en el bloque try no son accesibles desde fuera de este bloque, incluyendo a los manejadores de excepciones. Insistimos: si se desea pasar información desde el punto donde se lanza la excepción al punto donde se maneja, lo mejor es pasarla en la excepción misma. Esto último se consigue redefiniendo y extendiendo a la clase Exception.

Además de la información que logremos guardar en la excepción, tenemos también los métodos de Throwable, como el que muestra el estado de los registros de activación en el stack, o el que llena este stack en el momento inmediato anterior a lanzar la excepción. Todos estos métodos se pueden usar en las excepciones creadas por el programador.

Veamos en los listados 9.15 y 9.16 otro ejemplo de declaración y uso de excepciones creadas por el programador. En este ejemplo se agrega un constructor y un atributo que permiten a la aplicación recoger información respecto al orden en que se lanzan las excepciones y el contexto en el que esto sucede.

Código 9.15 Excepciones creadas por el programador (**MiExcepcion2**)

```
1:     // Adecuación de las excepciones
2: class MiExcepcion2 extends Exception  {
3:     private int i;
4:     public MiExcepcion2()  {
5:         super();
6:     }
7:     public MiExcepcion2(String msg)   {
8:         super(msg);
9:     }
10:    public MiExcepcion2(String msg, int x)  {
11:        super(msg);
12:        i = x;
13:    }
14:    public int val()  {
15:        return i;
16:    }
17: }
```

Código 9.16 Uso de excepciones creadas por el programador (**CaracteristicasExtra**)

```
18: public class CaracteristicasExtra {
19:     public static void f() throws MiExcepcion2  {
20:         System.out.println("Lanzando␣MiExcepcion2␣desde␣f()");
21:         throw new MiExcepcion2();
22:     }
23:     public static void g() throws MiExcepcion2    {
24:         System.out.println("Lanzando␣MiExcepcion2␣desde␣g()");
25:         throw new MiExcepcion2("Se␣originó␣en␣g()");
26:     }
27:     public static void h() throws MiExcepcion2    {
28:         System.out.println("Lanzando␣MiExcepcion2␣desde␣h()");
29:         throw new MiExcepcion2("Se␣originó␣en␣h()", 47);
30:     }
31:     public static void main(String[] args)    {
32:         try   {
33:             f();
34:         }
35:         catch (MiExcepcion2 e)   {
36:             e.printStackTrace(System.err);
37:         }
38:         try   {
39:             g();
40:         }
41:         catch  (MiExcepcion2 e)   {
42:             e.printStackTrace(System.err);
43:         }
44:         try   {
45:             h();
46:         }
47:         catch (MiExcepcion2 e)   {
48:             e.printStackTrace(System.err);
49:             System.err.println("e.val()␣=␣" + e.val());
50:         }
51:     }
52: }
```

En esta aplicación se muestra el uso de distintos constructores, las invocaciones a los métodos de Throwable y la extensión de la información que provee la clase agregando campos. El resultado de la ejecución se puede ver en la figura 9.7 en la siguiente página.

Figura 9.7 Ejecución de **CaracteristicasExtra**

```
elisa@lambda ...ICC1/progs/excepciones % java CaracteristicasExtra
Lanzando MiExcepcion2 desde f()
MiExcepcion2
        at CaracteristicasExtra.f(CaracteristicasExtra.java:23)
        at CaracteristicasExtra.main(CaracteristicasExtra.java:38)
Lanzando MiExcepcion2 desde g()
MiExcepcion2: Se originó en g()
        at CaracteristicasExtra.g(CaracteristicasExtra.java:28)
        at CaracteristicasExtra.main(CaracteristicasExtra.java:44)
Lanzando MiExcepcion2 desde h()
MiExcepcion2: Se originó en h()
        at CaracteristicasExtra.h(CaracteristicasExtra.java:33)
        at CaracteristicasExtra.main(CaracteristicasExtra.java:50)
e.val() = 47
```

Recalcando lo que ya vimos respecto a excepciones, notamos varias cosas en este listado:

- Los métodos f(), g() y h() tienen que avisar que lanzan una excepción, ya que MiExcepcion2 no hereda de RuntimeException y por lo tanto se debe vigilar cuando se ejecute cualquiera de estos tres métodos. Vale la pena decir que aunque el lanzamiento de la excepción fuera condicional, de cualquier manera el método tendría que avisar que existe la posibilidad de que lance la excepción.
- Como los métodos lanzan excepciones, cada uno de ellos tiene que ser invocado en un bloque try.
- Como el bloque try consiste únicamente de la invocación al método, una vez ejecutado el manejador de la excepción que se encuentra a continuación del respectivo catch, la ejecución continúa en la siguiente línea de código. Es por ello que aunque se lancen las excepciones, la ejecución continúa una vez ejecutado el manejador.
- Si alguno de los métodos lanzaran alguna otra excepción, el compilador exigiría que hubiera un manejador por cada tipo de excepción. Se puede cachar excepciones usando superclases, pero cada clase de excepción lanzada por un método tiene que tener su manejador propio o uno que se refiera a la superclase.
- Si un método lanza una excepción y no la cacha en el mismo método, su encabezado tiene que especificar que lanza aquellas excepciones que no sean

cachadas en el mismo método.

9.4.1. Relanzamiento de excepciones

Muchas veces el manejador de una excepción hace algo de administración de la clase y después de esto simplemente vuelve a lanzar la excepción. Si se le va a pedir a la excepción que reporte el punto donde estaba la ejecución en el momento en que fue lanzada la excepción – usando printStackTrace – la excepción lanzada va a tener registro del punto donde fue creada, no del punto desde donde es finalmente lanzada. Para que la excepción actualice su información respecto al stack de ejecución se utiliza el método fillInStackTrace al momento de relanzar la excepción; esto va a hacer que el stack refleje el último punto donde la excepción fue lanzada y no donde fue creada.

9.5 El enunciado finally

Cuando tenemos un programa en el que estamos vigilando el lanzamiento de excepciones, vamos a tener código que, por encontrarse después del punto donde se lanzó la excepción y dentro del bloque try, no va a ser ejecutado. Por ejemplo, si estoy tratando de asignarle un valor a una variable y la ejecución no pasa por ese enunciado porque antes se lanzó una excepción.

A continuación de los bloques correspondientes a cachar las excepciones – los bloques catch – podemos escribir un bloque de enunciados que se van a ejecutar ya sea que se haya lanzado una excepción o no en el bloque try. La cláusula finally *siempre* se ejecuta, no importa que se haya lanzado o no una excepción. Veamos un ejemplo muy sencillo en el listado 9.18 en la siguiente página.

Código 9.17 Ejemplo con la cláusula **finally** (Excepción)

```
1: class TresException extends Exception   {
2:     public TresException()  {
3:         super();
4:     }
5:     public TresException(String msg)  {
6:         super(msg);
7:     }
8: }
```

Código 9.18 Ejemplo con la cláusula **finally** (uso)

```
 9: public class FinallyTrabaja  {
10:     static int cuenta = 0;
11:     public static void main(String[] args)    {
12:         while ( true )    {
13:             try {
14:                 // Post-incremento. Es cero la primera vez
15:                 if (cuenta++ == 0) {
16:                     throw new TresException();
17:                 }
18:                 System.out.println("No hubo excepción");
19:             }
20:             catch  (TresException e)    {
21:                 System.err.println("TresException");
22:             }
23:             finally    {
24:                 System.err.println("En la cláusula finally");
25:                 if (cuenta == 2)  {
26:                     break;   // sal del while
27:                 }
28:             }
29:         }
30:     }
31: }
```

Como se puede ver la salida de la ejecución de este algoritmo en la figura 9.8, el mensaje mandado por el bloque finally se imprime siempre, sin importar si hubo o no excepción.

Figura 9.8 Ejecución de **FinallyTrabaja**

```
elisa@lambda ...ICC1/progs/excepciones % java FinallyTrabaja
TresException
En la cláusula finally
No hubo excepción
En la cláusula finally
```

Es interesante también notar cómo, aunque se lance una excepción, como el bloque try está dentro de una iteración, al salir de ejecutar todo el bloque asociado a la excepción, la ejecución continúa con el while.

finally funciona como una tarea que sirve para dar una última pasada al código, de tal manera de garantizar que todo quede en un estado estable. No siempre es necesario, ya que Java cuenta con recolección automática de basura y destructores de objetos también automáticos. Sin embargo, se puede usar para agrupar tareas que se desean hacer, por ejemplo en un sistema guiado por excepciones, ya sea que se presente un tipo de excepción o no. Veamos un ejemplo con unos interruptores eléctricos en el listado 9.19.

Código 9.19 Otro ejemplo con la cláusula **finally** (**Switch**)

```
1: class Switch   {
2:     boolean state = false;
3:     boolean read()  {
4:       return state;
5:     }
6:     void on()  {
7:       state = true;
8:     }
9:     void off() {
10:      state = false;
11:    }
12: }
```

Código 9.20 Otro ejemplo con la cláusula **finally** (**OnOffException1**)

```
1: class OnOffException1 extends Exception      {
2:     public OnOffException1()  {
3:         super();
4:     }
5:     public OnOffException1(String msg)  {
6:         super(msg);
7:     }
8: }
```

Código 9.21 Otro ejemplo con la cláusula **finally** (**OnOffException2**)

```
1: class OnOffException2 extends Exception      {
2:     public OnOffException2()  {
3:         super();
4:     }
5:     public OnOffException2(String msg)  {
6:         super(msg);
7:     }
8: }
```

Código 9.22 Otro ejemplo con la cláusula **finally** (**OnOffSwitch**)

```
1: class OnOffSwitch  {
2:     static Switch sw = new Switch();
3:     static void f()
4:          throws OnOffException1, OnOffException2 {
5:     }
6: }
```

Código 9.23 Otro ejemplo con la cláusula **finally** (**ConFinally**)

```
1: public class ConFinally  {
2:     static Switch sw = new Switch();
3:     public static void main(String[] args)  {
4:         try     {
5:             sw.on();
6:             //  Código que puede lanzar excepciones
7:             OnOffSwitch.f();
8:         }
9:         catch (OnOffException1 e)     {
10:            System.err.println("␣OnOffException1");
11:        }
12:        catch (OnOffException2 e)     {
13:            System.err.println("␣OnOffException2");
14:        }
15:        finally {
16:            sw.off();
17:        }
18:    }
19: }
```

En esta aplicación deseamos que, ya sea que se haya podido o no prender el interruptor, la aplicación lo apague antes de salir.

Los bloques try se pueden anidar para colocar de mejor manera las cláusulas finally, obligando a ejecutar de adentro hacia afuera. En el listado 9.24 tenemos un ejemplo de anidamiento de bloques try.

Código 9.24 Anidamiento de bloques **try** (**CuatroException**)

```
1: class CuatroException extends Exception  {
2:     public CuatroException()  {
3:         super();
4:     }
5:     public CuatroException(String msg)  {
6:         super(msg);
7:     }
8: }
```

Código 9.25 Anidamiento de bloques **try** (**SiempreFinally**)

```
1: public class SiempreFinally  {
2:     public static void main(String[] args)  {
3:         System.out.println("Entrando␣al␣primer␣bloque␣try");
4:         try     {
5:             System.out.println("Entrando␣al␣segundo␣bloque␣try");
6:             try     {
7:                 throw new CuatroException();
8:             }
9:             finally     {
10:                System.out.println("Finally␣en␣el␣segundo"
11:                                   + "bloque␣try");
12:            }
13:        }
14:        catch (CuatroException e)     {
15:            System.err.println("Cachando␣CuatroException␣en␣" +
16:                                                "el␣primer␣bloque␣try");
17:        }
18:        finally     {
19:            System.err.println("Finally␣en␣primer␣bloque␣try");
20:        }
21:    }
22: }
```

Como en la mayoría de los casos, la cláusula finally se ejecuta de adentro hacia afuera. No importa que el primer try no tenga manejador para la excepción, porque al lanzarse la excepción y no encontrar un manejador en su entorno inmediato, simplemente va a salir y utilizar el manejador del bloque try más externo. El resultado de la ejecución se puede ver en la figura 9.9.

Figura 9.9 Ejecución de **SiempreFinally**

```
elisa@lambda ...ICC1/progs/excepciones % java SiempreFinally
Entrando al primer bloque try
Entrando al segundo bloque try
Finally en el segundo bloque try
Cachando CuatroException en el primer bloque try
Finally en primer bloque try
```

9.6 Restricciones para las excepciones

Cuando se redefine el método de una clase, el método redefinido no puede lanzar más excepciones (o distintas) que el método original. Esto es para que si alguien usa herencia para manejar ciertos objetos, no resulte que el método en la superclase ya no funciona porque el método en la subclase lanza más excepciones que el original. Esto es, si un método en la superclase no lanza excepciones, ese método redefinido en las subclases tampoco puede lanzar excepciones.

Lo que si puede hacer un método redefinido es lanzar excepciones que resultan de extender a las excepciones que lanza el método de la superclase. En este caso no hay problema. Revisemos, por ejemplo, la aplicación del listado 9.26.

Código 9.26 Manejo de excepciones con herencia 1/3

```
1: class BaseballException extends Exception {
2: }
3: class Foul extends BaseballException {
4: }
5: class Strike extends BaseballException {
6: }
7:
8: abstract class Inning {
9:     Inning() throws BaseballException {
10:    }
11:    void event() throws BaseballException {
12:    }
13:    abstract void atBat() throws Strike, Foul;
14:    void walk() {
15:    }
16: }
17:
18: class StormException extends Exception {
19: }
20: class RainedOut extends StormException {
21: }
22: class PopFoul extends Foul {
23: }
24:
25: interface Storm {
26:    void event() throws RainedOut;
27:    void rainHard() throws RainedOut;
28: }
```

Código 9.26 Manejo de excepciones con herencia (**StormyInning**)2/3

```
29: public class StormyInning extends Inning implements Storm {
30:     /* OK añadir nuevas excepciones para los constructores, pero se
31:        deben manejar las excepciones de los constructores base:
32:      */
33:     StormyInning() throws RainedOut, BaseballException {
34:     }
35:
36:     StormyInning(String s) throws Foul, BaseballException {
37:     }
38:       /* Los métodos normales se tienen que adaptar a la clase base:
39:        * ! void walk() throws PopFoul {}  // Compile Error
40:        * Interface CANNOT add exceptions to existing methods from the
41:        * base class:
42:        * ! public void event() throws RainedOut { }
43:        * Si el método no existe en la clase base, entonces la
44:        * excepción se vale:
45:        */
46:     public void rainHard() throws RainedOut    {
47:     }
48:
49:          /* Puedes elegir no lanzar ninguna excepción, aún cuando la
50:             versión base sí lo haga:
51:           */
52:     public void event()  {
53:     }
54:
55:     /* Los métodos que redefinen a métodos básicos pueden lanzar
56:        excepciones heredadas:
57:      */
58:     void atBat() throws PopFoul  {
59:     }
60:     public static void main(String[] args)    {
61:         try    {
62:             StormyInning si = new StormyInning();
63:             si.atBat();
64:         } catch(PopFoul e)    {
65:             System.err.println("PopFoul");
66:         } catch(RainedOut e)    {
67:             System.out.println("Rained␣out");
68:         } catch(BaseballException e)    {
69:             System.out.println("Generic␣Error");
70:         }
```

Código 9.26 Manejo de excepciones con herencia (**StormyInning**)3/3

```
71:
72:         // En la versión derivada no se lanza un Strike
73:         try     {
74:             // ¿Qué pasa si se "upcast"?
75:             Inning i = new StormyInning();
76:             i.atBat();
77:             // Debes cachar las excepciones del método de la versión
78:             // en la clase base:
79:         }   catch(Strike e)     {
80:             System.out.println("Strike");
81:         }   catch(Foul e)     {
82:             System.out.println("Foul");
83:         }  catch(RainedOut e)     {
84:             System.out.println("Rained␣Out");
85:         }  catch(BaseballException e)   {
86:             System.out.println("Excepción␣Genérica␣Badallé");
87:         }
88:     }
89: }
```

9.6.1. Apareamiento de excepciones

En general, el manejador de una excepción se va a ejecutar en cualquiera de las situaciones siguientes:

- La clase a la que pertenece la excepción aparece en una cláusula catch que corresponde al bloque try en el que se lanzó la excepción.
- Alguna de las superclases de la excepción lanzada aparece en una cláusula catch que corresponde al bloque try en el que se lanzó la excepción.

Cualquiera de estas dos situaciones que se presente, se ejecutará el manejador que aparezca primero. Si en la lista de manejadores aparecen tanto la superclase como la clase, y la superclase aparece primero, el compilador dará un mensaje de error de que el segundo manejador nunca puede ser alcanzado. En la aplicación StormyInning del listado 9.26 se pueden observar todas las combinaciones de este tipo. Los comentarios ilustran también algunos puntos que no son válidos.

9.7 Recomendaciones generales

Las excepciones en general se usan en cualquiera de las siguientes circunstancias:

i. Arreglar el problema y llamar otra vez al método que causó la excepción.

ii. Parchar el proceso y continuar sin volver a intentar el método.

iii. Calcular algún resultado alternativo en lugar del que el método se supone que debía haber calculado.

iv. Hacer lo que se pueda en el contexto actual y relanzar la excepción para que sea manejada en un contexto superior.

v. Hacer lo que se pueda en el contexto actual y lanzar una excepción distinta para que sea manejada en un contexto superior.

vi. Terminar el programa.

vii. Simplificar el algoritmo.

viii. Hacer una aplicación (o biblioteca) más segura (se refleja a corto plazo en la depuración y a largo plazo en la robustez de la aplicación).

Con esto damos por terminado este tema, aunque lo usaremos extensivamente en los capítulos que siguen.

Entrada y salida 10

10.1 Conceptos generales

Uno de los problemas que hemos tenido hasta el momento es que las bases de datos que hemos estado construyendo no tienen *persistencia*, esto es, una vez que se descarga la aplicación de la máquina virtual (que termina) la información que generamos no vive más allá. No tenemos manera de almacenar lo que construimos en una sesión para que, en la siguiente sesión, empecemos a partir de donde nos quedamos. Prácticamente en cualquier aplicación que programemos y usemos vamos a requerir de mecanismos que proporcionen persistencia a nuestra información.

En los lenguajes de programación, y en particular en Java, esto se logra mediante *archivos*[1], que son conjuntos de datos guardados en un medio de almacenamiento externo. Los archivos sirven de puente entre la aplicación y el medio exterior, ya sea para comunicarse con el usuario o para, como acabamos de mencionar, darle persistencia a nuestras aplicaciones.

Hasta ahora hemos usado extensamente la clase Consola, que es una clase programada por nosotros. También hemos usado en algunos ejemplos del capítulo anterior dos archivos (objetos) que están dados en Java y que son System.out y System.err. Ambos archivos son de salida; el primero es para salida normal en consola y el segundo para salida, también en consola, pero de errores. Por ejemplo, cuando un programa que aborta reporta dónde se lanzó la excepción, el reporte

lo hace a System.err.

La razón por la que usamos nuestra propia clase hasta el momento es que en Java prácticamente toda la entrada y salida puede lanzar excepciones; eso implica que cada vez que usemos un archivo para leer, escribir, crearlo, eliminarlo, y en general cualquier operación que tenga que ver con archivos, esta operación tiene que ser vigilada en un bloque try, con el manejo correspondiente de las excepciones que se pudieran lanzar. Lo que hace nuestro paquete de entrada y salida es absorber todas las excepciones lanzadas para que cuando usan los métodos de estas clases ya no haya que vigilar las excepciones.

El diseñar los métodos de entrada y salida para que lancen excepciones en caso de error es no sólo conveniente, sino necesario, pues es en la interacción con un usuario cuando la aplicación puede verse en una situación no prevista, como datos erróneos, un archivo que no existe o falta de espacio en disco para crear un archivo nuevo.

Un concepto muy importante en la entrada y salida de Java es el de *flujos* de datos. Java maneja su entrada y salida como flujos de carácteres (ya sea de 8 o 16 bits). En el caso de los flujos de entrada, éstos proporcionan carácteres, uno detrás de otro en forma secuencial, para que el programa los vaya consumiendo y procesando. Los flujos de salida funcionan de manera similar, excepto que es el programa el que proporciona los carácteres para que sean proporcionados al mundo exterior, también de manera secuencial.

En las figuras 10.1 y 10.2 en la página opuesta vemos los algoritmos generales para lectura y escritura, no nada más para Java, sino que para cualquier lenguaje de programación.

Figura 10.1 Algoritmo para el uso de flujos de entrada

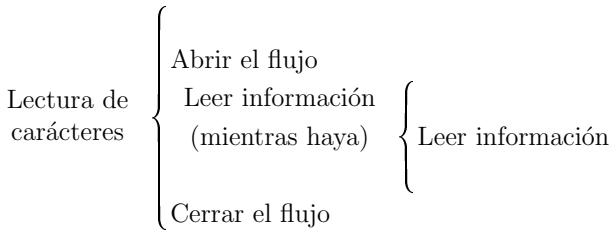
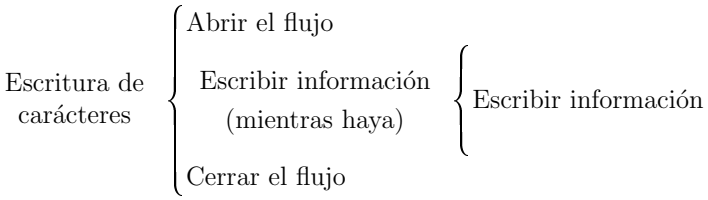


Figura 10.2 Algoritmo para el uso de flujos de salida



Podemos ver un esquema de este funcionamiento en las figuras 10.3 y 10.4.

Figura 10.3 Funcionamiento de flujo de entrada

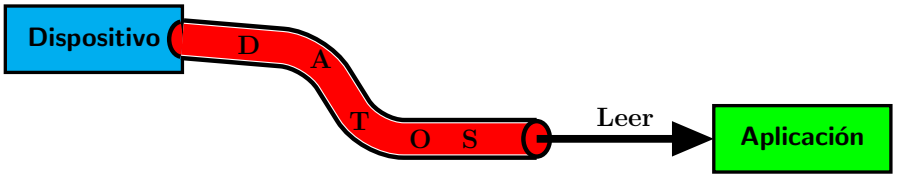
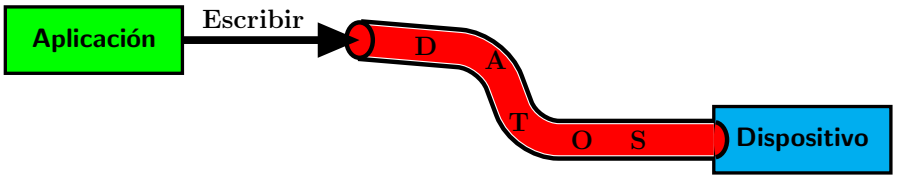


Figura 10.4 Funcionamiento de flujo de salida



Los flujos de entrada se manejan a través de clases específicas para ellos. Al construir el objeto se abre el flujo; se lee de él o escribe en él utilizando los distintos métodos que tenga la clase para ello; se cierra invocando al método close del objeto correspondiente.

En lo que sigue elaboraremos métodos para hacer persistente nuestra base de datos. Antes trataremos de tener una visión más general de cómo maneja Java la entrada y salida.

10.2 Jerarquía de clases

La entrada y salida se maneja en Java a través de una jerarquía que incluye clases e interfaces. Tenemos básicamente dos maneras de hacer entrada y salida: la primera es leyendo y escribiendo bytes, mientras que la segunda es leyendo y escribiendo carácteres Unicode. Dado que Java es, fundamentalmente, un lenguaje cuya característica principal es su portabilidad, se diseñó un juego de carácteres universales, de dos bytes cada uno, que cubre prácticamente todos los alfabetos conocidos. Para asegurar la portabilidad de datos, y dado que Java maneja internamente Unicode, es que se diseñaron estas clases que manejan carácteres.

Cada uno de los tipos de entrada y salida tiene una superclase abstracta para lectura y otra para escritura. De ella se derivan clases concretas que permiten manipular de alguna forma lo que se está leyendo o escribiendo. Iremos describiendo su uso conforme las vayamos presentando.

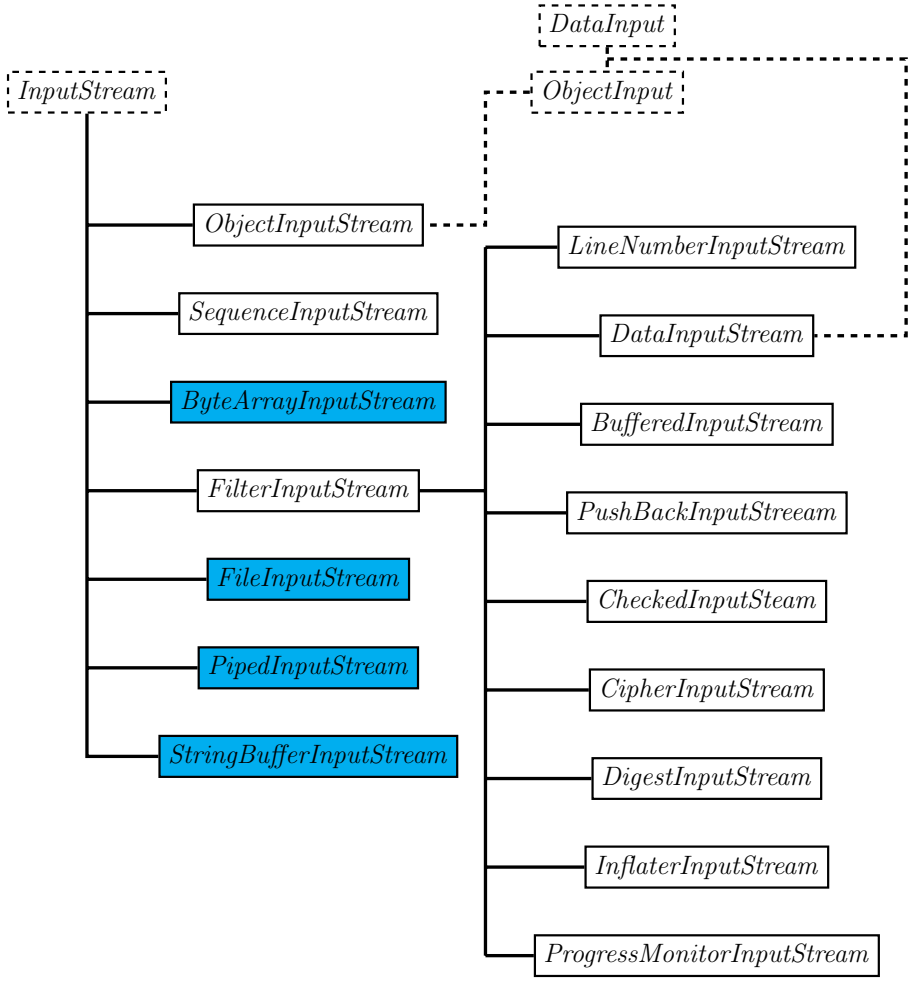
La entrada y salida se ve siempre como un flujo, ya sea de bytes o de carácteres. Se va tomando unidad por unidad y se procesa. Cuando este flujo se termina decimos que se acabó el archivo y tendremos un eof (fin de archivo). Generalmente procesaremos la información hasta encontrar un eof, en cuyo momento daremos fin al proceso de los datos. Es por esta característica que Java llama a su entrada y salida streams. Hablaremos de un flujo de bytes o de un flujo de carácteres (omitiendo Unicode).

10.3 Entrada y salida de bytes

Un byte es un entero que ocupa 8 bits, y en general se da como unidad de medida para otros tipos que ocupan más espacio. Los enteros que podemos almacenar en un byte van del -128 al 127. Sin embargo, cuando pensamos en código ASCII, pensamos en carácteres cuyo valor está entre 0 y 255. Para que podamos manejar así los bytes, la lectura (y la escritura) se hará siempre en enteros o en carácteres Unicode, de tal manera que el método utilice únicamente el byte más bajo (al que corresponden las posiciones más de la derecha).

En la figura 10.5 en la página opuesta se encuentra la jerarquía de clases para InputStream, mientras que en la figura 10.6 en la página 328 está el esquema de la jerarquía de clases para OutputStream.

Figura 10.5 Jerarquía de clases para InputStream.



A continuación damos una muy breve explicación en orden alfabético de cuál es el uso de cada una de las subclases para entrada. Todas las subclases se encuentran en el paquete java.io, excepto cuando indiquemos explícitamente que no es así.

public class BufferedInputStream **extends** FilterInputStream

Lee desde un InputStream guardando lo que va leyendo en un buffer. Esto permite a la aplicación marcar una posición o regresar a bytes ya leídos.

public class ByteArrayInputStream **extends** InputStream

Contiene un buffer interno que contiene bytes, que se leen del flujo, cuando es necesario.

public class CheckedInputStream **extends** FilterInputStream

(Paquete: java.util.zip)

Es un InputStream que mantiene una suma de verificación (*checksum*) de los datos que ha leído.

public class CipherInputStream **extends** FilterInputStream

(Paquete: javax.crypto)

Está compuesto de un InputStream y un Cipher que permite entregar cifrados los bytes que lee de la entrada.

public class DataInputStream **extends** FilterInputStream **implements** DataInput

Lee datos primitivos (enteros, reales, booleanos, etc.) de un InputStream subyacente de manera independiente de la máquina.

public class FileInputStream **extends** InputStream

El flujo de entrada reside en un archivo en disco.

public class FilterInputStream

Simplemente recibe el flujo de un flujo subyacente y los pasa a la aplicación. Redefine los métodos de entrada para poder "transformarla".

public class DigestInputStream **extends** FilterInputStream

(paquete: java.security)

Actualiza el mensaje digerido (MesasageDigest) usando para ello los bytes que pasan por el flujo.

public class InflaterInputStream **extends** FilterInputStream

(Paquete: java.util.zip)

Implementa un filtro para descomprimir datos comprimidos con *deflate* y para otros filtros de descompresión.

public abstract class InputStream **implements** Closeable

Es la superclase de todas las clases que manejan flujos de entrada de bytes.

public class LineNumberInputStream **extends** FilterInputStream

(**sobreseido**[2])

Este flujo lleva la cuenta del número de líneas que ha leído. Una línea es una sucesión de bytes que terminan con \r, \n o un retorno de carro seguido de una alimentación de línea. Al entregar las líneas leídas convierte cualquiera de los terminadores a \n.

class ObjectInputStream **extends** InputStream
implements ObjectInput, ObjectStreamConstants

Escribe datos primitivos y gráficas de objetos de Java en un flujo de bytes. Únicamente objetos que implementen la interfaz Serializable pueden ser escritos en este flujo.

public class PipedInputStream **extends** InputStream

Se usa con hilos paralelos de ejecución, donde uno de los hilos usa un PipedInputStream para adquirir datos y el otro usa un PipedOutputStream para entregar datos. Ambos hilos pueden hacer un proceso de la información del flujo correspondiente.

public class ProgressMonitorInputStream **extends** FilterInputStream

(Paquete: javax.swing)

Vigila el progreso al leer de un InputStream, presentando, en su caso, ventanas de diálogo.

public class PushBackInputStream **extends** FilterInputStream

Trabaja sobre un InputStream subyacente y permite "desleer" un byte. Este proceso es útil cuando estamos buscando, por ejemplo, un byte que tiene dos funciones: delimitar lo que está antes y empezar lo nuevo. En este caso es conveniente desleerlo para el segundo papel que juega.

public class SequenceInputStream **extends** InputStream

Es capaz de concatenar, para lectura, a varios flujos de entrada como si fueran uno solo.

public class StringBufferInputStream **extends** InputStream

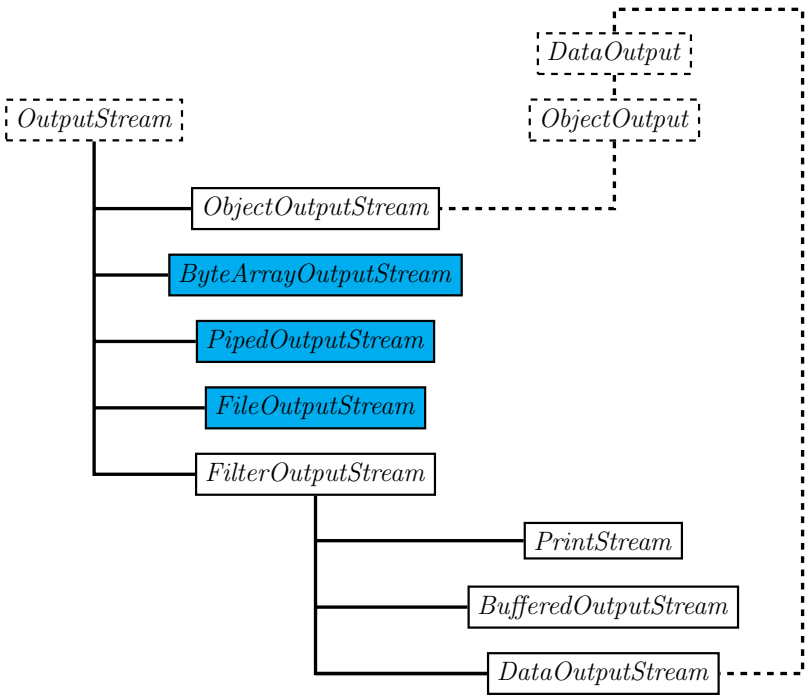
Permite crear una aplicación en la que el flujo proviene de una cadena de carácteres, en lugar de venir de un dispositivo.

La jerarquía de clases para los flujos de salida de bytes se da en la figura 10.6.

[2]Corresponde a *deprecated*, que indica que no se recomienda su uso porque ya no va a ser actualizada y soportada

Con sus marcadas excepciones, por el uso que se le pueda dar, hay una correspondencia entre ambas jerarquías.

Figura 10.6 Jerarquía de clases para OutputStream.



La única clase que no tiene contra parte en la jerarquía de entrada de bytes es PrintStream:

---

public class PrintStream **extends** FilterOutputStream
implements Appendable

Agrega funcionalidad a otro OutputStream aportando la habilidad de imprimir de manera conveniente diversos valores de datos. Adicionalmente, sus métodos nunca lanzan excepciones y puede construirse de tal manera que evacúe automáticamente.

10.4 Entrada y salida de carácteres

Cuando hablamos de carácteres en el contexto de Java nos referimos a carácteres Unicode, donde cada uno ocupa 2 bytes (16 bits). Tenemos una jerarquía similar a la que maneja bytes para carácteres Unicode, las superclases Writer y Reader, cuyas jerarquías se muestra en las figuras 10.7 y 10.8 respectivamente. En todos los casos las subclases sombreadas se refieren a clases que van a hacer un proceso intermedio de los datos entre el origen y el destino de los mismos. Un esquema de qué significa esto se encuentra en la figura 10.14. En ésta el origen de los datos puede ser la aplicación y el destino el dispositivo, en el caso de que se esté efectuando escritura; o bien el origen es el dispositivo que entrega los datos "en crudo" y la aplicación la que los recibe en el destino ya procesados.

Figura 10.7 Jerarquía de clases para Writer.

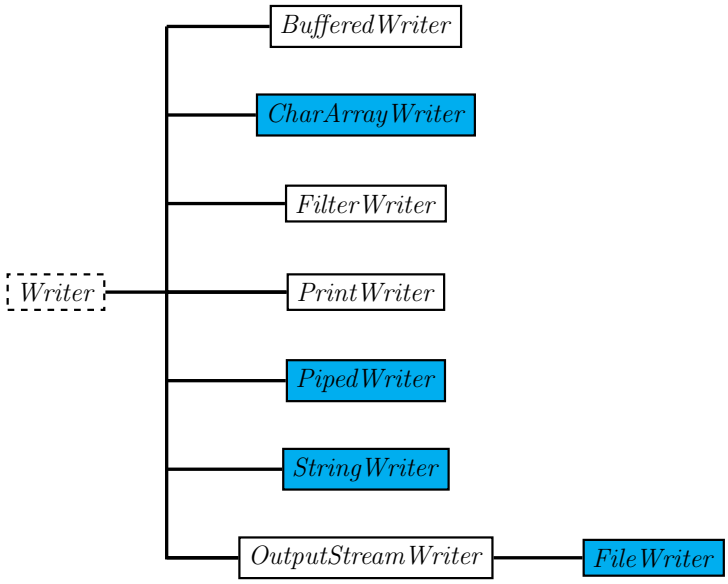


Figura 10.8 Jerarquía de clases para Reader.

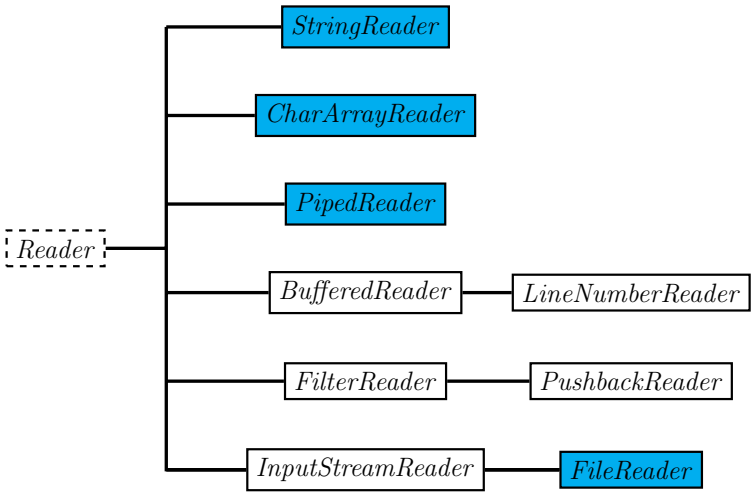
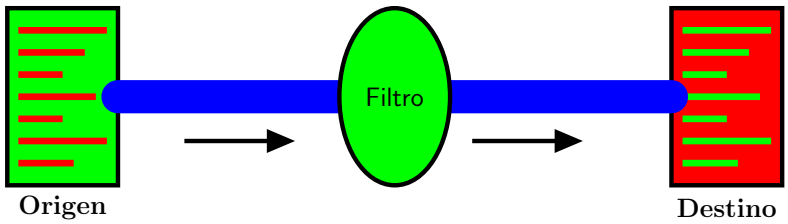


Figura 10.9 Entrada/Salida con proceso intermedio (filtros)



También estas jerarquías corren paralelas a las que trabajan con bytes, por lo que no daremos una nueva explicación de cada una de ellas. Se aplica la misma

explicación, excepto que donde dice "byte" hay que sustituir por "carácter". Únicamente explicaremos aquellas clases que no tienen contra parte en bytes.

public class StringWriter **extends** Writer
Escribe su salida en un buffer de tipo String, que puede ser utilizado a su vez para construir una cadena.
public class OutputStreamWriter **extends** Writer
Funciona como un puente entre flujos de carácteres y flujos de bytes, codificados de acuerdo a un conjunto de carácteres específico.

Vale la pena hacer la aclaración que en este caso los flujos que leen de y escriben a archivos en disco extienden a las clases InputStreamReader y OutputStreamWriter respectivamente, ya que la unidad de trabajo en los archivos es el byte (8 bits) y no el carácter (16 bits). Por lo demás funcionan igual que sus contra partes en los flujos de bytes.

Es conveniente mencionar que las versiones actuales de Java indican que las clases que se deben usar son las que derivan de Reader y Writer y no las que son subclases de InputStream y OutputStream. Ambas jerarquías (las de bytes y las de carácteres) definen prácticamente los mismos métodos para bytes y carácteres, pero para fomentar la portabilidad de las aplicaciones se ha optado por soportar de mejor manera las clases relativas a carácteres.

Sin embargo, como ya mencionamos, la entrada y salida estándar de Java es a través de clases que pertenecen a la jerarquía de bytes (System.in, System.out y System.err).

Lo primero que queremos poder hacer es leer desde el teclado y escribir a pantalla. Esto lo necesitamos para la clase que maneja el menú y de esta manera ir abriendo las cajas negras que nos proporcionaba la clase Consola para este fin.

10.4.1. Entrada y salida estándar

La entrada y salida desde teclado y hacia consola se hace a través de la clase System. Esta clase ofrece, además de los objetos para este tipo de entrada y salida, muchísimos métodos que van a ser útiles en aplicaciones en general. La clase System tiene tres atributos que son:

```
public static final PrintStream err;
public static final InputStream in;
public static final PrintStream out;
```

Por ser objetos estáticos de la clase se pueden usar sin construirlos. Todo programa en ejecución cuenta con ellos, por lo que los puede usar, simplemente refiriéndose a ellos a través de la clase System.

El primero de ellos es un archivo al que dirigiremos los mensajes que se refieran a errores, y que no queramos "mezclar" con la salida normal. El segundo objeto es para leer de teclado (con eco en la pantalla) y el tercero para escribir en la pantalla. Las dos clases mencionadas son clases concretas que aparecen en la jerarquía de clases que mostramos en las figuras 10.5 en la página 325 y 10.6 en la página 328.

Si bien la clase PrintStream se va a comportar exactamente igual a Consola, en cuanto a que "interpreta" enteros, cadenas, flotantes, etc. para mostrarlos con formato adecuado, esto no sucede con la clase InputStream que opera de manera muy primitiva, leyendo byte por byte, y dejándole al usuario la tarea de pegar los bytes para interpretarlos. Más adelante revisaremos con cuidado todos los métodos de esta clase. Por el momento únicamente revisaremos los métodos que leen byte por byte, y que son:

public class InputStream implements Closeable

Constructores:

public InputStream()

Constructor por omisión

Métodos:

public int read() **throws** IOException

Lee el siguiente byte del flujo de entrada. Devuelve un valor entre 0 y 255. Si se acaba el archivo (o desde el teclado se oprime Ctrl-D) regresa -1.

public int read(**byte**[] b) **throws** IOException

Lee un número de bytes al arreglo. Regresa el número de bytes leído. Bloquea la entrada hasta tener datos de entrada disponibles, se encuentre el fin de archivo o se lance una excepción. Se leen, a lo más, el número de bytes dado por el tamaño de b.

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

Lee a lo más len bytes de datos desde el flujo de entrada y los acomoda en el arreglo de bytes b. Regresa el número de bytes leídos. El primer byte leído se acomoda en b[off]. Lanza una excepción IndexOutOfBoundsException si off es negativo, len es negativo o off+len >= b.length.

Como podemos ver de los métodos de la clase InputStream, son muy primitivos y difíciles de usar. Por ello, como primer paso en la inclusión de entrada y salida completa en nuestra aplicación, para entrada utilizaremos una subclase de Reader, BufferedReader, más actual y mejor soportada.

10.5 El manejo del menú de la aplicación

En el caso de los flujos System.out y System.err no tenemos que hacer absolutamente nada pues existen como atributos estáticos de la clase System, por lo que los podemos usar directamente. Conviene, sin embargo, listar los métodos y atributos de la clase PrintStream, que es una subclase de FilterOutputStream, que es, a su vez, una subclase de OutputStream.

10.5.1. La clase OutputStream

Esta es una clase abstracta que deja sin implementar uno de sus métodos. El constructor y los métodos se listan a continuación:

public class OutputStream implements Closeable, Flushable

Constructores:

public OutputStream()

Constructor por omisión.

Métodos:

public abstract void write(**int** b) **throws** IOException

Toma el entero b y escribe únicamente los 8 bits más bajos, descartando los otros 24 bits. El programador de clases que hereden de ésta tienen que definir este método.

public void write(**byte**[] b)

Escribe el contenido de los b.length bytes del arreglo b al flujo de salida.

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

Escribe en el flujo de salida los bytes desde b[off] hasta b[off+len-1]. Si hay un error en el índice o si b es null, lanza la excepción correspondiente (como son ArithmeticException ambas no hay que vigilarlas). Si hay algún error de I/O se lanza la excepción correspondiente.

public void flush() **throws** IOException

Evacúa el flujo, obligando a que los bytes que estén todavía en el buffer sean escritos al medio físico.

public void close() **throws** IOException

Cierra el flujo y libera los recursos del sistema asociados al flujo. Una vez cerrado el flujo, cualquier otro intento de escribir en él va a causar una excepción. Lanza una excepción (IOException) si se intenta reabrir. En realidad no hace nada, sino que se tiene que reprogramar para que haga lo que tiene que hacer.

Esta es la superclase de la clase que estamos buscando. De manera intermedia hereda a la clase FilterOutputStream, que procesa el flujo antes de colocarlo en el dispositivo de salida. Además de los métodos heredados de OutputStream, agrega los siguientes métodos, además de implementar al método que lee de un entero. Tiene la siguiente definición:

public class FilterOutputStream extends OutputStream

Campo:

protected OutputStream out

El flujo de salida subyacente a ser filtrado.

Constructor:

public FilterOutputStream(OutputStream out)

Crea un filtro para el flujo out.

Método:

public void write(**int** b) **throws** IOException

Implementa el método abstracto write(int) de su superclase.

El resto de los métodos que hereda de OutputStream simplemente los redefine a que invoquen al método correspondiente de su superclase, por lo que no los listamos nuevamente. Sin embargo, como mencionamos antes, tanto out como err se construyen como objetos de la clase PrintStream, que presenta varios métodos, además de los que hereda de FilterOutputStream (hereda, entre otros, el campo que corresponde al flujo de salida FilterOutputStream out). Listaremos sólo algunos de estos métodos. La lista exhaustiva se puede ver en la documentación de Java.

public class PrintStream extends FilterOutputStream implements Appendable

Constructores:

public PrintStream(OutputStream out)

Construye un PrintStream que no auto-evacúa.

public PrintStream(OutputStream out, **boolean** autoFlush)

Construye un nuevo PrintStream. Si autoFlush es verdadero el buffer va a evacuar cuando se escriba un arreglo de bytes, se invoque un método println o se escriba un carácter de línea nueva ('\n').

public PrintStream(String fileName) **throws** FileNotFoundException

Busca escribir en un archivo en disco con nombre fileName. Crea el flujo intermedio OutputStreamWriter necesario.

Métodos:

public void close()

Cierra el flujo, evacuando y cerrando el flujo de salida subyacente.

public boolean checkError()

Evacúa el flujo y verifica su estado de error. Éste es verdadero si el flujo de salida subyacente lanza una IOException distinta de InterruptedIOException, y cuando se invoca al método setError.

protected void setError()

Establece en verdadero el estado de error del flujo.

public void write(**int** b)

Sobreescribe el método de OutputStream escribiendo el byte más bajo al dispositivo.

public void write(**byte**[] buf, **int** off, **int** len)

Hace lo mismo que OutputStream.

public void print(**boolean** b)

Escribe un valor booleano. Escribe en bytes el valor dado por String.valueOf(boolean).

public void print(**char** c)

Escribe un carácter, que se traduce a uno o más bytes, dependiendo de la plataforma.

public void print(**int** i)

Escribe un entero, el valor dado por String.valueOf(int).

public void print(⟨*tipo*⟩ ⟨*identif*⟩)

El ⟨*tipo*⟩ puede ser long, float, double y se escribe lo producido por String.valueOf(⟨*tipo*⟩).

public void print(**char**[] s)

Escribe un arreglo de carácteres, convirtiéndolos a bytes.

public void print(String s)

Escribe una cadena, tomando carácter por carácter y convirtiéndolo a byte.

public void print(Object obj)

Usa el método String.valueOf(Object) para escribir, en bytes, lo solicitado.

public void println()

Imprime únicamente un carácter de fin de línea

public void println(⟨*tipo*⟩ ⟨*identif*⟩)

Admite los mismos tipos de argumentos que print; al terminar de escribir el argumento, escribe un carácter de fin de línea.

public PrintStream printf(String format, Object... args)

Un método para escribir una lista de argumentos con un formato dado por format.

public PrintStream format(String format, Object ... args)

Método equivalente a printf de esta misma clase.

Nota: esta clase presenta muchísimos métodos más que no veremos por el momento. Para conocerlos consultar la documentación de las clases de Java.

Teniendo ya estas clases, y utilizando la salida estándar System.out y System.err, podemos proceder a reprogramar el menú en cuanto a las partes que corresponden a la salida. Como la salida a consola en general no lanza excepciones, lo único que hay que hacer es sustituir las escrituras en Consola por escrituras a estos dos archivos estándar. Sustituiremos cada comunicación normal con el usuario donde aparezca cons.imprimeln por System.out.println; mientras que donde sean mensajes de error para el usuario las sustituiremos por System.err.println. Las líneas que se van a cambiar aparecen a continuación, primero la original y a continuación la que fue cambiada y una breve explicación, en su caso, de por qué.

Antes:

```
import icc1.interfaz.Consola;
```

Después:

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

Dejamos de importar nuestra clase de entrada y salida e importamos el paquete io de Java. Si bien no lo necesitamos todavía, IOException está en este paquete y pudiéramos necesitarla.

Antes:

```
private void reportaNo(Consola cons, String nombre)  {
    cons.imprimeln("El␣estudiante␣:\n\t"
                   + nombre
                   + "\n No esta en el grupo");
}
```

Después:

```
private void reportaNo(PrintStream out,
                       String nombre)     {
    out.println("El␣estudiante␣:\n\t" + nombre
                + "\n no esta en el grupo");
}
```

Consola antes se usaba para entrada **y** salida, pero ahora tenemos que tener un flujo de entrada y dos de salida. Este método únicamente maneja el de salida, por lo cambiamos su parámetros a que sea un flujo de salida del mismo tipo que son out y err.

Antes:

```
String nombre = cons.leeString("Dame␣el␣nombre␣del␣"
                    + "estudiante␣empezando␣por␣"
                    + "apellido␣paterno:");
```

Después:

```
String nombre;
System.out.print("Dame␣el␣nombre␣del␣estudiante,"
                 + "empezando␣por␣apellido␣paterno:");
// Acá va la lectura
```

Ya no tenemos la posibilidad de escribir un mensaje y leer al final de este. Así que partimos en dos el proceso, escribiendo primero el mensaje, sin dar cambio de línea, para que empiece a leer a continuación del mensaje (todavía falta la lectura de la cadena).

Antes:

```
cons.imprimeln(menu);
String sopcion = cons.leeString("Elige␣una␣opción␣-->␣");
```

Después:

```
System.out.println(menu);
System.out.print("Elige␣una␣opción␣-->␣");
// Acá viene lo de la lectura de la opción
```

Estamos dejando pendiente lo referente a la lectura.

Antes:

```
case 0: // Salir
    cons.imprimeln("Espero␣haberte␣servido.\n"
                   + "Hasta␣pronto...");
```

Después:

```
case 0: // Salir
    System.out.println("Espero␣haberte␣servido.\n"
                       + "Hasta␣pronto...");
```

Antes:

```
        cons.imprimeln("El␣estudiante:\n\t"
                    + nombre + "\n Ha sido eliminado ");
    }
    else reportaNo(cons, nombre);
```

Después:

```
        System.out.println("El␣estudiante:\n\t"
                     + nombre + "\n Ha sido eliminado ");
    }
    else reportaNo(System.err, nombre);
```

Antes:

```
    subcad = cons.leeString("Dame␣la␣subcadena␣a␣"+
                                  "buscar:␣");
    do {
        scual = cons.leeString("Ahora␣dime␣de␣cuál␣campo␣:"
                               + "1:Nombre,␣2:Cuenta,"
                               + "␣3:Carrera,␣4:Clave");
        cual = "01234".indexOf(scual);
        if (cual < 1)
            cons.imprimeln("Opción␣no␣válida");
    } while (cual  < 1);
```

Después:

```
    try {
      System.out.print("Dame␣la␣subcadena␣a␣"
                       + "buscar:␣");
      // Leer subcadena
      do {
          System.out.print("Ahora␣dime␣de␣cuál␣campo␣:"
               + "1:Nombre␣2:Cuenta␣3:Carrera␣4:Clave");
          // Leer opción
          cual = "01234".indexOf(scual);
          if (cual < 1)
              System.out.println("Opción␣no␣válida");
      } while (cual  < 1);
```

```
-----------------------------------------------------------------
      donde = (Estudiante)miCurso.buscaSubcad(cual, subcad);
      if (donde != null)
        System.out.println(donde.daRegistro());
      else reportaNo(System.err, subcad);
    } catch (IOException e) {
      System.err.println("Error al dar los datos"
                         + " para buscar");
    } // end of try-catch
```

Dejamos sin llenar las lecturas, aunque las colocamos en un bloque try... catch porque las lecturas pueden lanzar excepciones.

Antes:

```
case 4: // Lista todos
    miCurso.listaTodos(cons);
```

Después:

```
case 4: // Lista todos
    miCurso.listaTodos(System.out);
```

Simplemente pasamos como parámetro el archivo de la consola.

Antes:

```
default: // Error, vuelve a pedir
    cons.imprimeln("No diste una opción válida.\n" +
                   "Por favor vuelve a elegir.");
    return 0;
```

Después:

```
default: // Error, vuelve a pedir
   System.out.println("No diste una opción válida.\n"
                   + "Por favor vuelve a elegir.");
```

En cuanto a la clase ListaCurso, se tienen que hacer las siguientes modificaciones:

Antes:

```
import icc1.interfaz.Consola;
```

Después:

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

Antes:

```
public void listaTodos(Consola cons) {
...
cons.imprimeln(actual.daRegistro());
...
cons.imprimeln("No hay registros en la base de datos");
```

Después:

```
public void listaTodos(PrintStream cons) {
...
cons.println(actual.daRegistro());
...
cons.println("No hay registros en la base de datos");
```

Antes:

```
public void losQueCazanCon(Consola cons, int cual,
                           String subcad)        {
...
cons.imprimeln(actual.daRegistro());
...
cons.imprimeln("No se encontró ningún registro "
               + "que cazara");
```

Después:

```
public void losQueCazanCon(PrintStream cons, int cual,
                           String subcad)        {
...
cons.println(actual.daRegistro());
...
cons.println("No se encontró ningún registro "
               + "que cazara");
```

También en lo que toca a la interfaz implementada por ListaCurso, ParaListas, hay que modificar los encabezados de estos dos métodos:

Antes:

```
public void listaTodos(Consola cons);
public void losQueCazanCon(Consola cons, int cual,
                           String subcad);
```

Después:

```
public void listaTodos(PrintStream cons);
public void losQueCazanCon(PrintStream cons, int cual,
                           String subcad);
```

Como se puede ver, y dado que escribir en consola (mediante PrintStream no lanza excepciones, el uso de esta clase en lugar de Consola para escribir no causa ningún problema.

10.5.2. Lectura desde teclado

Como ya mencionamos antes, si usamos directamente System.in tendremos que hacer un uso muy primitivo de la entrada, como convertir carácteres a cadenas, a enteros, etc. La clase que nos proporciona la conversión desde el teclado es DataInputStream – que es la que usamos en Consola – excepto por la lectura de una cadena que se reporta como "superada"(deprecated. La clase DataInputStream, como se muestra en la figura 10.5 en la página 325, es una subclase de FilterInputStream, que a su vez es una subclase de InputStream. En la documentación de la clase io.DataInputStream recomiendan, para leer cadenas, un ejemplar de la clase BufferedReader, dado que, argumentan, a partir de Java 1.1 ya no pueden garantizar que la conversión de bytes (externos) a carácteres (internos) se haga de manera adecuada. Como en el manejo del menú leemos únicamente cadenas, utilizaremos un ejemplar de esta clase para ello. Como lo hicimos para nuestro archivo de salida, veamos cuál es la línea hereditaria de esta clase, los métodos que hereda y los que redefine.

public class Reader implements Readable, Closeable[3]

Campos:

protected Object lock

Campo que permite bloquear los accesos al archivo desde otra aplicación u otro hilo de ejecución.

Constructores:

protected Reader()

protected Reader(Object lock)

Ambos constructores entregan un objeto de la clase Reader; el segundo asigna un cerrojo externo.

Métodos:

public int read() **throws** IOException

Lee un carácter; se bloquea hasta que se le proporcione uno, haya un error de I/O o se alcance el final del flujo. Regresa el carácter leído (0,. . . ,65535) o -1, si el flujo se acabó.

public int read(**char**[] cbuf)) **throws** IOException

Lee carácteres y los acomoda a partir del lugar 0 en el arreglo de carácteres cbuf; se bloquea hasta que se le proporcione uno, haya un error de I/O o se alcance el final del flujo. Regresa el número de carácteres leídos o -1 si el flujo se acabó.

public int read(**char**[] cbuf, **int** off,**int** len)) **throws** IOException

Lee a lo más len carácteres y los acomoda a partir del lugar off en el arreglo de carácteres cbuf; se bloquea hasta que se le proporcione uno, haya un error de I/O o se alcance el final del flujo. Regresa el número de carácteres leídos o -1 si el flujo se acabó.

public int read(CharBuffer target) **throws** IOException

Regresa -1 si el flujo de los carácteres se acabó o el número de carácteres agregados al buffer.

public long skip(**long** n) **throws** IOException

Salta carácteres (leídos); se bloquea hasta que se le proporcione uno, haya un error de I/O o se alcance el final del flujo. Puede lanzar también la excepción IllegalArgumentException – que no tiene que ser vigilada – si se le da un argumento negativo. Regresa el número de carácteres realmente saltados.

[3]Estamos trabajando con JDK Standard Edition 5.0 o posteriores

public boolean ready() **throws** IOException

Dice si el flujo de entrada tiene carácteres listos (true) o si va a tener que bloquear la entrada y esperar a que se proporcionen más carácteres (false).

public abstract void close() **throws** IOException

Cierra el flujo y libera los recursos asociados a él. Cualquier acción sobre el flujo después de cerrado va a lanzar una excepción. Si se cierra un flujo ya cerrado no pasa nada.

Como ya mencionamos, este flujo es sumamente primitivo. Podríamos leer a un arreglo de carácteres y posteriormente convertirlo a cadena, pero preferimos leer directamente a una cadena. Un amigo nos sugiere la clase DataInputStream, que aunque hereda de la clase InputStream la revisamos. Esta clase hereda directamente de FilterInputStream, quien a su vez hereda de InputStream. Esta última presenta los mismos métodos que Reader, excepto que, como ya vimos, trabaja con bytes en lugar de carácteres (lee a un byte, a un arreglo de bytes, cuenta número de bytes,...). El constructor es un constructor por omisión, y en lugar del método booleano ready de Reader, tiene el siguiente método, que no listamos al presentar a esta clase:

public abstract class InputStream implements Closeable

Método adicional:

public int available () **throws** IOException

Regresa el número de bytes que se pueden leer de este flujo sin bloquear la entrada. A este nivel de la jerarquía siempre regresa 0.

Al igual que en la clase Reader, el método read() es abstracto, por lo que no se pueden construir objetos de esta clase, sino que se tiene que recurrir a alguna de sus extensiones. Revisemos lo que nos interesa de la clase FilterInputStream, de la que hereda directamente DataInputStream:

public class FilterInputStream extends InputStream

Campo:

protected InputStream in

El flujo a ser filtrado (manipulado).

Constructor:

protected FilterInputStream(InputStream in)

Construye el flujo con in como flujo subyacente.

Métodos:

Hereda los métodos ya implementados de InputStream e implementa los declarados como abstractos en la superclase. Como los encabezados y su significado se mantiene, no vemos el caso de repetir esta información.

Tampoco FilterInputStream nos acerca más a leer valores, no carácteres. Veamos qué pasa en DataInputStream, que hereda el campo de FilterInputStream y los métodos de esta clase. Únicamente listaremos algunos de los que nos interesan y que no están en las superclases. No daremos una explicación de cada método ya que los nombres son autodescriptivos.

public class DataInputStream extends FilterInputStream implements DataInput

Constructores:

public DataInputStream(InputStream in)

Este flujo hereda de FilterInputStream y trabaja como intermediario entre el dispositivo y la aplicación, por lo que hay que proporcionarle el dispositivo (in).

Métodos que se agregan:

public final int readUnsignedByte() **throws** IOException

Regresa un entero entre 0 y 255.

public final short readShort() **throws** IOException

Lee un short.

public final int readUnsignedShort() **throws** IOException

public final char readChar() **throws** IOException

public final int readInt() **throws** IOException

public final long readLong() **throws** IOException

public final float readFloat() **throws** IOException

public final double readDouble() **throws** IOException

public final String readLine() **throws** IOException

public final String readUTF() **throws** IOException

Lee una cadena que ha sido codificada con formato UTF-8.

public final String readUTF(DataInput in) **throws** IOException

Lee una cadena que ha sido codificada con formato UTF-8 desde un flujo que implemente a DataInput.

No existe una clase paralela a ésta en la jerarquía de Reader. Sin embargo, al revisar la documentación vemos que nuestro método favorito, readLine(), está anunciado como descontinuado (*deprecated*), lo que quiere decir que no lo mantienen al día y lo usaríamos arriesgando problemas. Pero también en la documentación viene la sugerencia de usar BufferedReader, subclase de Reader, por lo que volteamos hacia ella para resolver nuestros problemas. Presentamos nada más lo agregado en esta clase, ya que los métodos que hereda los [presentamos en la clase Reader antes. Como su nombre lo indica, esta clase trabaja con un buffer de lectura. No listaremos los métodos que sobreescribe (*overrides*) ya que tienen los mismos parámetros y el mismo resultado, así que los pensamos como heredados aunque estén redefinidos en esta clase.

public class BufferedReader extends Reader

Constructores:

public BufferedReader(Reader in)

Construye el flujo con un buffer de tamaño por omisión (generalmente suficientemente grande).

public BufferedReader(Reader in, **int** sz)

Construye el flujo con un buffer de tamaño sz. (generalmente suficientemente grande). Puede lanzar la excepción IllegalArgumentException si el tamaño del buffer es negativo.

Métodos adicionales:

public String readLine() **throws** IOException

Lee una cadena hasta el fin de linea, pero no guarda el fin de línea. La línea se considera terminada con un fin de línea (\n), un retorno de carro o ambos.

Como se puede ver, tampoco esta clase es muy versátil, pero como lo único que

queremos, por el momento, es leer cadenas, nos conformamos. Tenemos ya todo lo que necesitamos para sustituir nuestra clase Consola en el programa. Listaremos nuevamente la forma anterior de las lecturas y la forma que ahora toman. Notar que ahora tenemos que colocar cada lectura en un bloque try... catch porque todos los métodos que llevan a cabo lectura pueden lanzar excepciones de entrada/salida.

Antes:

```
private String pideNombre(Consola cons)   {
    String nombre;
    System.out.print("Dame␣el␣nombre␣del␣estudiante,"
                + "empezando␣por␣apellido␣paterno:");
    nombre=cons.leeString();
    return nombre;
}
```

Después:

```
private String pideNombre(BufferedReader cons)
    throws IOException                          {
    String nombre;
    System.out.print("Dame␣el␣nombre␣del␣estudiante,"
                + "empezando␣por␣apellido␣paterno:");
    try {
        nombre=cons.readLine();
    } catch (IOException e) {
        System.err.println("Error␣al␣leer␣nombre");
        throw e;
    } // end of try-catch
    return nombre;
}
```

El mismo trato se da a los métodos pideCarrera, pideClave y pideCuenta, ya que todos estos métodos leen cadenas. En el menú los cambios que tenemos que hacer se listan a continuación:

Antes:

```
sopcion = cons.leeString("Elige␣una␣opción␣-->␣");
```

```
-----------------------------------------------------------------
```

Después:

```
System.out.print("Elige una opción --> ");
try {
    sopcion = cons.readLine();
} catch (IOException e) {
  System.err.println("Error al leer opción");
  throw new IOException("Favor de repetir elección");
} // end of try-catch
```

Antes:

```
case AGREGA: // Agrega Estudiante
  nombre = pideNombre(cons);
  cuenta = pideCuenta(cons);
  carrera = pideCarrera(cons);
  clave = pideClave(cons);
  miCurso.agregaEstOrden
      (new Estudiante(nombre, cuenta, clave, carrera));
```

Después:

```
case AGREGA: // Agrega Estudiante
  try {
    nombre = pideNombre(cons);
    cuenta = pideCuenta(cons);
    carrera = pideCarrera(cons);
    clave = pideClave(cons);
    miCurso.agregaEstOrden
       (new Estudiante(nombre, cuenta, clave, carrera));
  } catch (IOException e) {
    System.out.println("Error al leer datos del "
                + "estudiante.\nNo se pudo agregar");
  } // end of try-catch
```

Antes:

```
case QUITA: // Quita estudiante
    nombre = pideNombre(cons);
    donde = (Estudiante)miCurso.buscaSubcad
        (Estudiante.NOMBRE, nombre);
```

```
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
        if ( donde != null) {
            nombre = donde.daNombre();
            miCurso.quitaEst(nombre);
            System.out.println("El estudiante:\n\t"
                    + nombre + "\n Ha sido eliminado ");
        }
        else reportaNo(nombre);
```

Después:

```
    case QUITA: // Quita estudiante
        try {
            nombre = pideNombre(cons);
            donde = (Estudiante)miCurso.buscaSubcad
                (Estudiante.NOMBRE, nombre);
            if ( donde != null) {
                nombre = donde.daNombre();
                miCurso.quitaEst(nombre);
                System.out.println("El estudiante:\n\t"
                      + nombre + "\nHa sido eliminado ");
            }
            else reportaNo(nombre);
        } catch (IOException e) {
            System.err.println("No se proporcionaron bien"
                  + " los datos.\nNo se pudo eliminar.");
        } // end of try-catch
```

Antes:

```
    case BUSCA: // Busca subcadena
        subcad = cons.leeString("Dame la subcadena a "
                        + "buscar: ");
        do {
            scual = cons.leeString("Ahora dime de cuál "
                + "campo : 1:Nombre 2:Cuenta, 3:Carrera"
                + ", 4:Clave -->");
            cual = "01234".indexOf(scual);
            if (cual < 1)
                System.out.println("Opción no válida");
        } while (cual  < 1);
-----------------------------------------------------------------
```

Antes: (continúa...)

```
        donde = (Estudiante)miCurso.
                          buscaSubcad(cual, subcad);
        if (donde != null)
            System.out.println(donde.daRegistro());
        else reportaNo(subcad);
```

Después:

```
    case BUSCA: // Busca subcadena
        try {
            System.out.print("Dame la subcadena a "+
                             "buscar: ");
            subcad = cons.readLine();
            do {
                System.out.print("Ahora dime de cuál "
                    + "campo : 1:Nombre 2:Cuenta"
                    + " 3:Carrera 4:Clave -->");
                scual = cons.readLine();
                cual = "01234".indexOf(scual);
                if (cual < 1)
                  System.out.println("Opción no válida");
            } while (cual  < 1);
            donde = (Estudiante)miCurso.
                           buscaSubcad(cual, subcad);
            if (donde != null)
                System.out.println(donde.daRegistro());
            else reportaNo(subcad);
        } catch (IOException e) {
            System.err.println("Error al dar los "
                               + "datos para buscar");
        } // end of try-catch
```

Antes:

```
    case LISTAALGUNOS: // Lista con criterio
      subcad = cons.leeString("Da la subcadena que "
                        + " quieres contengan los "
                        + "registros:");
      do {
         scual = cons.leeString("Ahora dime de cuál campo:"
             + " 1:Nombre 2:Cuenta 3:Carrera 4:Clave -->");
```

Antes: (continúa...)

```
        cual = "01234".indexOf(scual);
        if (cual < 1)
            System.out.println("Opción no válida");
    } while (cual  < 1);
    miCurso.losQueCazanCon(System.out, cual, subcad);
```

Después:

```
  case LISTAALGUNOS: // Lista con criterio
    try {
      System.out.println("Da la subcadena que " +
                         " quieres contengan los " +
                         "registros:");
      subcad = cons.readLine();
      do {
         System.out.print("Ahora dime de cuál campo:"
            + "1:Nombre 2:Cuenta 3:Carrera 4:Clave -->");
         scual = cons.readLine();
         cual = "01234".indexOf(scual);
         if (cual < 1)
              System.out.println("Opción no válida");
      } while (cual  < 1);
      miCurso.losQueCazanCon(System.out, cual, subcad);

    } catch (IOException e) {
      System.err.println("Error al dar los datos"
                         + "para listar");
    } // end of try-catch
```

Antes:

```
  public static void main(String[] args)  {
      ...
      Consola consola = new Consola();
```

Después:

```
  public static void main(String[] args)  {
      ...
      BufferedReader consola = new BufferedReader
          (new InputStreamReader(System.in));
```

Antes:

```
while ((opcion = miMenu.daMenu(consola, miCurso))
                                                != -1);
```

Después:

```
while (opcion != -1) {
    try {
        opcion = miMenu.daMenu(consola, miCurso);
    } catch (IOException e) {
        System.out.println("Opción mal elegida");
        opcion=0;
    } // end of try-catch
} // while
```

10.6 Redireccionamiento de in, out y err

Muchas veces queremos que los resultados de un programa, o los mensajes de error, en lugar de ir a los dispositivos estándar (todos a la consola) se graben en algún archivo en disco para poder examinarlos con calma. Además de la manera en que Unix permite redireccionar la salida, podemos, desde el programa, conseguir esto. Para ello contamos con métodos en java.lang.System que permiten hacerlo. Ellos son:

```
public final static void setIn(InputStream newIn)
public final static void setOut(OutputStream newOut)
public final static void setErr(PrintStream newErr)
```

A continuación dos ejemplos:

```
System.setIn(new FileInputStream("misdatos.txt"));
System.setOut(new PrintStream(new FileOutputStream("misdatos.out")));
```

10.7 Persistencia de la base de datos

Hasta ahora únicamente hemos trabajado con la consola o bien con redireccionamiento de la consola, pero no hemos entrado a la motivación principal de este capítulo y que consiste en lograr guardar el estado de nuestra base de datos para que pueda ser utilizado posteriormente como punto de partida en la siguiente ejecución.

Podemos almacenar, en primera instancia, la base de datos como un conjunto de cadenas, y para ello podemos volver a utilizar a los flujos BufferedReader y BufferedWriter que ya conocemos, pero en esta ocasión queremos que el flujo subyacente sea un archivo en disco y no un flujo estándar. Revisemos entonces la clase FileReader y FileWriter que me van a dar esa facilidad. Estos flujos extienden, respectivamente, a InputStreamReader y OutputStreamWriter, que a su vez heredan, respectivamente, de Reader y Writer. De esta jerarquía únicamente hemos revisado la clase Reader, así que procedemos a ver las otras clases de la jerarquía que vamos a requerir.

public abstract class Writer implements Appendable, Closeable, Flushable

Campo:

protected Object lock
Sincroniza el acceso a este flujo.

Constructores:

protected Writer()
Construye un flujo de salida de carácteres a ser sincronizado por él mismo.

protected Writer(Object lock)
Construye un flujo de carácteres que será sincronizado usando lock.

Métodos:

public Writer append(**char** c) **throws** IOException
Agrega el argumento al flujo this.

public Writer append(CharSequence csq) **throws** IOException
CharSequence es una interfaz de Java que básicamente proporciona métodos para convertir una sucesión de carácteres, codificados en cualquier código de 16 bits en cadenas o subsucesiones. Agrega el argumento al flujo this.

public Writer append(CharSequence csq, **int** start , **int** end) **throws** IOException

Agrega al flujo this la subsucesión de csq que empieza en start y termina en el carácter inmediatamente a la izquierda de end.

public abstract void close () **throws** IOException

Cierra el flujo vaciándolo primero. Una vez cerrado, cualquier intento de vaciarlo o escribir en él provocará una excepción de entrada/salida. No importa que un flujo se intente cerrar una vez cerrado.

public abstract void flush () **throws** IOException

Provoca que todas las escrituras pendientes sean vaciadas al flujo proporcionado por el sistema operativo. Sin embargo, el sistema operativo podría no vaciar *su* buffer.

public void write(**int** c) **throws** IOException

Escribe un único carácter en el flujo, tomando los 16 bits bajos del entero proporcionado. Este método debería ser sobreescrito por las subclases correspondientes.

public void write(**char**[] cbuf) **throws** IOException

Escribe en el flujo los carácteres presentes en el arreglo cbuf.

public abstract void write(**char**[], **int** off , **int** len) **throws** IOException

Escribe en el flujo el contenido del arreglo cbuf a partir del carácter en la posición off y un total de len carácteres.

public void write(String str) **throws** IOException

Escribe el contenido de la cadena en el flujo.

public void write(String str , **int** off , **int** len) **throws** IOException

Escribe la subcadena de str desde la posición off un total de len carácteres.

Realmente el único método con el que hay que tener cuidado es el que escribe un carácter (entero), porque el resto de los métodos se construyen simplemente invocando a éste.

Las clases que heredan directamente de Reader y Writer son, respectivamente,

InputStreamReader y OutputStreamWriter que pasamos a revisar. Primero revisaremos la clase InputStreamReader.

public class InputStreamReader extends Reader

Hereda el campo lock de Reader y los métodos definidos en la superclase. Listamos los que son redefinidos en esta subclase.

Constructores:

public InputStreamReader(InputStream in)

Construye un flujo de entrada *sobre* el flujo de bytes que se le proporcione (y que debe existir como objeto). Básicamente va a traducir bytes en carácteres.

public InputStreamReader(InputStream in, String charsetName) **throws** UnsupportedEncodingException

Construye un flujo de entrada *sobre* un flujo de bytes. Va a traducir de acuerdo al código nombrado en charsetName. Si el código no existe lanza una excepción de código inválido.

public InputStreamReader(InputStream in, Charset cs)

Construye un flujo sobre in y que va a traducir los bytes de acuerdo al código dado en cs.

Métodos:

public void close() **throws** IOException

Cierra el flujo correspondiente. Lanza una excepción si hay algún error de entrada/salida.

public String getEncoding()

Regresa el nombre del código de traducción de bytes que es el usado por este flujo.

public int read() **throws** IOException

Redefine el método correspondiente en Reader (recuérdese que en Reader era un método abstracto).

public int read(**char**[] cbuf, **int** offset, **int** length) **throws** IOException

Redefine el método correspondiente en Reader.

public boolean ready() **throws** IOException

Redefine el método ready en Reader.

Siguiendo en orden en la jerarquía, ahora revisaremos OutputStreamWriter.

public class OutputStreamWriter extends Writer

Hereda el campo lock de la superclase. Listaremos los constructores y los métodos abstractos de la superclase que se implementan en esta clase.

Constructores:

public OutputStreamWriter(OutputStream out)

Construye un flujo de salida que va a convertir carácteres en bytes. Se monta *sobre* un flujo de salida de bytes.

public OutputStreamWriter(OutputStream out, Charset cs)

Construye un flujo de salida de carácteres a bytes, montado sobre un flujo de salida de bytes, out que usa la codificación dada por cs.

public OutputStreamWriter(OutputStream out, CharsetEncoder enc)

Construye un OutputStreamWriter sobre un flujo de bytes out y que usa la codificación dada por enc.

Métodos:

public void close() **throws** IOException

Cierra el flujo vaciando el buffer de salida. Lanza una excepción si tiene problemas de entrada/salida.

public void flush() **throws** IOException

Implementa el método correspondiente en Writer.

public String getEncoding()

Regresa el nombre del código que se está usando para escribir los bytes correspondientes a los carácteres en memoria.

public void write(**int** c) **throws** IOException

Implementa el método correspondiente en Writer.

public void write(**char**[] cbuf, **int** off **int** len) **throws** IOException

Implementa al método correspondiente en Writer.

public void write(String str, **int** off **int** len) **throws** IOException

Implementa al método correspondiente en Writer.

Ahora sí ya podemos pasar a revisar las clases FileReader y FileWriter que heredan respectivamente de InputStreamReader y OutputStreamWriter. Empezaremos por el flujo de entrada. En esta subclase únicamente se definen los constructores, ya que se heredan precisa y exactamente los métodos implementados en InputStreamReader.

public class FileReader extends InputStreamReader

Dado que ésta es una subclase de InputStreamReader, va a leer bytes y convertirlos en carácteres. Al igual que su superclase, también trabaja *sobre* un InputStream.

Constructores:

public FileReader(String fileName) **throws** FileNotFoundException

Crea un flujo para lectura de disco cuyo nombre es fileName. Si no encuentra el flujo lanza una excepción de archivo no encontrado.

public FileReader(File file) **throws** FileNotFoundException

File es una representación abstracta de un archivo del sistema operativo en cuestión, e incluye aspectos como definir el separador y terminador de archivos, la ruta en el disco, la identificación del archivo en disco, etc. (ver la definición de esta clase en la documentación de Java). Crea un flujo para lectura identificado con file. Si no encuentra el flujo lanza una excepción de archivo no encontrado.

public FileReader(FileDescriptor fd)
throws FileNotFoundException

Un FileDescriptor es un objeto que describe a un archivo en disco (ver documentación de Java). Crea un flujo de lectura desde disco, donde el archivo está asociado al FileDescriptor. Si no lo encuentra, lanza una excepción de archivo no encontrado.

Métodos:

Los heredados de Reader y de InputStreamReader, que ya revisamos.

Para los flujos de salida que escriben a disco tenemos una situación similar a la de archivos de entrada, pues lo único que se define para la subclase FileWriter son los constructores. Para el resto de los métodos y campos se heredan las implementaciones dadas por OutputStreamWriter. Veamos la definición.

public class FileWriter extends OutputStreamWriter

Enlaza a un flujo de salida de bytes con un archivo en disco. Lo único que se implementa a nivel de esta subclase son los constructores.

Constructores:

FileWriter (File file) **throws** IOException

Construye un flujo a disco sobre el flujo file, que podría ser, a su vez, un FileWriter o cualquier OutputStream o subclases de ésta. La excepción la lanza si file es un directorio y no un archivo, no existe pero no puede ser creado o no puede ser abierto por cualquier otra razón.

FileWriter (File file , **boolean** append) **throws** IOException

Construye un flujo a disco sobre el flujo file, que podría ser, a su vez, un FileWriter o cualquier OutputStream o subclases de ésta. La excepción la lanza si file es un directorio y no un archivo, no existe pero no puede ser creado o no puede ser abierto por cualquier otra razón. Si append es verdadero, la escritura se realiza al final del archivo; si es falsa se realiza al principio del archivo, como en el constructor sin parámetro booleano.

FileWriter (FileDescriptor fd)

Construye un flujo a disco asociado con el FileDescriptor.

FileWriter (String fileName) **throws** IOException

Construye el flujo dado un nombre. Lanza una excepción de entrada/salida por las mismas causas que los otros constructores.

FileWriter (String fileName, **boolean** append)
throws IOException

Construye el flujo dado un nombre. Si append es verdadera entonces escribe al final del archivo; si el archivo no existe lo crea. Lanza una excepción de entrada/salida por las mismas causas que los otros constructores.

Hay que recordar que la herencia permite que donde quiera que aparezca una clase como parámetro, los argumentos pueden ser objetos de cualquiera de sus subclases. Con esto en mente pasamos a implementar las opciones en el menú de leer de un archivo en disco o escribir a un archivo en disco para guardar la información generada en una sesión dada.

10.7.1. Cómo guardar datos en un archivo en disco

La información en disco se guarda invariablemente en bytes. De hecho, un archivo en disco es, simplemente, una sucesión muy grande de bytes, que puede ser interpretado de muy diversas maneras: podemos tomar los bytes de cuatro en cuatro, e interpretar cada cuatro bytes como un entero; o podemos tomarlos de seis en seis e interpretar a cada grupo como un doble. En ningún otro caso es más cierto el dicho de que *todo depende del color del cristal con que se mira* que en la lectura (interpretación) de archivos en disco. Y esta interpretación depende del enunciado con que se tenga acceso al archivo y el tipo de flujo que se utilice para ello.

Como mencionamos al principio, por lo pronto escribiremos y leeremos (recuperaremos) cadenas, representadas por sucesiones de bytes y separadas entre sí por carácteres de fin de línea. Dada esta situación utilizaremos para entrada objetos de la clase FileReader y para salida objetos de la clase FileWriter.

Agregaremos a nuestro menú tres opciones nuevas, leer registros desde disco, escribir registros a disco y agregar registros en disco a un archivo que ya tiene información. Qué archivo usar deberá ser una decisión que se toma una vez elegida alguna de estas opciones. Consideramos que cualquiera de estas acciones puede llevarse a cabo en cualquier momento del proceso y por lo tanto, la elección del archivo debe hacerse en el momento en que se elige la opción. La otra opción pudiese ser elegir el archivo al entrar al proceso. Pero eso obligaría al usuario, ya sea que quiera o no leer de/guardar en archivos de disco, proporcionar el nombre para los mismos, algo que no consideramos adecuado. Este curso de acción también evitaría que se pudiera leer o escribir, en una misma sesión, más de un archivo. Por lo tanto, la definición del archivo a usar estará situada dentro de la opción correspondiente.

Podemos implementar ya esta opción. Lo primero que hacemos es declarar tres constantes simbólicas para usarlas en el switch, y que son:

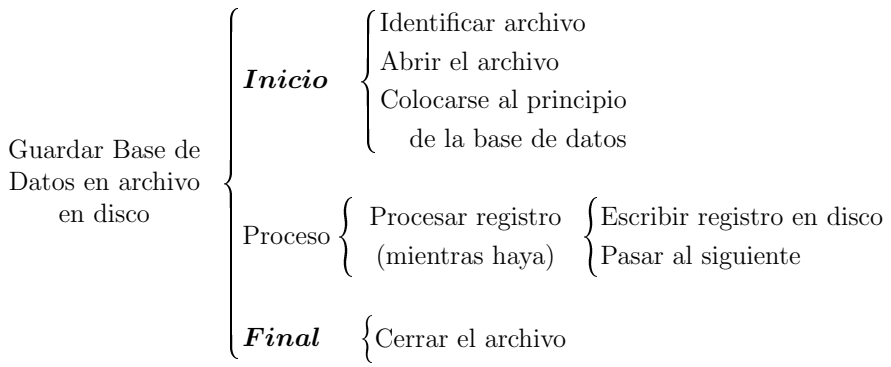
```
static final int AGREGA = 1,
    ...
    LEER = 6,
    GUARDAR = 7,
    PEGARDISCO = 8;
```

La opción de guardar lo que llevamos en un archivo en disco debe seguir el guión dado en la figura 10.10.

Para identificar el archivo que deseamos usar, algo que tendremos que hacer en los tres casos, construimos un método que lea el nombre del archivo de disco

en consola, y que queda como se ve en el listado 10.1.

Figura 10.10 Algoritmo para guardar la base de datos en disco



Código 10.1 Método que solicita al usuario nombre de archivo (**MenuListaIO**)

```
50:     /**
51:      * Pide al usuario el nombre del archivo en el que desea
52:      * escribir o del que desea leer.
53:      * @param cons Dispositivo del que va a leer el nombre
54:      * @param lectura True: lectura, false: escritura.
55:      * @return el archivo solicitado.
56:      */
57:     public String pideNombreArch(BufferedReader cons, int caso)
58:         throws IOException {
59:         String mensaje = "Por favor dame el nombre del archivo\n"
60:             + (caso == LEER
61:                 ?"del que vas a leer registros"
62:                 :(caso == GUARDAR
63:                     ?"en el que vas a guardar la base de datos"
64:                     :"en el que vas a agregar registros"))
65:             + ":\t";
66:         String nombre;
67:         try {
68:             System.out.print(mensaje);
69:             nombre = cons.readLine();
70:         } catch (IOException e) {
71:             throw e;
72:         } // end of try-catch
73:         return nombre;
74:     }
```

Al método pideNombreArch le pasamos el flujo que estamos usando para comu-

nicarnos con el usuario. Queremos que el mensaje sea preciso respecto a qué vamos a hacer con el archivo, pero como usamos el mismo método simplemente le pasamos de cual caso se trata caso para que pueda armar el mensaje correspondiente (líneas 59: a 64:). Después entramos a un bloque try...catch en el que vamos a leer del usuario el nombre del archivo. El método, como lo indica su encabezado, exporta la excepción que pudiera lanzarse al leer el nombre del archivo. Estamos listos ya para programar el algoritmo de la figura 10.10 en la página opuesta. El código lo podemos ver en el listado 10.2.

Código 10.2 Código para guardar la base de datos (**MenuListaIO**)

```
200:  case GUARDAR:
201:      try {
202:          sArchivo = pideNombreArch(cons,GUARDAR);
203:          archivoOut = new BufferedWriter(new FileWriter(sArchivo));
204:          System.out.println("Abrí archivo");
205:          Estudiante lista = ((Estudiante)miCurso.daLista());
206:          while (lista != null) {
207:              archivoOut.write(lista.daNombre());
208:              archivoOut.newLine();
209:              archivoOut.write(lista.daCuenta());
210:              archivoOut.newLine();
211:              archivoOut.write(lista.daCarrera());
212:              archivoOut.newLine();
213:              archivoOut.write(lista.daClave());
214:              archivoOut.newLine();
215:              System.out.println(lista.daNombre()+"\n");
216:              System.out.println(lista.daCuenta()+"\n");
217:              System.out.println(lista.daCarrera()+"\n");
218:              System.out.println(lista.daClave()+"\n");
219:              lista = lista.daSiguiente();
220:          } // end of while (lista != null)
221:          archivoOut.flush();
222:          archivoOut.close();
223:      } catch (IOException e) {
224:          System.err.println("No pude abrir archivo");
225:      } // end of try-catch
226:      finally {
227:        try {
228:          if (archivoOut != null)  {
229:              archivoOut.close();
230:          }
231:        } catch (IOException e) {
232:           System.err.println(``No pude cerrar archivo'');
233:      } // end of finally
234:      return GUARDAR;
```

Colocamos toda la opción en un bloque try. . . catch porque queremos suspender en cuanto se presente una primera excepción, ya sea que no podemos abrir el archivo o que haya algún registro que no podemos escribir. El bloque tiene cláusula finally – líneas 226: a 233: – para que en caso de que haya algún problema se proceda a cerrar el archivo. Se verifica antes de intentar cerrarlo que el archivo exista y se haya logrado abrir.

En las líneas 202: y 203: solicitamos el nombre del archivo a usar y procedemos a abrir el archivo. En este punto la única excepción que pudo haber sido lanzada es en la interacción con el usuario, ya que la apertura de un archivo en disco difícilmente va a lanzar una excepción.

En la línea 205: nos colocamos al principio de la lista. Como el método miCurso.daLista() regresa un objeto de tipo Object tenemos que hacer un *casting*. Una vez que estamos al principio de la lista, procedemos a escribir registro por registro – líneas 207: a 210:–. Escribimos campo por campo, en el orden en que están en el registro, separando los campos entre sí por un carácter de fin de línea – archivoOut.newLine() – propio del sistema operativo en el que esté trabajando la aplicación. En las líneas 212: a 215: simplemente se hace eco de lo que se escribió en el disco como medida de verificación.

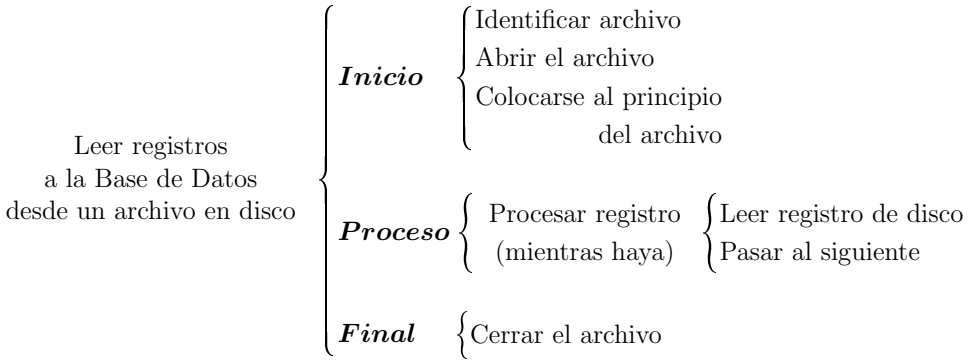
En seguida se pasa al siguiente registro para procesarlo de la misma manera. Al terminar simplemente se cierra el archivo, haciendo persistente el contenido de la lista en memoria.

10.7.2. Cómo leer registros de un archivo de disco

El algoritmo para leer registros de una archivo en disco es la imagen del proceso para guardar. Éste se puede ver en la figura 10.11.

Para identificar el archivo del que vamos a leer usamos el mismo método, excepto que con un mensaje apropiado. Al abrir el archivo automáticamente nos encontraremos frente al primer registro. A partir de ahí, suponemos que el archivo está correcto y que hay cuatro cadenas sucesivas para cada registro que vamos a leer. El código que corresponde a esta opción se encuentra en el listado 10.3 en la página opuesta.

Figura 10.11 Algoritmo para leer registros desde disco



Código 10.3 Opción para leer registros desde disco (**MenuListaIO**) 1/2

```
234:     case LEER: // Leer de disco
235:       try {
236:           sArchivo = pideNombreArch(cons, LEER);
237:           archivoIn = new BufferedReader(new FileReader(sArchivo));
238:
239:           while ((nombre = archivoIn.readLine()) != null) {
240:               cuenta = archivoIn.readLine();
241:               carrera = archivoIn.readLine();
242:               clave = archivoIn.readLine();
243:
244:               miCurso.agregaEstFinal
245:                   (new Estudiante(nombre,cuenta,carrera, clave));
246:           } // end of while ((nombre = archivoIn.readLine()) != null)
247:       } catch (FileNotFoundException e)  {
248:           System.out.println("El archivo" + sArchivo
249:              + "no existe.");
250:           throw e;
251:       } catch (IOException e) {
252:           System.err.println("No pude abrir archivo");
253:       } catch (Exception e) {
254:           System.out.println("NO alcanzaron los datos");
255:           if (carrera == null) {
256:               carrera = "????";
257:               System.out.println("No hubo carrera");
258:           } // end of if (carrera == null)
```

Código 10.3 Opción para leer registros desde disco (**MenuListaIO**) 2/2

```
234:            if (cuenta == null) {
235:                cuenta = "000000000";
236:                System.out.println("No hubo cuenta");
237:            } // end of if (cuenta == null)
238:            if (clave == null) {
239:                clave = "????";
240:                System.out.println("No hubo clave");
241:            } // end of if (clave == null)
242:        } // end of catch
243:        finally {
244:            if (archivoIn != null) {
245:                try {
246:                    archivoIn.close();
247:                } catch (IOException e) {
248:                    System.err.println("No pude cerrar el"
249:                                       + " archivo de lectura");
250:                } // end of try-catch
251:            } // end of if (archivoIn != null)
252:        } // end of finally
253:        return LEER;
```

Nuevamente tenemos que encerrar nuestro proceso en un bloque try...catch, ya que así nos lo exigen los métodos de entrada/salida que estamos utilizando. El proceso principal, si todo marcha bien, consiste en leer el nombre del archivo en la línea 239: y luego proceder a abrirlo. Una vez abierto el archivo se van a leer cuatro cadenas para considerarlas como los datos para un registro de estudiante, el cual se agrega a la base de datos en las líneas244: y 245:. Conforme se van leyendo las cadenas va avanzando el archivo, por lo que no hay necesidad de avanzarlo. Sin embargo, al leer el nombre sí verificamos si se alcanzó el fin de archivo; estamos suponiendo que los registros vienen completos en grupos de cuatro cadenas y en el orden en que se intentan leer. En la línea 239: se verifica que no se haya alcanzado el fin de archivo; si se alcanzó el fin de archivo, el método regresará una referencia nula.

Podemos encontrarnos con varios errores en este proceso. El primero de ellos es que pretendamos leer de un archivo que no existe. En este caso se lanza una excepción de la clase FileNotFoundException que manejamos parcialmente: escribimos un mensaje y exportamos la excepción, ya que no se va a poder hacer nada.

El siguiente error que podríamos tener en nuestro proceso es que los grupos de cuatro cadenas no estén completos y no haya en el archivo un múltiplo de cuatro en el número de las cadenas. En este caso, al intentar leer cadenas nos encontraremos más allá del fin de archivo – líneas 242: a 245: – lo que lanzará una

excepción de la clase IOException, que es atrapada en la línea 251: y manejada simplemente no agregando ese registro incompleto y absorbiendo la excepción.

La ausencia de suficientes datos también puede lanzar una excepción de tipo aritmético, lo que preveremos en la línea 253: donde tratamos de averiguar cuáles fueron los datos que no pudimos leer, mandando el mensaje adecuado y absorbiendo la excepción.

Sin importar si hubo una excepción o no trataremos de cerrar el archivo si es que éste se abrió, mediante una cláusula finally – en las líneas 243: a 252:– en la que, si al cerrar el archivo se lanzó una excepción, ésta se absorbe después del mensaje correspondiente.

10.7.3. Cómo agregar a un archivo ya creado

El algoritmo para agregar registros a un archivo creado previamente es exactamente igual que el que crea un archivo nuevo, excepto que al abrir el archivo hay que indicar que se busca uno que ya existe; adicionalmente, en lugar de "escribir" en el archivo procedemos a agregar (*append*). Usando entonces el mismo algoritmo que para guardar, con los cambios que acabamos de mencionar, el código para esta opción queda como se muestra en el listado 10.4.

Código 10.4 Opción de agregar registros a un archivo en disco (**MenuListaIO**) 1/2

```
300:   case PEGARDISCO:
301:       try {
302:          sArchivo = pideNombreArch(cons,PEGARDISCO);
303:          archivoOut = new BufferedWriter
304:                       (new FileWriter(sArchivo,true));
305:          Estudiante lista = ((Estudiante)miCurso.daLista());
306:          while (lista != null) {
307:              archivoOut.append(lista.daNombre());
308:              archivoOut.newLine();
309:              archivoOut.append(lista.daCuenta());
310:              archivoOut.newLine();
311:              archivoOut.append(lista.daCarrera());
312:              archivoOut.newLine();
313:              archivoOut.append(lista.daClave());
314:              archivoOut.newLine();
```

Código 10.4 Opción de agregar registros a un archivo en disco (**MenuListaIO**) 2/2

```
300:                System.out.println(lista.daNombre()+"\n");
301:                System.out.println(lista.daCuenta()+"\n");
302:                System.out.println(lista.daCarrera()+"\n");
303:                System.out.println(lista.daClave()+"\n");
304:                lista = lista.daSiguiente();
305:            } // end of while (lista != null)
306:            archivoOut.flush();
307:            archivoOut.close();
308:        } catch (FileNotFoundException e) {
309:            System.err.println("El archivo " + sArchivo
310:                               + "no existe!!");
311:            throw e;
312:        } catch (IOException e) {
313:            System.err.println("No pude abrir archivo");
314:        } // end of try-catch
315:        finally {
316:            try {
317:                if (archivoOut != null) {
318:                    archivoOut.close();
319:                } // end of if (archivoOut != null)
320:            } catch ( IOException e) {
321:                System.err.println("No pude cerrar el archivo");
322:            } // end of try-catch
323:        } // end of finally
324:        return PEGARDISCO;
```

Los únicos cambios en esta opción son:

- En las líneas 303: y 304:, donde se usa otro constructor para el flujo, el que permite indicar si se usa un archivo ya existente para agregar a él.
- En las líneas 307: a 313: se usa el método append en lugar del método write, ya que deseamos seguir agregando al final del archivo.
- En la línea 308: se trata de atrapar una excepción de archivo no encontrado, ya que en el caso de querer agregar a un archivo éste debe existir, mientras que en el contexto de crear un archivo nuevo, esta excepción no puede presentarse.

Como se pudo observar, el manejo de flujos en Java tiene un trato uniforme y, una vez que se manejan las excepciones, no representa mayor problema. Un punto que hay que vigilar, sin embargo, es la construcción de flujos que se montan sobre otros flujos. Acá hay que tener cuidado en usar el constructor adecuado

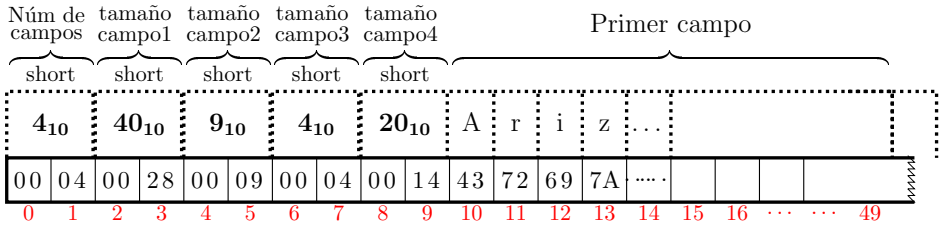
y proporcionar un flujo que pertenezca a la clase que indica el constructor. La elección del flujo adecuado depende de qué es lo que queremos hacer con él y cuáles son las operaciones más frecuentes que esperamos. Los flujos que hemos usado hasta ahora manejan de manera idónea lectura y escritura de cadenas, pero puede suceder que ésta no sea la operación más frecuente o la que busquemos facilitar.

Un comentario final: cuando en nuestra aplicación requerimos de flujos para montar en ellos al flujo final con el que queríamos, los construimos "al vuelo", es decir, de manera anónima sin asignarles un identificador. Todas estas construcciones las podíamos haber hecho en dos pasos, primero construir el archivo más primitivo (el que tiene contacto con el sistema) asignándole un identificador y posteriormente construir el flujo de nuestra aplicación. Como no se utiliza el archivo base de ninguna otra manera más que para establecer la conexión no vimos necesario hacer esto último.

10.8 Escritura y lectura de campos que no son cadenas

Supongamos que queremos tener en el disco una imagen similar a nuestra primera implementación de la base de datos, una sucesión de carácteres, donde sabemos donde empieza un registro y termina el otro porque contamos carácteres (o bytes). Supongamos también que nuestro archivo es, de alguna manera, autodescriptivo, esto es que tendrá como encabezado del archivo (primera información) el número de campos y el tamaño de cada uno de ellos. De lo anterior, nuestro archivo se vería como se muestra en la figura 10.12. Mostramos el contenido de cada uno de los bytes[4] en hexadecimal (4 bits para cada símbolo). Por ejemplo, el encabezado de este archivo nos indica que cada registro consiste de cuatro (4) campos; el primer campo (nombre) ocupa 40 bytes; el segundo campo (cuenta) ocupa nueve bytes; el tercer campo (carrera) ocupa cuatro bytes; el cuarto campo (clave) ocupa 20 bytes. Cabe aclarar que la denominación entre paréntesis está fuera de la aplicación; simplemente lo anotamos para tener una idea más clara de qué es lo que estamos haciendo. Como los enteros que estamos manejando son relativamente pequeños, los guardaremos en variables tipo `short`, que ocupa cada una dos bytes. Las líneas punteadas indican la "máscara" que estamos aplicando al archivo (cómo interpretamos la información) y el valor dentro de estas celdas indica el número en base 10 que está grabado. Abajo de cada celda se encuentra la posición del byte correspondiente en el archivo, empezando de 0 (cero).

Figura 10.12 Formato de un archivo binario autodescrito



Lo que conviene es que la clase para cada registro nos entregue el tamaño de cada campo. Podemos suponer que esto es así, agregando a la clase InfoEstudiante un arreglo con esta información:

```
short[] tamanhos = {4, 40, 9, 4, 20};
```

quedando en tamanhos[0] el número de campos, en tamanhos[1] el tamaño del primer campo y así sucesivamente. Agregamos a la clase un método

```
public short getTamanho(int campo)  {
  return tamanhos[campo];
}
```

que simplemente regresa el tamaño del campo solicitado.

Podemos pensar en un archivo que es heterogéneo, en el sentido de que lo que llamamos el encabezado del mismo no tiene la forma que el resto de los elementos; éstos se componen de $\boldsymbol{n}$ campos – la $\boldsymbol{n}$ viene en los primeros dos bytes del archivo con formato binario de un entero corto (short) – con un total de $\boldsymbol{k}$ bytes que corresponde a la suma de los $\boldsymbol{n}$ enteros cortos que aparecen a partir del byte 2 del archivo. El encabezado del archivo consiste de $\mathbf{2}(\boldsymbol{n}+\mathbf{1})$ bytes. Una vez procesados estos $\boldsymbol{n}+\mathbf{1}$ enteros cortos, el resto del archivo lo podemos ver como un arreglo unidimensional de bytes (similarmente a como manejamos la base de datos en cadenas al principio).

Deseamos insistir en lo que dijimos al principio de esta sección: todos los archivos en disco se componen de bytes; la manera de agrupar los bytes para obtener información que tenga sentido depende del software que se use para verlo, de las máscaras que le apliquemos al archivo. Una vez que terminemos de armar nuestro archivo con el formato que acabamos de ver, podrán observar el archivo con alguno de los visores de su sistema operativo y verán que también los primeros $\mathbf{2}(\boldsymbol{n}+\mathbf{1})$ bytes podrían tratar de interpretarlos como carácteres ASCII, no como variables de Java; por supuesto que si hacen esto la mayoría de estos carácteres no se podrán ver en pantalla (por ejemplo, el 0 binario) o aparecerán carácteres que no guardan ninguna relación con lo que ustedes esperarían ver.

Como queremos escribir y leer "binario" (imágenes de variables de Java) para el encabezado del archivo, usaremos el flujo que nos permite hacer esto directamente y que es DataOutputStream y DataInputStream respectivamente. Esta última ya la revisamos en la página 345, por lo que pasamos a revisar la clase DataOutputStream, aunque van a ver que corre paralela al flujo correspondiente de entrada.

public class DataOutputStream extends FilterOutputStream implements DataOutput

Campos:

protected int written

Indica el número de bytes que se han escrito sobre este flujo hasta el momento.

Constructores:

public DataOutputStream(OutputStream out)

Construye sobre un flujo de salida que esté conectado a algún dispositivo. Éste es el único constructor de este flujo.

Métodos:

public void flush () **throws** IOException

Vacía el buffer de los bytes almacenados. Usa para ello el método dado por el flujo de salida dado como argumento en el constructor.

public final int size ()

Regresa el valor del campo written.

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

Escribe la porción del arreglo de bytes b que empieza en off y tiene un tamaño máximo de len bytes. Si no se lanza una excepción, written se incrementa en len unidades.

void write(**int** b) **throws** IOException

Implementa el método write de la clase OutputStream.

public final void write***XXX***(***YYY*** par) **throws** IOException

Esta denominación incluye en realidad a varios métodos, que toman la representación interna del tipo ***YYY*** y lo transfieren tal cual al flujo de salida. A continuación damos las combinaciones de ***XXX*** y ***YYY*** que tenemos en los distintos métodos:

XXX	*YYY*	Descripción
Boolean	boolean	Escribe una booleana como un valor de 1 byte.
Byte	int	Escribe un byte que corresponde a la parte baja del entero
Bytes	String	Escribe la cadena como una sucesión de bytes.
Char	int	Escribe un carácter (los 2 bytes más bajos del entero), el byte alto primero.
Chars	String	Escribe la cadena como una sucesión de carácteres (2 bytes por carácter).
Double	double	Convierte el double a un long usando el método doubleToLongBits de la clase Double y luego escribe el valor obtenido como una sucesión de 8 bytes, el byte alto primero.
Float	float	Convierte el valor float a un valor entero (int) usando el método floatToIntBits de la clase Float para luego escribirlo como u7n entero de 4 bytes, el byte alto primero.
Int	int	Escribe un entero en 4 bytes, byte alto primero.
Long	long	Escribe el entero largo en 8 bytes, byte alto primero.
Short	int	Escribe el entero corto en 2 bytes (los dos bytes más bajos del entero) byte alto primero.
UTF	String	Escribe una cadena en el flujo usando codificación UTF-8 modificada de manera que es independiente de la computadora.

Como se puede ver, este flujo sirve para escribir en disco imágenes (copias) del contenido de variables en memoria, siguiendo el patrón de bits dado para su codificación binaria. Por esta última caracterización, a los archivos creados con este tipo de flujos se les conoce como archivos binarios, esto es, que los bytes deben interpretarse como si fueran variables en memoria.

Es en este tipo de archivos donde realmente se puede utilizar el método skip, ya que el número de bytes que componen un registro lógico (que depende de la manera como lo tratemos de leer) es constante.

Conocemos ya todo lo que requerimos para proponer una nueva opción en nuestro menú, la que escribe y lee archivos binarios. Revisemos el código agregado o modificado que se encuentra en los listados 10.5 para lo que tiene que ver con el proceso de la opción y el listado 10.6 en la siguiente página para lo que tiene que ver con la opción misma.

Código 10.5 Declaraciones de flujos binarios (**MenuListaIO**)

```
 6:   static final int AGREGA = 1,
              ......
15:          LEERREGS = 9,
16:          GUARDARREGS = 10;
......
128:     public int daMenu(BufferedReader cons, ListaCurso miCurso)
129:                                              throws IOException   {
              ......
141:         DataInputStream archivoRegsIn = null;
142:         DataOutputStream archivoRegsOut = null;
              ......
156:             + "(9)\tLeer de archivo binario\n"
157:             + "(A)\tGuardar en archivo binario\n"
                  ......
170:         opcion = "0123456789AB".indexOf(sopcion);
              ......
```

Nuevamente optamos por declarar los flujos necesarios dentro del método que maneja el menú. La razón de esto es que estas opciones se pueden elegir en cualquier momento y más de una vez, en cada ocasión con flujos físicos distintos, por lo que hacerlos globales a la clase o, peor aún, al uso de la clase, amarraría a utilizar únicamente el flujo determinado antes de empezar, cuando existe la posibilidad de que no se elija esta opción o que, como ya mencionamos, se desee hacer varias copias de l;a información. En las líneas 15:, 16: 156: a 170: simplemente agregamos dos opciones al menú, y los mecanismos para manejarlas – posponemos por el momento el desarrollo de la opción correspondiente dentro del switch –. La decla-

ración de los flujos la hacemos en las líneas 141: y 142:. Tanto en este caso como en el los flujos BufferedReader y BufferedWriter podríamos haberlos declarado como objetos de las superclases correspondiente:

```
134:        Reader archivoIn = null;
135:        Writer archivoOut = null;


141:        InputStream archivoRegsIn = null;
142:        OutputStream archivoRegsOut = null;
```

y determinar la subclase correspondiente en el momento de construirlos. De haberlo hecho así, para usar los métodos que se agregaron al nivel de Buffered... – como readLine y writeLine – y Data...Stream – como readShort y writeShort – hubiésemos tenido que hacer *casting* para que el compilador los identificara. Pero, aún cuando requerimos básicamente un flujo de entrada y uno de salida, no podríamos tener nada más una declaración por función ya que la superclase común a Reader e InputStream es Object, y el usar a Object en la declaración hubiese requerido de *casting* prácticamente en cada momento de uso de los flujos. La asignación de un valor null en la declaración es para detectar, en su caso, si el flujo pudo construirse o no. Adicionalmente, dadas las características de las excepciones, donde puede haber código que no se ejecute, en el bloque catch no se tiene la seguridad de que a las variables se les haya asignado efectivamente un valor en el bloque try, por lo que el compilador no va a permitir que las variables inicien sin valor asignado.

En las líneas 156: y 157: del listado 10.5 simplemente agregamos las opciones correspondientes al menú que se despliega, mientras que en la línea 170: del mismo listado modificamos para que estas opciones sean reconocidas.

Código 10.6 Opciones de leer y escribir a archivo binario (**MenuListaIOReg**) 1/3

```
379:    case LEERREGS:
380:      try {
381:          sArchivo = pideNombreArch(cons, LEER);
382:          archivoRegsIn = new DataInputStream
383:              (new FileInputStream(sArchivo));
384:      } catch (FileNotFoundException e) {
385:          System.err.println("el archivo de entrada"
386:                             + (sArchivo!=null?sArchivo:"nulo")
387:                             + " no existe");
388:          throw e;
389:      } // end of try-catch
```

Código 10.6 Opciones de leer y escribir a archivo binario (**MenuListaIOReg**) 2/3

```
390:      try {
391:          short tam = archivoRegsIn.readShort();
392:          tamanhos = new short[tam + 1];
393:          tamanhos[0] = tam;
394:          for (int i = 1; i <= tam; i++)
395:              tamanhos[i] = archivoRegsIn.readShort();
396:          short maxt = 0;
397:          for (int i = 0; i <= tamanhos[0]; i++)
398:              maxt =(short)(Math.max(maxt, tamanhos[i]));
399:          bCadena = new byte[maxt];
400:          while (archivoRegsIn.read(bCadena, 0, tamanhos[1]) > 0) {
401:              nombre = new String(bCadena, 0, tamanhos[1]);
402:              for (int i = 2; i <= tamanhos[0]; i++) {
403:                  archivoRegsIn.read(bCadena, 0, tamanhos[i]);
404:                  switch (i) {
405:                  case 2:
406:                      cuenta = new String(bCadena, 0, tamanhos[i]);
407:                      break;
408:                  case 3:
409:                      carrera = new String(bCadena, 0, tamanhos[i]);
410:                      break;
411:                  case 4:
412:                      clave = new String(bCadena, 0, tamanhos[i]);
413:                      break;
414:                  default:
415:                      break;
416:                  } // end of switch (i)
417:              } // end of for (int i = 1; i <= tamanhos[0]; i++)
418:              if (miCurso == null) {
419:                  System.out.println("No existe miCurso");
420:                  throw new NullPointerException("Uuups");
421:              } // end of if (miCurso == null)
422:              miCurso.agregaEstFinal
423:                  (new Estudiante(nombre,cuenta,carrera, clave));
424:          } // end of while (archivoRegsIn.read(bCadena
425:      } catch (IOException e) {
426:          System.err.println("Error de I/O");
427:          throw e;
428:      } // end of try-catch
429:      finally {
430:          try {
431:              archivoRegsIn.close();
432:          } catch (IOException e) {
433:              System.err.println("No se pido cerrar el archivo");
434:          } // end of try-catch
435:      } // end of finally
436:      return LEERREGS;
```

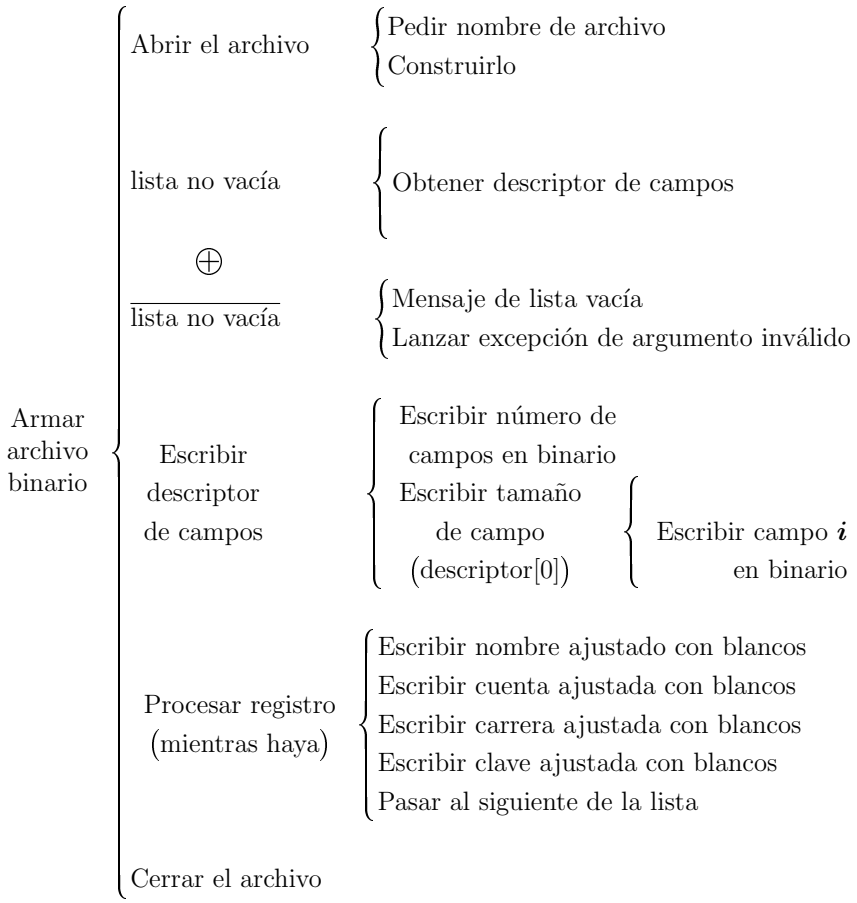
Código 10.6 Opciones de leer y escribir a archivo binario (**MenuListaIOReg**) 3/3

```
437:  case GUARDARREGS:
438:      try {
439:          sArchivo = pideNombreArch(cons,GUARDAR);
440:          archivoRegsOut = new DataOutputStream
441:              (new FileOutputStream(sArchivo));
442:          Estudiante lista = ((Estudiante)miCurso.daLista());
443:          if (lista != null) {
444:              tamanhos = lista.getTamanhos();
445:          } // end of if (lista != null)
446:          else {
447:              System.out.println("No hay nadie en la base"
448:                                 + " de datos");
449:              throw new IllegalArgumentException("Lista vacía");
450:          } // end of else
451:
452:          archivoRegsOut.writeShort(tamanhos[0]);
453:          for (int i = 1; i <= tamanhos[0]; i++) {
454:              archivoRegsOut.writeShort(tamanhos[i]);
455:          } // end of for (int i = 1; ...
456:          // Ahora procedemos a vaciar la base de datos
457:          while (lista != null ) {
458:              for (int i = 1; i <= tamanhos[0]; i++) {
459:                  nombre = (lista.daCampo(i) + blancos).
460:                      substring(0,tamanhos[i]);
461:                  System.out.println(lista.daCampo(i)+"\t"
462:                                     +i);
463:                  archivoRegsOut.writeBytes(nombre);
464:              } // end of for (int i = 1; i <= tamanhos[0]; i++)
465:              lista = lista.daSiguiente();
466:          } // end of while (lista != null)
467:          archivoRegsOut.flush();
468:          archivoRegsOut.close();
469:      } catch (IOException e) {
470:          System.err.println("No se qué pasó");
471:          throw new IOException("Algo salió mal");
472:      } // end of try-catch
473:      return GUARDARREGS;
```

Como se puede observar, las dos opciones son prácticamente paralelas, excepto que cuando una escribe la otra lee. Pasemos a revisar primero la opción de escritura, que sería el orden en que programaríamos para probar nuestra aplicación.

10.8.1. Escritura en archivos binarios

Figura 10.13 Algoritmo para escribir archivo binario



El algoritmo que se usó para escribir un archivo binario se encuentra en la figura 10.13 y corresponde a la discusión de la página 367 y mostrado en la figura 10.12 en la página 368.

En las líneas 440: a 442: del listado 10.6 se implementa la parte correspondiente a abrir el archivo seleccionado mediante el método pideNombreArch. Como este

enunciado puede lanzar una excepción de entrada/salida se enmarca en un bloque try. Sin embargo es difícil que la excepción sea lanzada – y en el caso de que esto suceda quiere decir que hay problemas que no puede resolver el usuario – no se conforma un bloque try sólo para esta operación. El constructor de la clase DataOutputStream requiere de un flujo de salida (OutputStream) que realice la conexión física. En este caso le pasamos como argumento un objeto de la clase FileOutputStream construido al vuelo, y que se construye utilizando el nombre del archivo deseado. Como FileOutputStream es una subclase de OutputStream no hay ningún problema ni de compilación ni de ejecución.

En las líneas 443: a 450: se verifica que la lista en memoria no esté vacía. De ser así se procede a obtener el descriptor de los campos (el encabezado del archivo binario) en un arreglo de enteros pequeños (short); si la lista está vacía se sale del menú lanzando una excepción, que es atrapada desde el método principal (main) de la aplicación.

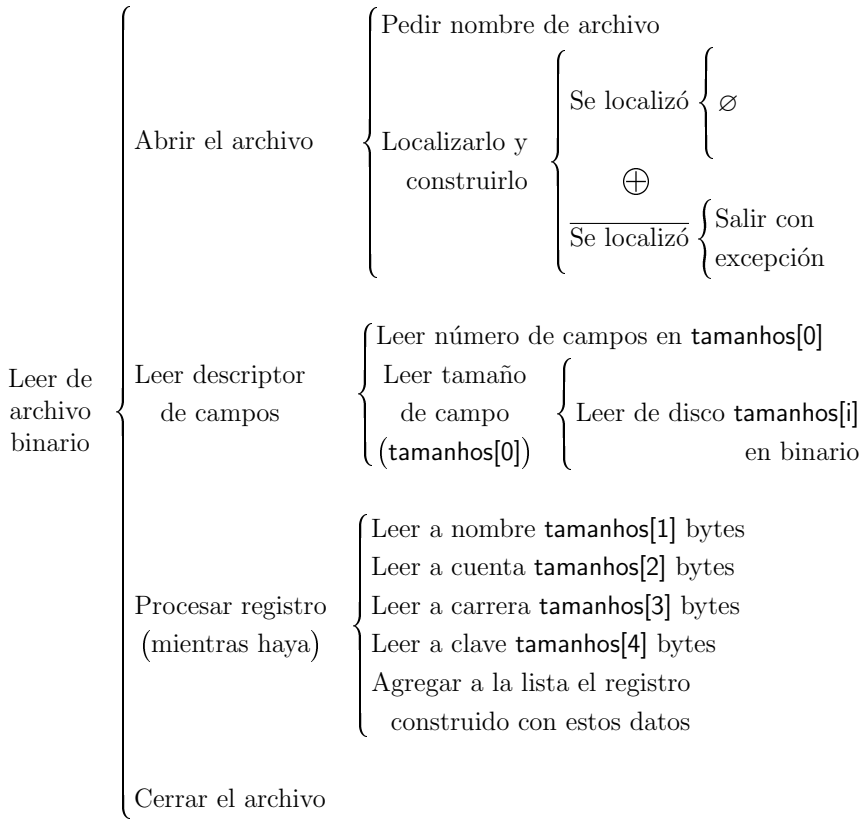
Se procede a escribir el encabezado del archivo binario en las líneas 452: a 455: como lo indica el diagrama del algoritmo, utilizando para ello el método writeShort de la clase DataOutputStream – ver documentación de la clase en las páginas 10.8 y 10.8 –. Una vez hecho esto se procede a escribir cada uno de los registros de la base de datos, como un arreglo de bytes, con cada campo ocupando el número de bytes que indica el encabezado del archivo – en las líneas 459: y 460: se ajusta el campo a su tamaño agregando blancos y en la línea 463: se escribe utilizando el método writeBytes de la clase DataOutputStream, que convierte una cadena a un arreglo de bytes –. Para escribir toda la lista seguimos nuestro algoritmo usual que recorre listas.

Finalmente en las líneas 467: y 468: se procede a cerrar el archivo. En el caso de archivos de disco esto es mucho muy importante, pues si no se hace el archivo no pasa a formar parte del sistema de archivos y por lo tanto no existirá más allá de la ejecución de la aplicación.

10.8.2. Lectura de archivos binarios

Como ya mencionamos antes, la lectura desde un archivo binario corre prácticamente paralelo a la escritura, pues se tiene que usar la misma máscara para leer que la que se utilizó para escribir. En la figura 10.14 mostramos en detalle el algoritmo para esta operación.

Figura 10.14 Algoritmo para leer de archivo binario



Comparemos ahora el algoritmo con el código del listado 10.6. La parte que corresponde a abrir el archivo y localizarlo se encuentra en las líneas 380: a 389:. En esta opción sí procesamos por separado la excepción que nos pueda dar la localización del archivo binario del que el usuario solicita leer porque es posible que no exista dicho archivo. Por eso, en lugar de la tradicional excepción IOException, acá tratamos de atrapar una excepción que nos indica que no se encontró el archivo. Igual que en el caso anterior, sin embargo, salimos con una excepción del método, pues tenemos que regresar a solicitar otra opción.

De manera similar a como lo hicimos para escribir, construimos un flujo de la

clase DataInputStream y le pasamos como argumento un objeto de la clase FileInputStream, que es subclase de InputStream, esto último lo que pide el constructor. Una vez abierto el flujo – si llegamos a la línea 390: – procedemos a adquirir la descripción de los campos en el archivo binario.

En la línea 391: leemos el número de campos de cada registro almacenado en el archivo en los primeros dos bytes del mismo, usando para ello formato de short; y en la línea 392: construimos un arreglo para almacenar ahí los tamaños de cada campo. En las líneas 393: a 395: leemos con máscara de short cada uno de los tamaños y los almacenamos en el arreglo de shorts tamanhos.

Una vez que tenemos esto, procedemos a leer del flujo arreglos de k bytes, donde k está dado píe la posición correspondiente de tamanhos. Para poder hacer esto en una iteración construimos un arreglo de bytes del máximo tamaño de los campos – esto se obtiene en las líneas 396: a 400: – utilizando para ello el método read(byte[] b, int off, int len) que me permite ir leyendo pedazos de len bytes. Como en el caso de la lectura de cadenas, intentamos primero leer el primer campo, y si esta lectura nos indica que no pudo leer damos por terminado el flujo – línea 401: –. En cambio, si pudo leer al primer campo, procedemos a leer tantos campos como nos indique tamanhos[0], en "automático", esto es, sin la garantía de que se encuentren en el flujo – líneas 403: a 417: – dejando en manos del mecanismo de excepciones si hay algún problema. Para no tener que llamar a cada campo por su nombre, simplemente usamos un switch – líneas 404: a 417: que convierte al arreglo de bytes en una cadena, que es lo que espera la clase Estudiante.

A continuación, en las líneas 422: y 423: procedemos a agregar a la base de datos el registro completo recién leído, construyendo el registro al vuelo.

Si hay algún error de entrada/salida, la excepción correspondiente se atrapa y se sale del método, mandando un mensaje alusivo y repitiendo la excepción. En esta opción presentamos una cláusula finally, ya que queremos que, aunque haya un error a la mitad, el flujo se cierre para liberar recursos. Esto se hace en las líneas 429: a 435:; sin embargo, tenemos que poner el enunciado que cierra el archivo en un bloque try pues puede lanzar una excepción. Si sucede esto último, simplemente absorbemos la excepción, ya que no importa que haya pasado se va a salir del método.

Avance del flujo sin leer

Como en el archivo que estamos construyendo tenemos tamaño fijo de los registros, sabemos en qué byte empieza, digamos, el i-ésimo registro. Todo lo que tenemos que hacer es leer el número de campos y el tamaño de cada campo para calcular el tamaño del registro, y desde ese punto saltar $i - 1$ registros de tamaño k, donde k es el tamaño de registro calculado. El único problema que tenemos

acá es que el acceso al flujo sigue siendo secuencial, así que los bytes que saltemos en la lectura ya no los podemos regresar[5]. n el listado 10.7 agregamos una opción al menú para poder acceder al i-ésimo registro, siempre y cuando se haga al principio del proceso. Esto pudiéramos usarlo para descartar un cierto número de registros y leer únicamente a partir de cierto punto.

Código 10.7 Salto de bytes en lectura secuencial (**MenuListaIOReg**) 1/3

```
481:     case LEEREGISTROK:
482:       // Pedir el nombre del flujo
483:       try {
484:           sArchivo = pideNombreArch(cons, LEER);
485:           archivoRegsIn = new DataInputStream
486:               (new FileInputStream(sArchivo));
487:       } catch (FileNotFoundException e) {
488:           System.err.println("el␣archivo␣de␣entrada"
489:                              + (sArchivo!=null?sArchivo:"nulo")
490:                              + "␣no␣existe");
491:           throw e;
492:       } // end of try-catch
493:       // Calcular el tamanho del registro
494:       int tamR = 0;
495:       try {
496:           short tam = archivoRegsIn.readShort();
497:           tamanhos = new short[tam + 1];
498:           tamanhos[0] = tam;
499:           tamR = 0;
500:           for (int i = 1; i <= tam; i++) {
501:               tamanhos[i] = archivoRegsIn.readShort();
502:               tamR += tamanhos[i];
503:           } // end of for (int i = 1; i <= tam; i++)
504:       } catch (IOException e) {
505:           System.err.println("No␣pude␣leer␣parámetros");
506:           return LEEREGISTROK;
507:       } // end of catch
508:       // Calcular el número total de registros en el flujo
509:       int fileSize = 0;
510:       try {
511:           fileSize = archivoRegsIn.available() / tamR;
512:
513:       } catch (IOException e) {
514:           System.err.println("Error␣al␣calcular␣Núm␣bytes");
515:       } // end of try-catch
516:       // Pedir el registro solicitado que este en rangos
517:       int numR = 0;
```

Código 10.7 Salto de bytes en lectura secuencial (**MenuListaIOReg**)2/3

```
518:        try {
519:            System.out.print("Ahora␣dime␣el␣numero␣de␣registro␣(0.."
520:                             + (fileSize - 1) + ")␣-->");
521:            subcad = cons.readLine();
522:            numR = 0;
523:            for (int i = 0; i < subcad.length(); i++) {
524:                numR = numR*10 + subcad.charAt(i) - '0';
525:            } // end of for (int i = 0; i < subcad.length(); i++)
526:            if ( numR < 0 || numR >= fileSize) {
527:                System.out.println("Hay␣menos␣de␣" + numR
528:                                   + ".␣Del␣0␣al␣"
529:                                   + (fileSize - 1));
530:                return LEEREGISTROK;
531:            }
532:        } catch (IOException e) {
533:            System.out.println("Error␣al␣leer␣n'umero␣de␣registro");
534:        } // end of try-catch
535:        // Saltar el numero de bytes requeridos para ubicarse
536:        // en el registro solicitado.
537:        int aSaltar = (numR - 1) * tamR;
538:        int pude = 0;
539:        try {
540:            // Saltar bytes
541:            pude = archivoRegsIn.skipBytes(aSaltar);
542:            // Si es que hubo los suficientes
543:            if (pude >= aSaltar) {
544:                bCadena = new byte[tamR];
545:                // Leemos el registro solicitado.
546:                archivoRegsIn.read(bCadena, 0, tamanhos[1]);
547:                nombre = new String(bCadena, 0, tamanhos[1]);
548:                for (int i = 2; i <= tamanhos[0]; i++) {
549:                    archivoRegsIn.read(bCadena, 0, tamanhos[i]);
550:                    switch (i) {
551:                    case 2:
552:                        cuenta = new String(bCadena, 0, tamanhos[i]);
553:                        break;
554:                    case 3:
555:                        carrera = new String(bCadena, 0, tamanhos[i]);
556:                        break;
557:                    case 4:
558:                        clave = new String(bCadena, 0, tamanhos[i]);
559:                        break;
560:                    default:
561:                        break;
562:                    } // end of switch (i)
563:                } // end of for (int i = 1; i <= tamanhos[0]; i++)
```

Código 10.7 Salto de bytes en lectura secuencial (**MenuListaIOReg**) 3/3

```
554:                // Se arma un objeto de la clase Estudiante
555:                Estudiante nuevo = new Estudiante(nombre, cuenta,
556:                                                  carrera, clave);
557:                System.out.println(nuevo.daRegistro());
558:            } // end of if (pude >= aSaltar)
559:        } catch (IOException e) {
560:            System.out.println("Hubo error");
561:        } // end of try-catch
562:        return LEEREGISTROK;
```

10.8.3. Acceso semi directo a archivos binarios

Un archivo binario tiene un tamaño fijo de registros; en muchas ocasiones los registros, o la información propiamente dicha, viene precedida de un descriptor de los registros – lo que antes llamamos el encabezado del archivo – pero esta información aparece una única vez y siempre al principio del archivo. Los flujos que hemos visto hasta ahora, a'un los binarios, son secuenciales, esto es, para leer el registro $\boldsymbol{i}$ hay que leer previamente los $\boldsymbol{i-1}$ registros que lo preceden, o en todo caso saltar los bytes correspondientes a $\boldsymbol{i-1}$ registros. Debemos notar que incluso con los saltos (skip), éstos son siempre hacia el final del archivo: lo que no podemos hacer es querer leer el registro $\boldsymbol{i}$ y posteriormente el registro $\boldsymbol{j}$ con $\boldsymbol{j < i}$.[6]

Sin embargo, es muy común querer un acceso directo[7]. Por ejemplo, supongamos que tenemos una base de datos ordenada alfabéticamente y queremos listar en orden inverso. Con los flujos esto equivaldría a cerrar y abrir el archivo por cada registro. Lo ideal es que sin costo adicional, pudiéramos recorrer el archivo de atrás hacia adelante. También se presenta esta necesidad en las bases de datos, donde se guarda en disco una tabla con alguna llave (*key*) que identifica a un registro, y a continuación la posición de ese registro en el archivo. Se podría hacer una búsqueda inteligente sobre la tabla (por ejemplo, búsqueda binaria) que requiere la posibilidad de ir hacia adelante o hacia atrás en la lectura del archivo.

[6]Para poder hacer esto en archivos secuenciales tenemos que cerrar y volver a abrir el archivo para que se vuelva a colocar al principio del mismo y poder saltar hacia el final del archivo. Otra opción es que el archivo tenga implementado los métodos mark y reset.

En Java tenemos una clase que nos da esta facilidad y que es la clase RandomAccessFile. Hay que tener presente que aunque esta clase maneja archivos en disco no hereda de ningún flujo (stream) o de lector/escritor (Reader/Writer), sino que hereda directamente de Object, aunque sí se encuentra en el paquete java.io.

Los ejemplares de esta clase proveen tanto lectura como escritura en el mismo objeto. en general podemos pensar en un archivo de acceso directo como un arreglo en disco, donde cada elemento del arreglo es un registro y al cual queremos tener acceso directamente a cualquiera de los registros sin seguir un orden predeterminado.

Con un archivo de este tipo tenemos siempre asociado un *apuntador de archivo* (en adelante simplemente apuntador), que se encarga de indicar en cada momento a partir de donde se va a realizar la siguiente lectura/escritura. Si el apuntador está al final del archivo y viene una orden de escritura, el archivo simplemente se extiende (el arreglo crece); si estando al final del archivo viene una orden de lectura, la máquina virtual lanzará una excepción EOFException. Si se intenta realizar alguna operación después de que el archivo fue cerrado, se lanzará una IOException. Se tiene un método que se encarga de mover al apuntador, lo que consigue que la siguen te lectura/escritura se lleve a cabo a partir de la posición a la que se movió el apuntador. Estas posiciones son absolutas en términos de bytes.

Este tipo de archivos se pueden usar para lectura, escritura o lectura/escritura, dependiendo de qué se indique al construir los ejemplares.

Si bien no hereda de ninguna de las clases Stream, como ya dijimos, implementa a las interfaces DataOutput, DataInput y Closeable. Como se pueden imaginar, las dos primeras también son implementadas por DataOutputStream y DataInputStream, por lo que tendremos prácticamente los mismos métodos que ya conocemos de estas dos últimas clases, todos en una única clase. Revisaremos únicamente aquellos métodos que no conocemos todavía, dando por sentado que contamos con los métodos para leer y escribir de DataInputStream y DataOutputStream.

public class RandomAccessFile implements DataOutput, DataInput, Closeable

Constructores:

public RandomAccesFile(File file , String mode)
throws FileNotFoundException

[7] Llamamos *directo* a lo que en inglés se conoce también como *random* – aleatorio –, porque consideramos que aquél es un término más adecuado. *Aleatorio* tiene un sentido de no determinismo y lo que se hace con este tipo de acceso es poder llegar directamente (**no secuencialmente**) a un cierto registro.

class RandomAccessFile (continúa...)

Nuevamente tenemos que File representa a un archivo físico, ya sea que ya ha sido construido a través de un flujo o que se le dé la identificación completa de un archivo en disco. La cadena mode representa el tipo de uso que se le va a dar al archivo y puede ser:

- "r" Abre el archivo sólo para lectura. Si con un archivo abierto en este modo se intenta escribir, se lanzará una IOException1'.
- "rw" El archivo se abre para lectura/escritura. Si el archivo no existe, se intenta crear nuevo.
- "rws" Igual que "rw", excepto que exige que todo cambio al archivo o al descriptor (meta-datos) del archivo se refleje inmediatamente en el dispositivo, que se haga sincronizado con el enunciado.
- "rwd" Igual que "rws", excepto que los meta-datos no tienen que actualizarse de manera sincronizada.

Estos dos últimos modos son muy útiles para garantizar que si hay algún problema con el sistema, todo lo que se escribió desde la aplicación en efecto se vea reflejado en el dispositivo local. Si el dispositivo no es local, no hay garantía de que esto suceda.

Las excepciones que pueden ser lanzadas (aunque no es necesario anunciarlas a todas) son:

IllegalArgumentException Si el modo del constructor no es uno de los especificados.

FileNotFoundException Si no se encontró el archivo solicitado y se intentó abrir sólo para lectura.

SecurityException Si se intentó abrir un archivo para lectura o para lectura/escritura que el manejador de la seguridad no permita leer, o para lectura/escritura que el manejador de seguridad no permita escribir.

public RandomAccesFile(String name, String mode)
throws FileNotFoundException

En este caso la primera cadena da una identificación en disco del vuelve a ser el modo en que vamos a usar el archivo. Se comporta exactamente igual que el primer constructor que mostramos.

Métodos adicionales a los de *DataInput* y *DataOutput*

class RandomAccessFile (continúa...)

En el caso de los archivos de acceso directo siempre existe el problema de que se trate de leer más allá del fin de archivo. Por ello, prácticamente todos los métodos que intentan leer más de un carácter incondicionalmente llegando al final del archivo sin haber leído todos los bytes y que son redefinidos lanzan la excepción EOFException.

Como ésta es una subclase de IOException no hay necesidad de modificar el encabezado – que por otro lado no se puede – para que avise que también puede lanzar una EOFException, que se refiere a tratar de leer más allá del fin de archivo.

public long length() **throws** IOException

Regresa el tamaño del archivo en bytes.

void setLength(**long** newLength) **throws** IOException

Establece un nuevo tamaño para el archivo. Si el tamaño anterior era mayor, se trunca el archivo al nuevo tamaño, perdiéndose lo que había más allá del nuevo tamaño. Si el tamaño anterior era menos, se reserva espacio para agrandar el archivo; sin embargo hay que tener en cuenta que el contenido del espacio agregado no estará definido. En el caso de que el archivo sea truncado y que el filePointer esté más allá del nuevo final del archivo, el apuntador se colocará al final del archivo.

public void seek(**long** pos) **throws** IOException

Coloca el apuntador del archivo (filePointer) en la posición dada por pos, contado en bytes a partir del principio del archivo. Ésta posición puede estar más allá del fin de archivo sin que se lance una excepción de fin de archivo (EOFException), pero sin que cambie tampoco el tamaño del archivo. Sin embargo, si una vez colocado el apuntador más allá del final, se lleva a cabo una escritura, esta acción si modificará el tamaño del archivo. La excepción se lanza si **pos** $< \mathbf{0}$ u ocurre un error de entrada/salida.

public long getFilePointer () **throws** IOException

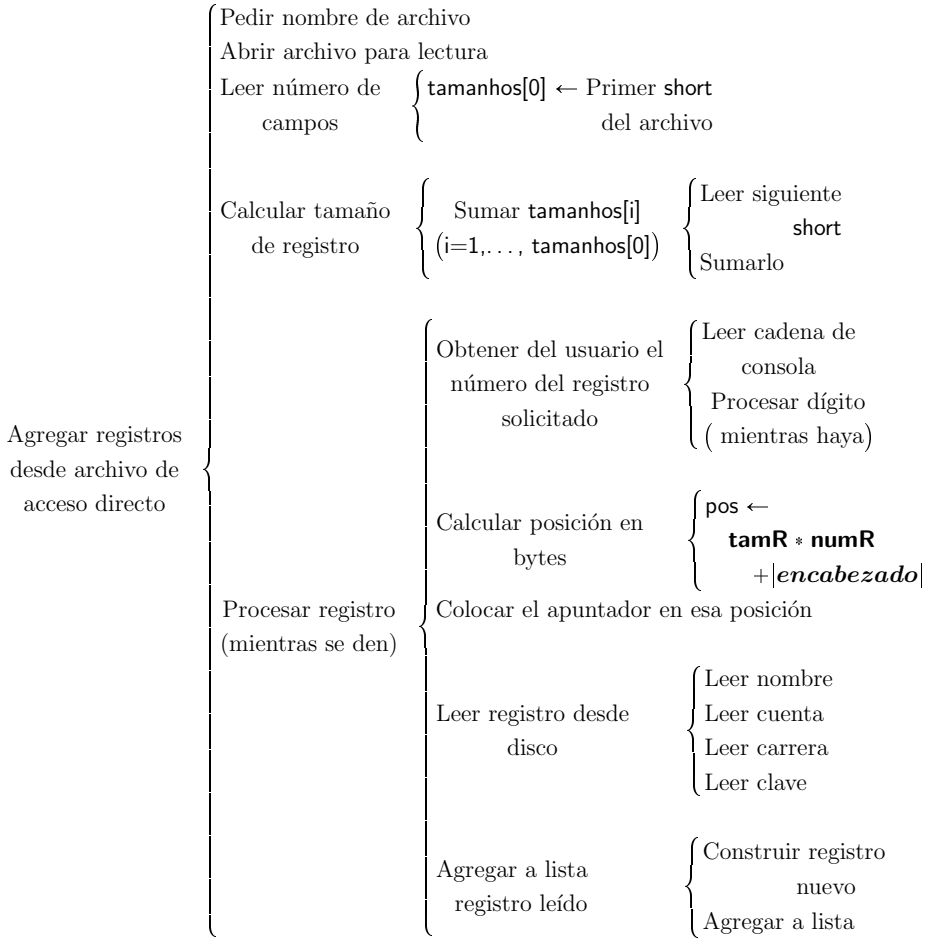
Regresa la distancia al principio del archivo, en bytes, de la posición actual del archivo (el próximo byte que va a ser leído o escrito).

Con esto ya tenemos las herramientas necesarias para acceder al disco con acceso directo.

10.8.4. Lectura directa de registros

Como ya mencionamos, cualquier archivo en disco (o algún otro dispositivo de almacenamiento en bytes) puede ser leído o escrito bajo cualquier tipo de flujo; el tipo de flujo nos indica únicamente cómo interpretar los bytes: **no describe** el contenido del archivo.

Figura 10.15 Algoritmo para agregar registros desde archivo de acceso directo



El único problema que enfrentamos es si suponemos un flujo en el que queramos leer cadenas y no tenemos carácteres de fin de línea en el archivo; en este caso una única lectura nos daría **todo** el contenido del archivo, terminando la cadena con un fin de archivo. En este sentido, **sí** hay que distinguir (desde la aplicación) cuando estamos trabajando con un archivo de cadenas o de registros de tamaño fijo.

La lectura directa nos permite "brincar" dentro del archivo, hacia adelante y hacia atrás, buscando un byte en una posición determinada. Lo que quisiéramos hacer con nuestra base de datos, guardada en un archivo en disco, es poder leer y agregar *en desorden* a los registros que se encuentran en el disco. También debe ser posible agregar únicamente a un subconjunto de estos registros. El algoritmo para este caso se encuentra en la figura 10.15 en la página anterior.

Podemos revisar cada uno de estos subprocesos en términos del diagrama. Por ejemplo, pedir el nombre del archivo consiste de exactamente el mismo proceso que en el caso del acceso al ***k***-ésimo registro. La implementación se puede ver en el listado 10.8. La parte correspondiente a la lectura y apertura del archivo tienen que estar en un bloque `try`, ya que ambas operaciones pueden lanzar excepciones – líneas 605: a 612:. Lo primero que hacemos es solicitar del usuario el nombre del archivo en disco que vamos a utilizar – línea 606: – y a continuación lo tratamos de abrir exclusivamente para lectura, lo que que hacemos poniendo como segundo argumento del constructor `"r"` en la línea 607:.

Si se lanza la excepción de archivo no encontrado, simplemente emitimos un mensaje acorde y salimos a mostrar nuevamente el menú – líneas 608: a 612: –,

mientras que si lo que tenemos es un error de entrada/salida realmente ya no podemos hacer nada y exportamos la excepción – líneas 613: y 614:.

Lo siguiente que queremos hacer es leer el encabezado del archivo para calcular el tamaño del registro. También esto se lleva a cabo de la misma manera que lo hicimos en la lectura de archivos binarios, por lo que ya no es necesaria una explicación. El código se encuentra en el listado 10.9.

Código 10.8 Lectura del nombre del archivo y apertura del mismo (**case LEERDIRECTO**) 1/2

```
603:     case LEERDIRECTO:
604:         // Pedir el nombre del flujo
605:         try {
606:             sArchivo = pideNombreArch(cons, opcion);
607:             archivoRndm = new RandomAccessFile (sArchivo, "r");
608:         } catch (FileNotFoundException e) {
```

Código 10.8 Lectura del nombre del archivo y apertura del mismo (case **LEERDIRECTO**) 2/2

```
609:             System.err.println("el archivo de entrada"
610:                                + (sArchivo!=null?sArchivo:"nulo")
611:                                + " no existe");
612:             return LEERDIRECTO;
613:         } catch (IOException e) {
614:             throw e;
615:         } // end of try-catch
```

Código 10.9 Cálculo del tamaño del registro (case **LEERDIRECTO**)

```
616:                 // Calcular el tamaño del registro
617:         tamR = 0;
618:         numBytes = 0;
619:         try {
620:             short tam = 0;
621:             tam = archivoRndm.readShort();
622:             numBytes = 2;
623:             tamanhos = new short[tam + 1];
624:             tamanhos[0] = tam;
625:             tamR = 0;
626:             for (int i = 1; i <= tam; i++) {
627:               tamanhos[i] = archivoRndm.readShort();
628:               numBytes+=2; tamR += tamanhos[i];
629:             } // end of for (int i = 1; i <= tam; i++)
630:         } catch (IOException e) {
631:             System.err.println("No pude leer parámetros");
632:             return opcion;
633:         } // end of catch
634:     // Calcular el número total de registros en el flujo
635:         // para detectar posiciones erróneas de registros
636:         fileSize = 0;
637:         try {
638:             fileSize = archivoRndm.length() / tamR;
639:         } catch (IOException e) {
640:             System.err.println("Error al calcular Núm bytes");
641:         } // end of try-catch
```

Una vez que calculamos el tamaño del encabezado (numBytes), el tamaño del registro (tamR) y el número de registros lógicos en el archivo (fileSize) procedemos a iterar, pidiéndole al usuario el número de registro, tantos como desee, del registro que desea leer y agregar a la lista en memoria. Esto se lleva a cabo en las líneas 642: a 667: en el listado 10.10 en la siguiente página.

Código 10.10 Petición del número de registro al usuario (case **LEERDIRECTO**)

```
642:        while (numR >= 0) {
643:          // Pedir el registro solicitado que este en rangos
644:          numR = 0;
645:          try {
646:             System.out.print("Ahora␣dime␣el␣número␣de␣registro"
647:                              + "(0.." + (fileSize - 1)
648:                              + ",␣-1␣para␣terminar)"
649:                              + "␣a␣agregar␣-->");
650:             subcad = cons.readLine();
651:             if (subcad.charAt(0) < '0' || subcad.charAt(0) >= '9') {
652:                 numR = -1;
653:                 continue;
654:             } // end of if
655:             numR = 0;
656:             for (int i = 0; i < subcad.length(); i++) {
657:                 numR = numR*10 + subcad.charAt(i) - '0';
658:             } // end of for
659:             if ( numR < 0 || numR >= fileSize) {
660:                 System.out.println("Hay␣menos␣de␣" + numR
661:                                    + ".␣Del␣0␣al␣"
662:                                    + (fileSize - 1));
663:                 return opcion;
664:             }
665:          } catch (IOException e) {
666:            System.out.println("Error␣al␣leer␣número␣de␣registro");
667:          } // end of try-catch
```

Elegimos leer del usuario una cadena, para evitar errores de lectura en caso de que el usuario no proporcione un entero – línea 650:. Calculamos el entero correspondiente usando la regla de Horner, que proporciona una manera sencilla, de izquierda a derecha, de calcular un polinomio. Supongamos que tenemos un polinomio

$$c_n x^n + c_{n-1} x^{n-1} + \ldots + c_0$$

Podemos pensar en un número en base b como un polinomio donde $x = b$ y tenemos la restricción de que $0 \leqslant c_i < b$. En este caso para saber el valor en base **10** evaluamos el polinomio. La manera fácil y costosa de hacerlo es calculando cada una de las potencias de b para proceder después a multiplicar c_i por b^i.

$$P(x) = \sum_{i=0}^{n} c_i x^i$$

Pero como acabamos de mencionar, esta es una manera costosa y poco elegante

de hacerlo. La regla de Horner nos dice:

$$\boldsymbol{P(x) = c_0 + x(c_1 + x(c_2 + x(\ldots))\ldots)}$$

lo que nos permite evaluar el polinomio sin calcular previamente las potencias de $\boldsymbol{b}$. Por ejemplo, el número base **10** 8725 lo podemos expresar como el polinomio

$$\mathbf{8 * 10^3 + 7 * 10^2 + 2 * 10^1 + 5 * 10^0 = 8000 + 700 + 20 + 5 = 8725}$$

Si usamos la regla de Horner lo calculamos de la siguiente manera:

$$\mathbf{5 + 10(2 + 10(7 + 10(8)))}$$

Pero como tenemos que calcular de adentro hacia afuera, el orden de las operaciones es el siguiente:

$$\mathbf{(((8 * 10) + 7) * 10 + 2) * 10 + 5}$$

que resulta en la siguiente sucesión de operaciones:

$$\begin{aligned}\mathbf{8 * 10} &= \mathbf{80 + 7}\\ &= \mathbf{87 * 10} = \mathbf{870 + 2}\\ &= \mathbf{872 * 10} = \mathbf{8720 + 5}\\ &= \mathbf{8725}\end{aligned}$$

lo que permite leer los dígitos de izquierda a derecha e ir realizando las multiplicaciones y sumas necesarias. La ventaja de esta regla es que cuando leemos el **8**, por ejemplo, no tenemos que saber la posición que ocupa, sino simplemente que es el que está más a la izquierda. Dependiendo de cuántos dígitos se encuentren a su derecha va a ser el número de veces que multipliquemos por **10**, y por lo tanto la potencia de **10** que le corresponde. Este algoritmo se encuentra codificado en las líneas 655: a 658:.

En las líneas 651: a 654: verificamos que el usuario no esté proporcionando un número negativo (que empieza con '-'). Si es así, damos por terminada la sucesión de enteros para elegir registros.

Quisiéramos insistir en que no importa si el flujo es secuencial o de acceso directo, una lectura se hace **siempre** a partir de la posición en la que se encuentra el flujo. Si se acaba de abrir esta posición es la primera – la cero (0) –. Conforme se hacen lecturas o escrituras el flujo o archivo va avanzando; en los flujos secuenciales de entrada, mediante el método `skip` se puede avanzar sin usar los bytes saltados, pero siempre hacia adelante. En cambio, en los archivos de acceso directo se cuenta

con el comando seek que es capaz de ubicar la siguiente lectura o escritura a partir de la posición dada como argumento, colocando el apuntador de archivo en esa posición.

En el listado 10.11 se encuentra el código que corresponde a la ubicación del apuntador del archivo, frente al primer byte del registro solicitado por el usuario en la iteración actual.

Código 10.11 Posicionamiento del apuntador del archivo y lectura (**case LEERDIRECTO**)

```
687:            try {
688:                // Saltar el numero de bytes requeridos para ubicarse
689:                // en el registro solicitado.
690:                aSaltar = numR*tamR + numBytes;
691:                archivoRndm.seek(aSaltar);
692:                bCadena = new byte[tamR];
693:                // Leemos el registro solicitado.
694:                archivoRndm.read(bCadena, 0, tamanhos[1]);
695:                nombre = new String(bCadena, 0, tamanhos[1]);
696:                for (int i = 2; i <= tamanhos[0]; i++) {
697:                    archivoRndm.read(bCadena, 0, tamanhos[i]);
698:                    switch (i) {
699:                    case 2:
700:                        cuenta = new String(bCadena, 0, tamanhos[i]);
701:                        break;
702:                    case 3:
703:                        carrera = new String(bCadena, 0, tamanhos[i]);
704:                        break;
705:                    case 4:
706:                        clave = new String(bCadena, 0, tamanhos[i]);
707:                        break;
708:                    default:
709:                        break;
710:                    } // end of switch (i)
711:                } // end of for (int i = 1;
712:                // Se arma un objeto de la clase Estudiante
713:                Estudiante nuevo = new Estudiante
714:                    (nombre, cuenta, carrera, clave);
715:                if (miCurso == null) {
716:                    System.out.println("No existe Curso");
717:                    throw new NullPointerException("Uuups");
718:                } // end of if (miCurso == null)
719:                miCurso.agregaEstFinal
720:                    (new Estudiante(nombre,cuenta,carrera, clave));
721:            } catch (IOException e) {
722:                System.out.println("Hubo error en LEERDIRECTO");
723:            } // end of try-catch
724:        } // end while se pidan registros
```

En la línea 690: calculamos la posición que corresponde al entero dado en esta iteración y en la línea 691: colocamos al apuntador del archivo frente al siguiente registro a leer. Como los registros están numerados del 0 en adelante, el número de registros que se tienen que saltar son precisamente el entero proporcionado por el usuario. Una vez colocados frente al registro deseado procedemos a leer el registro desde disco – líneas 692: a 711: –. Una vez leído el registro procedemos a agregarlo a la lista que se está construyendo – líneas 713: a 720: –. En este bloque hay varias operaciones que pueden lanzar una excepción, por lo que se colocan en un bloque `try`. En caso de que haya algún problema, simplemente descarta esta escritura y procede a pedir el siguiente número de registro.

10.8.5. Escritura directa de registros

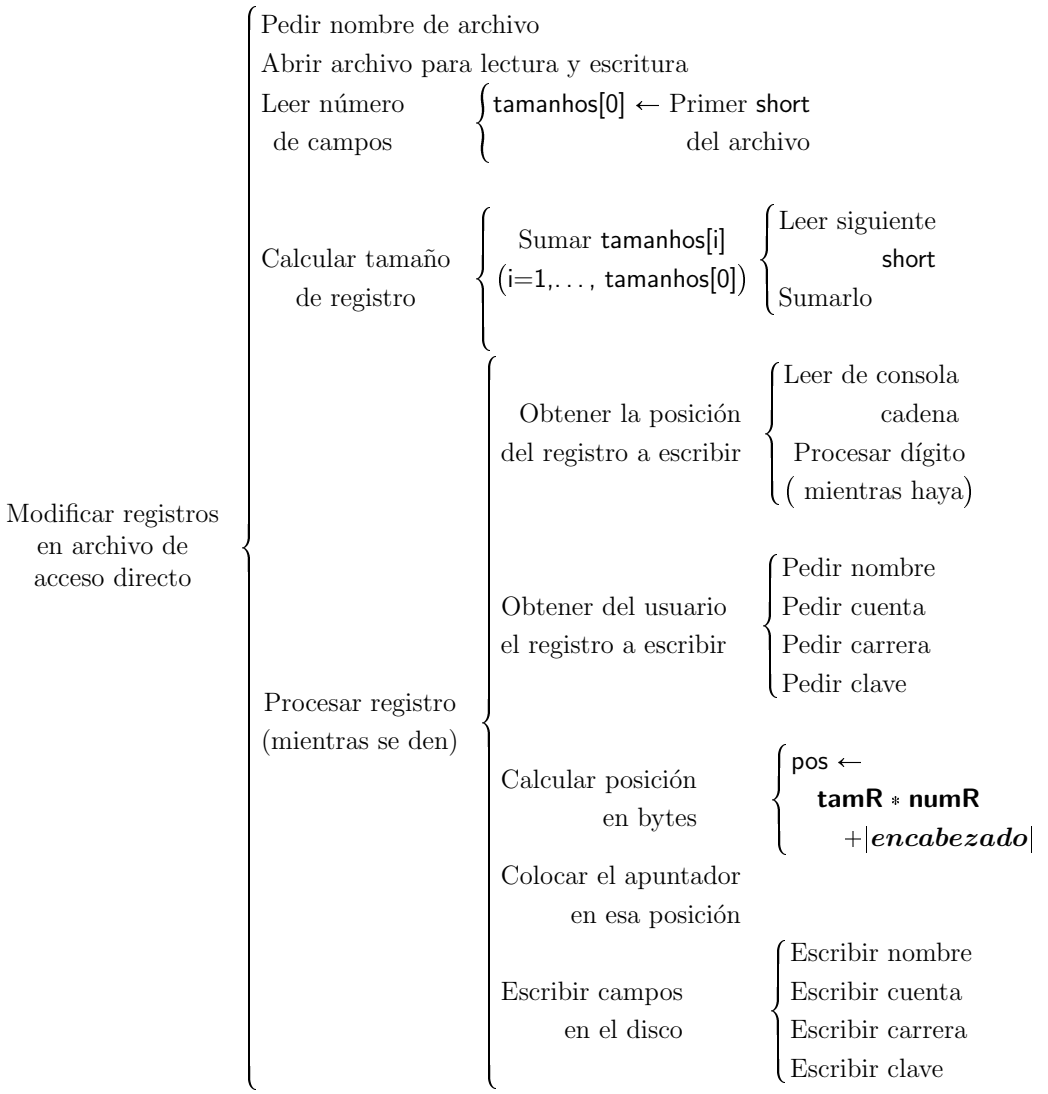
En un archivo de acceso directo podemos tener varias situaciones:

1. Queremos guardar en un archivo de acceso directo a los registros de la lista en memoria.
2. Queremos agregar registros al final del archivo.
3. Deseamos sustituir un registro en el archivo por otro proporcionado por el usuario.

Las primeras dos situaciones no guardan ninguna diferencia con la de estar trabajando con archivos binarios; insistimos en que las opciones que vamos a tener con un archivo en disco depende "del color del cristal con el que lo veamos". En general, si tenemos el encabezado del archivo y registros de tamaño uniforme, podemos verlo como un archivo binario o como uno de acceso directo. Por esta razón nos concentraremos en la tercera situación, aclarando que la posición que se dé para escribir el registro puede estar más allá del fin de archivo, lo que resultará, en este tipo de archivos, en que se extienda el archivo, dejando como indefinidos los bytes que se encuentren entre el final anterior y el actual. El algoritmo para esta opción se encuentra en la figura 10.16.

Por supuesto que en otras aplicaciones puede ser la misma aplicación la que modifique un registro y lo reescriba. En el contexto de nuestra aplicación pensaremos que el usuario nos da la posición del registro y el registro a escribir en esa posición – esto para poder seguir trabajando en este contexto. El algoritmo para esta opción corre paralelo al que dimos para lectura de archivo de acceso directo, excepto que además de pedir número de registro, la aplicación tiene que pedir el registro a escribir. Por lo anterior, esta opción es una combinación de la opción que agrega un registro y de la que lee de un archivo de acceso directo. El código correspondiente se encuentra en al listado 10.12.

Figura 10.16 Algoritmo para sobrescribir registros en archivo de acceso directo



Código 10.12 Opción de modificar registros (**case GUARDARDIRECTO**)1/2

```
742:     case GUARDARDIRECTO:
743:       Estudiante nuevo = null;
744:       // Pedir el nombre del flujo
745:       try {
746:           sArchivo = pideNombreArch(cons, opcion);
747:               archivoRndm = new RandomAccessFile
748:                   (sArchivo, "rw");
749:       } catch (FileNotFoundException e) {
750:           System.err.println("el archivo de entrada"
751:                              + (sArchivo!=null?sArchivo:"nulo")
752:                              + " no existe");
753:           return opcion;
754:       } catch (IOException e) {
755:           throw e;
756:       } // end of try-catch
757:       // Calcular el tamaño del registro
758:       tamR = 0;
759:       numBytes = 0;
760:       try {
761:           short tam = 0;
762:           tam = archivoRndm.readShort();
763:           numBytes = 2;
764:           tamanhos = new short[tam + 1];
765:           tamanhos[0] = tam;
766:           tamR = 0;
767:           for (int i = 1; i <= tam; i++) {
768:               tamanhos[i] = archivoRndm.readShort();
769:               numBytes+=2;
770:               tamR += tamanhos[i];
771:           } // end of for (int i = 1; i <= tam; i++)
772:       } catch (IOException e) {
773:           System.err.println("No pude leer parámetros");
774:           return opcion;
775:       } // end of catch
776:       // Calcular el número total de registros en el flujo
777:       fileSize = 0;
778:       try {
779:           fileSize = archivoRndm.length() / tamR;
780:       } catch (IOException e) {
781:           System.err.println("Error al calcular Num bytes");
782:       } // end of try-catch (int i = 1; i <= tamanhos[0]; i++)
783:       while (numR >= 0) {
784:         // Pedir el registro solicitado que este en rangos
785:         try {
```

Código 10.12 Opción de modificar registros (**case GUARDARDISCO**)2/2

```
786:           System.out.print("Ahora␣dime␣el␣numero␣de␣registro␣(0.."
787:                            + ",␣-1␣para␣terminar)"
788:                            + "␣a␣escribir␣-->");
789:           subcad = cons.readLine();
790:           char cual = subcad.charAt(0);
791:           if (cual < '0' || cual >= '9') {
792:               numR = -1;
793:               continue;
794:           } // end of if
795:           numR = 0;
796:           for (int i = 0; i < subcad.length(); i++)
797:               numR = numR*10 + subcad.charAt(i) - '0';
798:         } catch (IOException e) {
799:           System.out.println("Error␣al␣leer␣número␣de␣registro");
800:         } // end of try-catch
801:         // Pedir los campos a escribir
802:         try {
803:           nombre = pideNombre(cons);
804:           cuenta = pideCuenta(cons);
805:           carrera = pideCarrera(cons);
806:           clave = pideClave(cons);
807:           nuevo = new Estudiante(nombre, cuenta, carrera, clave);
808:         } catch (IOException e) {
809:           System.out.println("Error␣al␣leer␣datos␣del␣"
810:                              + "estudiante.\nNo␣se␣pudo␣escribir");
811:         } // end of try-catch
812:         try {
813:           // Saltar el numero de bytes requeridos para ubicarse
814:           // en el registro solicitado.
815:           aSaltar = numR*tamR + numBytes;
816:           archivoRndm.seek(aSaltar);
817:           // Escribo el registro construido en nuevo
818:           for (int i = 1; i <= tamanhos[0]; i++) {
819:               nombre = (nuevo.daCampo(i) + blancos).
820:                   substring(0,tamanhos[i]);
821:               archivoRndm.writeBytes(nombre);
822:           } // end of for (int i = 1; i <= tamanhos[0]; i++)
823:         } catch (IOException e) {
824:             System.out.println("Hubo␣error␣en␣GUARDARDIRECTO");
825:         } // end of try-catch
826:     } // end while se pidan registros
827:     try {
828:         archivoRndm.close();
829:     } catch (IOException e) {
830:         System.err.println("No␣pude␣cerrar␣archivo␣directo");
831:     } // end of try-catch
832:     return GUARDARDIRECTO;
```

Realmente ya no hay mucho que explicar en esta opción, pues prácticamente todo lo interesante ya se vio y los comentarios del código explican el orden. Sin embargo es necesario insistir en un punto. Supongamos que leemos un registro (en modo directo), lo modificamos y lo reescribimos; esto quiere decir que leemos y escribimos de la misma posición. Por lo tanto, tendremos que colocar el apuntador del archivo en la posición correspondiente; leemos el registro; lo modificamos; volvemos a colocar el apuntador de archivo en la posición anterior; y escribimos.

Si no "regresamos" al apuntador a la posición donde empieza el registro, vamos a sobreescribir en el siguiente registro, no en el que leímos. Esto se debe a que cada lectura (o escritura) avanza al apuntador tantos bytes como se hayan leído (o escrito); al final de la lectura el apuntador va a estar posicionado un byte más allá del final del registro leído, por lo que es necesario regresarlo a que apunte al principio del registro leído.

10.9 Lectura y escritura de objetos

Siendo Java un lenguaje orientado a objetos, uno se hace la pregunta de por qué se usan mecanismos "anticuados" – como la transformación de y a bytes – para lectura y escritura del ingrediente fundamental del lenguaje. Una respuesta, aunque no del todo precisa, es que el almacenamiento en dispositivos externos sigue siendo todavía en términos de bytes. Pero no es del todo precisa esta respuesta pues lo que intentamos con el uso de un lenguaje orientado a objetos es elevar el nivel de abstracción de nuestras aplicaciones; esto es, independientemente de cuál sea la representación subyacente de la información – el lenguaje de máquina que corresponde as la representación de la información – deseamos, al nivel de la aplicación, pensar en que tenemos almacenados objetos, independientemente de cómo estén codificados estos objetos.

La característica que buscamos escribiendo y leyendo objetos es el autoconocimiento que tienen de sí mismos los objetos: buscamos que tanto en el dispositivo externo como en la memoria durante la ejecución el objeto pueda describirse a sí mismo; que la lectura, escritura, y por lo tanto la interpretación, no dependa de la aplicación, sino que dependa de lo que está almacenado (o se esté almacenando). En resumen, deseamos que los objetos se almacenen con una descripción de sí mismos; y que esta descripción sea manejada automáticamente por el lenguaje, descargando de esta tarea a la aplicación.

Para ello cuenta Java con dos flujos, uno de entrada y otro de salida, que contempla la lectura y escritura de objetos, ObjectInputStream y ObjectOutputS-

tream respectivamente. Las lecturas y escrituras se llevan a cabo como acabamos de mencionar, por lo que los archivos presentan un tamaño mayor al que esperaríamos. También hay que tomar en cuenta que este modo de entrada/salida no es muy eficiente, sobre todo si los objetos son grandes o tenemos una cantidad grande de los mismos. En este último caso se recomienda "vaciar" el objeto como lo hicimos antes y dejar que sea la aplicación la que lo interprete. Se conoce como *serialización* a la conversión que se realiza de un objeto a un flujo de bytes y *deserialización* la acción inversa: construir el estado de un objeto a partir de su imagen en bytes.

Un aspecto muy importante e interesante es que al serializar un objeto estamos guardando su *estado*, que es lo que lo distingue de otros objetos de la misma clase. Al deserializarlo recuperamos precisamente el estado del objeto en el momento en que fue serializado. De esta manera se puede recuperar fácilmente el estado de una aplicación utilizando estos mecanismos. El contexto principal en el que estos mecanismos se requieren es cuando se ejecuta una aplicación distribuida en varias plataformas o de manera remota.

Regresando a los mecanismos de Java para leer/escribir objetos, llama la atención que ObjectInputStream y ObjectOutputStream no heredan de File***xxx***putStream sino que heredan directamente de ***xxx***putStream, donde ***xxx*** corresponde a "in" o "out". Pero veamos parte de la definición de estas clases y los métodos que nos aportan.

No es trivial guardar el estado de un objeto, pues algunos de los campos pueden depender de la máquina virtual en la que se está ejecutando o del contexto. Por ejemplo, si tenemos una lista ligada con objetos, la referencia al siguiente elemento de la lista depende de la secuencia de ejecución y de la localidad asignada por la máquina virtual: no es probable que en otra ejecución, posiblemente hasta en otra máquina o con distinto sistema operativo, la lista quede armada en las mismas posiciones que las dadas por el estado de los objetos. Otro ejemplo es el de un objeto que guarda la posición de un cierto flujo donde se encuentra el apuntador del archivo en el momento en que quedó definido su estado: en una ejecución posterior o distinta puede suceder que el archivo ya haya cambiado y esa posición no corresponda; o que al momento de leer el objeto el archivo ni siquiera esté disponible. Por lo tanto, hay campos en un objeto serializado que no tiene sentido guardar o leer porque se encuentran fuera de contexto. A este tipo de campos los marcamos con el calificativo de transient para indicarle a la aplicación que no los incluya en la serialización del objeto.

Otros campos que no vale la pena tampoco guardar son los campos estáticos (static) de la clase. También en este caso, dado que esos campos pueden ser modificados por cualquiera de los objetos de esa clase, el valor que tienen cuando se serializa al objeto no tiene por que ser el mismo que cuando se le deserializa. Por

lo tanto, tampoco los campos estáticos van a ser incluidos en la serialización de un objeto.

Si tenemos una clase que contiene campos que corresponden a otra(s) clase(s), los objetos de la primera van a poder ser serializados si y sólo si cada una de los campos que corresponden a clases contenidos en el objeto son también serializables. Asimismo, si una superclase no es serializable, ninguna subclase de ella puede ser serializable, ya que si serializamos a la subclase le damos la vuelta a la seguridad de la superclase que no desea ser serializada.

Cada objeto en el flujo consiste de un bloque de datos, en el que primero se encuentra la descripción de los datos y después los datos propiamente dichos.

Como se puede ver de estos breves comentarios, la serialización y deserialización de objetos tiene muchísimos puntos finos que hay que observar. Haremos una lista más precisa de estos puntos hacia el final de esta sección. Pasemos a revisar los flujos de objetos.

---

public class ObjectInputStream extends InputStream
implements ObjectInput, ObjectStreamConstants

Un flujo de entrada de objetos que deserializa datos primitivas que fueron previamente escritos usando un flujo de salida de objetos (ObjectOutputStream). Sólo los objetos que implementan la interfaz Serializable son sujetos a ser serializados.

La interfaz ObjectInput hereda de la interfaz DataInput todos los métodos que interpretan valores primitivas y agrega métodos para leer objetos y bytes "en crudo", sin interpretación.

La interfaz ObjectStreamConstants simplemente proporciona constantes simbólicas para evitar el uso de números mágicos. Además de implementar los métodos de DataInput (que no listaremos) provee métodos específicos para trabajar con objetos.

Clase anidada:

---

public abstract static class ObjectInputStream.GetField

Provee acceso para los campos persistentes leídos del flujo de entrada.

Campos:

Los heredados de ObjectStreamConstants.

Constructores:

---

protected ObjectInputStream() **throws** IOException,
SecurityException

La excepción de I/O es la usual. La excepción relativa a la seguridad tiene que ver con el administrador de seguridad, si es que se violan los permisos dados para la serialización.

---

public ObjectInputStream(InputStream in) **throws** IOException

Crea un ObjectInputStream que se asocia al flujo dado por in.

Métodos: (además de los de DataInput)

---

public void defaultReadObject() **throws** IOException, ClassNotFoundException

Lee los datos no-estáticos y no-transeúntes de la clase asociada a este flujo. Sólo puede ser invocada desde el método readObject. Si el flujo no está activo, lanza la excepción NotActiveException. Si no encuentra en su entorno la clase a la que pertenecen los objetos serializados, lanza la excepción ClassNotFoundException.

---

protected boolean enableResolveObject(**boolean** enable)

Habilita al flujo para permitir que los objetos que son leídos del flujo puedan ser reemplazados. Regresa el valor anterior antes de esta invocación.

Lanza la excepción si el manejador de seguridad no permite que se reescriba en este flujo.

---

public Object readObject()
throws IOException, ClassNotFoundException

Permite leer un objeto de un flujo serializado.

Además de las excepciones mencionadas en el encabezado, puede lanzar una de las siguientes excepciones:

InvalidClassException Lanza esta excepción cuando la clase descrita en la serialización no coincide con la que se encuentra declarada, no reconoce el tipo de alguno de los campos o bien no cuenta con el constructor requerido.

StreamCorruptedException La descripción del objeto viola la consistencia del sistema y no lo puede leer.

OptionalDataException En el flujo no hay objetos, sino datos primitivos; no está la descripción del objeto.

---

protected ObjectStreamClass readClassDescriptor ()

throws IOException, ClassNotFoundException

Lee un descriptor de clase del flujo serializado. Se utiliza cuando la aplicación espera un descriptor de clase como siguiente elemento en el flujo.

protected Object resolveObject(Object obj)

throws IOException

Permite a subclases en las que se confía sustituir a un objeto por otro durante la deserialización. Este método se llama después de invocar a readObject pero antes de regresar de esta última invocación.

La lista que acabamos de dar dista mucho de ser exhaustiva, pero contiene la suficiente información para poder cumplir con nuestro objetivo, que consiste en serializar y deserializar la base de datos. Terminemos de reunir las herramientas que requerimos mostrando la definición de la clase ObjectOutputStream.

public class ObjectOutputStream extends OutputStream implements ObjectOutput, ObjectStreamConstants

Escribe al objeto serializado en un flujo de bytes. Únicamente los objetos de clases que declaran ser Serializable pueden ser serializadas. Tanto la superclase como todos los campos de la clase que a su vez sean objetos deben ser serializables.

Clase anidada:

public abstract static class ObjectOutputStream.PutField

Aporta acceso de programación a los campos persistentes a escribirse e la salida de objetos (ObjectOutput).

Campos:

Los heredados de ObjectStreamConstants.

Constructores:

public ObjectOutputStream(OutputStream out)

throws IOException

Crea un flujo ObjectOutputStream que escribe sobre el flujo de salida especificado (out). Escribe el encabezado de la serialización en el flujo de salida. Además de la excepción IOException puede lanzar una excepción de seguridad (SecurityException) si hubiese clases que pasen por encima de las medidas de seguridad de acceso; y finalmente también puede lanzar una excepción NullPointerException si out es nulo.

protected ObjectOutputStream() **throws** IOException, SecurityException

Sirve para construir subclases que redefinan el flujo para no tener que asignar espacio para datos privados que se usan únicamente en esta implementación de ObjectOutputStream. Realiza las verificaciones de seguridad necesarias.

protected void annotateClass(Class<?> cl) **throws** IOException

Permite escribir en el flujo datos que corresponden a la clase. Corresponde al método resolveClass de ObjectInputStream. Lo escrito por este método debe ser leído por resolveClass.

public void defaultWriteObject () **throws** IOException

Escribe los campos no estáticos y no transeúntes al flujo. Sólo puede ser invocado desde writeObject. So no se hace así lanza una excepción del tipo NotActiveException.

protected void drain() **throws** IOException

Vacía cualquier información acumulada en el flujo que no ha sido escrita en el dispositivo. Similar a flush pero no se encadena con el dispositivo.

protected boolean enableReplaceObject (**boolean** enable) **throws** SecurityException

Cuando es verdadera, permite el reemplazo de objetos en el flujo. Si está habilitada, se invoca al métodoreplaceObject cada vez que se serializa un objeto.

public ObjectOutputStream.PutField putFields () **throws** IOException

Obtiene el objeto que se usa para almacenar campos persistentes que van a ser escritos en el flujo. Los campos se escriben cuando se invoca al método writeFields.

protected Object replaceObject(Object obj) **throws** IOException

Permite que clases confiables sustituyan un objeto por otro durante la serialización. El objeto que reemplaza (obj) puede no ser serializable. El objeto debe ser de la clase que está descrita en el flujo, o de una subclase de ésta. De otra manera lanza una excepción.

public void reset () **throws** IOException

Ignora el estado de los objetos que ya fueron escritos en el flujo. El estado del flujo se establece igual al de haberlo creado (new ObjectInputStream).

public void write(***args***) **throws** IOException

La especificación de ***args*** puede ser cualquiera de las especificadas en las interfaces DataOutput. Los argumentos son, como en otros flujos, un arreglo de bytes (con, opcionalmente, la primera posición y el tamaño) o un valor entero.

Además de aquellos métodos que escriben enteros, bytes, flotantes, etc., tenemos métodos que provienen de la interfaz ObjectOutput. Únicamente mencionaremos los que provienen de la segunda interfaz, ya que el resto funciona exactamente de la misma manera que en todos los flujos que implementa DataOutput y que ya revisamos.

public void writeFields () **throws** IOException

Escribe los campos que se encuentren en el buffer al flujo de objetos.

protected void writeStreamHeader() **throws** IOException

Se proporciona para que las subclases puedan pegar su propio encabezado al flujo.

protected void writeClassDescriptor(ObjectStreamClass desc) **throws** IOException

Se utiliza para que las subclases puedan determinar la manera como van a codificar la descripción de la clase.

public final void writeObject(Object obj) **throws** IOException

Escribe el objeto indicado en el flujo, incluyendo a la clase y la firma de la clase a la que pertenece el objeto, los valores de las variables de estado que no sean transeúntes y todos los supertipos de la clase. Se usa para brincar la serialización y hacerla más *ad-hoc* para el objeto dado. Si se reimplementa este método se tiene que hacer lo mismo con readObject de ObjectInputStream. Las excepciones que puede lanzar son:

InvalidClassException Algo está mal con la clase utilizada por la serialización.

NotSerializableException Alguno de los objetos que forman parte del objeto que se desea escribir no implementa la interfaz Serializable.

IOException Excepciones lanzadas por el flujo subyacente.

protected void writeObject(Object obj) **throws** IOException

Lo usan las subclases para redefinir el método writeObject.

public void writeUnshared(Object obj) **throws** IOException

Escribe el objeto como único, sin referencias hacia objetos anteriores en el mismo flujo.

Como se puede ver de la descripción de los métodos (y nos faltaron algunos) la escritura y lectura en flujos de objetos es muy sofisticada, ya que permite transferir objetos de una aplicación a otra, conservando cada uno su estado, de manera transparente y muy efectiva. Cuando se están ejecutando aplicaciones donde los datos (en este caso objetos) tiene que transitar por la red, tal vez el tiempo adicional de escribir/leer en un flujo de objetos no es muy significativo. Sin embargo, hay que tener presente que la lectura/escritura de objetos es un proceso lento y que no debiera utilizarse en aplicaciones locales que manejen un número grande de datos.

Regresemos a nuestro objetivo de esta sección que es la de escribir y leer a flujos con formato de objetos. Nuevamente permitiremos en cualquier momento de las ejecución de nuestra aplicación cargar o descargar objetos a y desde la base de datos. Pero lo primero que tenemos que hacer es permitir que cada uno de los objetos que pretendemos escribir implementen a la interfaz Serializable. Lo más sencillo sería simplemente agregar este aspecto a nuestra clase Estudiante, y como ésta extiende a InfoEstudiante, ambas tenemos que corregirlas para que implementen a la interfaz[8]. Como InfoEstudiante originalmente implementa interfaces, a las que no hay que ponerles el calificativo de ser serializables, con estas dos clases ya cubrimos el requisito de que todas las superclases sean serializables. También hay que notar que el registro consiste de cadenas y 7un referencia; como las cadenas son serializables y la referencia también lo es, se cubre ampliamente el requisito de ser serializable. Los cambios hechos a InfoEstudiante y Estudiante se pueden ver en los listados 10.13 y 10.14 en la página opuesta respectivamente.

Código 10.13 Cambios a **InfoEstudiante** (**InfoEstudianteSerial**) 1/2

```
 1: import java.io.*;
 2: /**
 3:  * Base de datos, a base de listas de registros, que emula
 4:  * la lista de un curso de licenciatura. Tiene las opciones
 5:  *  normales de una base de datos y funciona mediante un Menú
 6:  */
 7: class InfoEstudianteSerial implements DaEstudiante, Serializable  {
.....
16:     /**
17:      * Constructor sin parámetros
18:      */
```

[8]Reescribimos estas dos clases agregando al nombre Serial para mantener intacto el trabajo que realizamos con anterioridad.

Código 10.13 Cambios a **InfoEstudiante** (**InfoEstudianteSerial**) 2/2

```
19:     public InfoEstudianteSerial() {
20:         nombre = carrera = cuenta = null;
21:     }
22:     /** Constructor a partir de datos de un estudiante.
23:      * los campos vienen separados entre sí por comas,
24:      * mientras que los registros vienen separados entre sí
25:      * por punto y coma.
26:      * @param String, String, String, String los valores para
27:      *  cada uno de los campos que se van a llenar.
28:      * @return InfoEstudiante una referencia a una lista
29:      */
30:     public InfoEstudianteSerial(String nmbre, String cnta,
31:                                 String crrera)             {
32:         nombre = nmbre.trim();
33:         cuenta = cnta.trim();
34:         carrera = crrera.trim();
35:     }
  ......
```

Como ya habíamos mencionado, en la línea 7: le cambiamos el nombre a la clase y agregamos el que implementa la interfaz Serializable. Por este cambio de nombre, también tuvimos que cambiar el nombre a los constructores – líneas 19: y 30: –. De similar manera tenemos que modificar a la clase Estudiante.

Código 10.14 Modificaciones a la clase Estudiante **(EstudianteSerial) 1/2**

```
1: import java.io.*;
2:
3: //  import icc1.interfaz.Consola;
4: /**
5:  * Base de datos, a base de listas de registros, que emula la lista de un
6:  * curso de licenciatura. Tiene las opciones normales de una base de
7:  * datos y funciona mediante un Menú
8:  */
9: class EstudianteSerial extends InfoEstudianteSerial
10:                             implements Serializable   {
  ......
13:     /** Constructor sin parámetros */
14:     public EstudianteSerial()  {
15:         super();
16:         clave = null;
17:         siguiente = null;
18:         short[] tam = {(short)(tamanhos[0]+1),
19:                        tamanhos[1],tamanhos[2],tamanhos[3],20};
```

Código 10.14 Modificaciones a la clase **Estudiante** (**EstudianteSerial**) 2/2

```
20:         tamanhos = tam;
21:     }
22:     /** Constructor a partir de datos de un estudiante.
23:      * los campos vienen separados entre sí por comas,
24:      * mientras que los registros vienen separados entre sí
25:      * por punto y coma.
26:      * @param String, String, String, String los valores para
27:      *             cada uno de los campos que se van a llenar.
28:      * @return Estudiante una referencia a una lista
29:      */
30:     public EstudianteSerial(String nmbre, String cnta,
31:                             String crrera, String clve)    {
50:     public EstudianteSerial daSiguiente()  {
51:         return siguiente;
52:     }
53:     /** Actualiza el campo siguiente.
54:      * @params Estudiante la referencia que se va a asignar.
55:      */
56:     public void ponSiguiente(EstudianteSerial sig)  {
57:         siguiente = sig;
58:     }
.....
```

En las líneas 9:, 13:, 30:, 50: y 56: se encuentran las consecuencias de haber cambiado el nombre a ambas clases de las que hablamos. En la línea 10: se encuentra la declaración de que esta clase implementa a la interfaz Serializable, lo que la hace susceptible de ser escrita o leída de un flujo de objetos.

Dado que la base de datos está compuesta por objetos de la clase Estudiante y eso no lo queremos modificar, tenemos que dar métodos que conviertan de EstudianteSerial a Estudiante y viceversa en la clase EstudianteSerial. Son métodos sencillos que únicamente copian los campos. El código se puede ver en el listado 10.15.

Código 10.15 Conversión de **Estudiante** a **EstudianteSerial** y viceversa (**EstudianteSerial**) 1/2

```
99:     /**
100:     * Convierte a un estudiante en un estudiante serial, simplemente
101:     * copiando los campos correspondientes.
102:     * @param   nuevo el Estudiante.
103:     * @returns Un EstudianteSerial con el mismo contenido que nuevo.
104:     */
```

Código 10.15 Conversión de **Estudiante** a **EstudianteSerial** y viceversa (**EstudianteSerial**) 2/2

```
105:     public void daEstudianteSerial(Estudiante nuevo) {
106:         nombre = nuevo.daNombre();
107:         cuenta = nuevo.daCuenta();
108:         carrera = nuevo.daCarrera();
109:         clave = nuevo.daClave();
110:     }
111:     /**
112:      * Convierte a un EstudianteSerial en un Estudiante construyendo
113:      * uno de 'estos.
114:      * @param el estudiante serial.
115:      * @returns un Estudiante.
116:      */
117:     public Estudiante daEstudiante() {
118:         String nmbre = this.daNombre();
119:         String cnta = this.daCuenta();
120:         String crrera = this.daCarrera();
121:         String clve = this.daClave();
122:         return new Estudiante(nmbre, cnta, crrera, clve);
123:     }
```

Habiendo hecho estos cambios, nos avocamos a la clase ListaCursoIO – a la que identificaremos como ListaCursoIOObj – para ver los cambios necesarios en el manejo de nuestra base de datos, que se localizan en la clase MenuListaIOObj.

Lo primero que tenemos que hacer es agregarle al método que pide el nombre del archivo la opción para que pida un archivo del que se van a leer (o escribir) objetos, lo que se encuentra en el listado 10.16.

Código 10.16 Solicitud de nombre de archivo para flujo de objetos (**MenuListaIOObj**)

```
182:     public String pideNombreArch(BufferedReader cons, int caso)
183:         throws IOException {
184:         String mensaje = "Por␣favor␣dame␣el␣nombre␣del␣archivo\n";
185:         switch (caso) {

              ......

219:         case LEEROBJETOS:
220:             mensaje += "de␣dónde␣vas␣a␣leer␣objetos";
221:             break;
222:         case GUARDAROBJETOS:
223:             mensaje += "en␣dónde␣vas␣a␣escribir␣objetos";
224:             break;

              ......
```

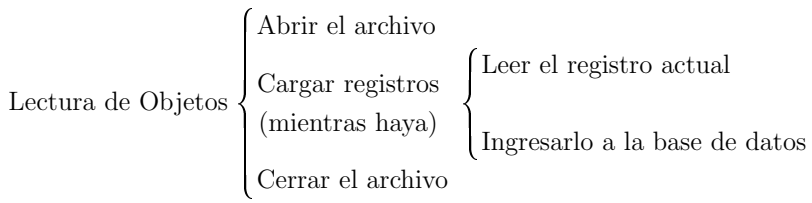
Necesitamos también declarar un flujo de objetos, al principio del método daMenu, lo que se encuentra en el listado 10.17.

Código 10.17 Declaración de flujo de objetos (**MenuListaIOObj**)

```
267:        ObjectInputStream archvoObjetosIn = null;
268:        ObjectOutputStream archvoObjetosOut = null;
```

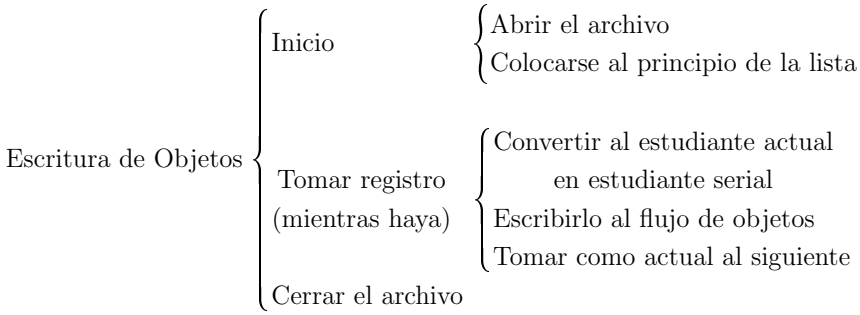
Con esto ya estamos listos para llenar los casos que nos ocupan. El algoritmo para la lectura de los registros se muestra en la figura 10.17 y como se puede observar es prácticamente idéntico al de leer registros de un archivo directo, excepto que por el hecho de leer objetos la máquina virtual se encarga de interpretarlos.

Figura 10.17 Algoritmo para la lectura de objetos



En este diagrama no pusimos el manejo de excepciones porque no forman parte de la lógica general del algoritmo. En la figura 10.18 mostramos el diagrama para la escritura a flujos de objetos, que es sumamente similar al de la lectura.

Figura 10.18 Escritura a un flujo de objetos



Antes de proceder con la codificación de estos esquemas debemos nuevamente declarar los flujos correspondientes y una variable para almacenamiento temporal de objetos de la clase EstudianteSerial, así como modificar el menú y el manejo de las opciones. El código correspondiente a esto se encuentra en el listado 10.18.

Código 10.18 Caso de lectura de objetos (declaraciones) (**MenuListaIOObj**)

```
          ......
267:      ObjectInputStream archvoObjetosIn = null;
268:      ObjectOutputStream archvoObjetosOut = null;
          ......
272:      EstudianteSerial nuevoSrl = null;
          ......
293:          + "(F)\tLeer objetos de flujo\n"
294:          + "(G)\tEscribir objetos en flujo\n"
          ......
308:      opcion = "0123456789ABCDEFGZ".indexOf(sopcion);
```

En el listado 10.19 se encuentra el código que corresponde al manejo de estos dos casos en el menú.

Código 10.19 Caso de lectura de objetos (**MenuListaIOObj**) 1/3

```
          ......
966:  case LEEROBJETOS:
967:    try {
968:      archvoObjetosIn = null;
969:      sArchivo = pideNombreArch(cons, LEEROBJETOS);
970:      archvoObjetosIn = new ObjectInputStream
971:                        (new FileInputStream(sArchivo));
972:      if (archvoObjetosIn == null) {
973:        System.out.println("el archivo NO quedó abierto");
974:      } // end of if (archvoObjetosIn == null)
975:      boolean yaNoHay = false;
976:      while (!yaNoHay) {
977:        nuevoSrl = null;
978:        /* Cuando se acaba un archivo de objetos y se
979:         * trata de leer simplemente lanza una excepción
980:         */
981:        try {
982:          nuevoSrl =
983:              (EstudianteSerial)archvoObjetosIn.readObject();
```

Código 10.19 Caso de lectura de objetos (**MenuListaIOObj**) 2/3

```
984:         } catch (IOException e) {
985:           yaNoHay = true;
986:         } // end of try-catch
987:         if (yaNoHay) {
988:           continue;
989:         } // end of if (archvoObjetosIn.available > 0)
990:         nuevo = nuevoSrl.daEstudiante();
991:         miCurso.agregaEstOrden(nuevo);
992:       } // end of while ((nombre = archivoIn.readLine()...
993:     } // end of try
994:     catch (ClassNotFoundException e) {
995:       System.out.println("Se está leyendo el archivo equivocado");
996:     } // end of catch
997:     catch (InvalidClassException e) {
998:       System.out.println("La clase no permite ser serializada");
999:     } // end of catch
1000:    catch (StreamCorruptedException e) {
1001:      System.out.println("La descripción del objeto "
1002:                         + "es inconsistente");
1003:    } // end of catch
1004:    catch (OptionalDataException e) {
1005:      System.out.println("No se encontraron objetos sino "
1006:                         + "datos directos");
1007:    } // end of catch
1008:    catch (IOException e) {
1009:      System.err.println("No pude abrir archivo");
1010:    } // end of try-catch
1011:    catch (Exception e) {
1012:      System.out.println("No alcanzaron los datos");
1013:    } // end of catch
1014:    finally {
1015:      if (archvoObjetosIn != null) {
1016:        try {
1017:          archvoObjetosIn.close();
1018:        } catch (IOException e) {
1019:          System.err.println("No pude cerrar el"
1020:                             + " archivo de lectura");
1021:        } // end of try-catch
1022:      } // end of if (archivoIn != null)
1023:    } // end of finally
1024:    return LEEROBJETOS;
1025:
1026:  case GUARDAROBJETOS:
1027:    try {
1028:      sArchivo = pideNombreArch(cons,GUARDAROBJETOS);
1029:      archvoObjetosOut = null;
```

Código 10.20 Caso de lectura de objetos (**MenuListaIOObj**) 3/3

```
1030:       archvoObjetosOut = new ObjectOutputStream
1031:             (new FileOutputStream(sArchivo));
1032:       System.out.println("Abrí el archivo");
1033:       if (archvoObjetosOut == null) {
1034:             System.out.println("el archivo NO está abierto!");
1035:       } // end of if (archvoObjetosOut == null)
1036:       lista = ((Estudiante)miCurso.daLista());
1037:       int contador = 0;
1038:       while (lista != null) {
1039:         nuevoSrl = new EstudianteSerial();
1040:         nuevoSrl.daEstudianteSerial(lista);
1041:         archvoObjetosOut.writeObject(nuevoSrl);
1042:         lista = lista.daSiguiente();
1043:       } // end of while (lista != null)
1044:     }catch (SecurityException e) {
1045:        System.out.println("Se violaron condiciones de seguridad");
1046:     } // end of catch
1047:     catch (IOException e) {
1048:       System.err.println("No pude abrir archivo");
1049:     } // end of try-catch
1050:     finally {
1051:       try {
1052:          if (archvoObjetosOut != null) {
1053:             archvoObjetosOut.close();
1054:          } // end of if (archivoOut != null)
1055:       } catch ( IOException e) {
1056:          System.err.println("No pude cerrar el archivo");
1057:       } // end of try-catch
1058:     } // end of finally
1059:     return GUARDAROBJETOS;
```

Como hasta ahora, la lectura es un poco más compleja que la escritura, ya que sabemos lo que estamos escribiendo, pero al tratar de leer pueden venir mal los datos, no existir el archivo, leer de más, etc.. Pasamos a explicar únicamente aquellas líneas de código que no sean del todo claras.

Para abrir el archivo de entrada o de salida se construye un flujo de objetos *sobre* un flujo de bytes – líneas 970:–971: y 1030: – 1031: – ya que esa es la definición del constructor de un flujo de objetos.

Se lee el registro a un EstudianteSerial – línea 982: – 983: –, pero como la lista que tenemos es de objetos de la clase Estudiante, lo tendremos que convertir antes de agregarlo a la lista. En el caso de flujos de objetos no tenemos un indicador

adecuado de cuándo se acabó el archivo y lo único que podemos usar es la excepción de entrada y salida que se va a alcanzar cuando ya no pueda leer del flujo. Por ello, colocamos estas líneas en un bloque try–catch – 981: a 986: que atrape la excepción y simplemente prenda una variable para avisar que ya no hay datos. En las líneas 987: a 989: se detecta que ya se alcanzó el fin de archivo (o que no se pudo leer) y se regresa al encabezado del while, para evitar tratar de procesar datos erróneos.

Finalmente, en las líneas 990: y 991: se obtiene un objeto Estudiante a partir del que se leyó y se agrega a la lista.

Las cláusulas catch externas a la lectura y que corresponden únicamente al caso de lectura – líneas 994: a 1013: simplemente proporcionan un mensaje de error de qué es lo que sucedió, lo mismo que las cláusulas catch del caso de escritura a flujo de objetos – líneas 1044: a 1049: –.

En ambos casos que nos ocupan tenemos una cláusula finally que se encarga de cerrar el archivo, preguntando antes si es que el archivo fue abierto adecuadamente – líneas 1015: a 1022: en lectura y 1050: a 1058: en escritura – que además tiene que estar, nuevamente, en un bloque try-catch para el caso en que se lance una excepción.

Es interesante ver con un visor de texto en ASCII un archivo creado como flujo de objetos ya que se ve, de alguna manera, la gran cantidad de información que se encuentra adicional a los datos mismos. Esto se debe a que con cada objeto se tiene que describir al mismo, lo que sucede ya que en un mismo flujo de objetos podemos escribir objetos de distintas clases y al leerlos, automáticamente se carga la descripción que trae consigo el objeto. Es por eso que la lectura siempre es de un Object y se tiene que hacer un cast para obtener el objeto que creemos estamos leyendo. Para "decodificar" un objeto de un flujo del que no sabemos a qué clase pertenece, podemos utilizar la clase Class que proporciona toda clase de herramientas para ello.

10.10 Colofón

Revisamos varias clases de flujos de entrada y salida, pero de ninguna manera fuimos exhaustivos. Sin embargo, los conceptos vertidos en este capítulo deber servir de plataforma para poder revisar (y utilizar adecuadamente) flujos de otras clases que no hayan sido tratados. Una revisión completa de toda la entrada y salida que ofrece Java está más allá del alcance de este material.

Hilos de ejecución 11

11.1 ¿Qué es un hilo de ejecución?

Hasta ahora, cada vez que ejecutamos un programa estamos trabajando de solamente sobre el programa. Sin embargo, cualquier sistema operativo moderno tiene más de un programa trabajando al mismo tiempo: el servidor de impresión, el shell, los demonios para correo electrónico, etc. Cada uno de estos programas constituyen un *hilo de ejecución* – *thread*, proceso, tarea – que trabaja independiente de otros procesos y que usan un juego común de datos.

Un ejemplo más sencillo y más usual es el manejo de las cuentas de cheques de un banco. El banco cuenta con su base de datos y tenemos a miles de cajeros teniendo acceso a la base de datos, cada cajero en su propio hilo de ejecución. No sería pensable que fuera un único proceso el que se encargara de todos los accesos. Decimos entonces que tenemos *procesos o hilos de ejecución concurrentes*, ya que concurren a un mismo conjunto de datos o recursos.

11.2 La clase Thread

En Java tenemos la posibilidad de que un proceso lance, a su vez, a otros procesos. En principio, los procesos lanzados pueden o no mantener comunicación con el proceso padre que los lanzó. Asimismo, los procesos lanzados, a diferencia de los métodos invocados, no regresan al lugar desde el que se les lanzó. Pero tienen acceso a todos los recursos y atributos con que cuenta el método que los haya lanzado.

La clase Thread es la encargada de iniciar hilos de ejecución independientes. Es una clase muy extensa que tiene muchísimos métodos y atributos. En este momento nos interesan dos métodos: el que define al hilo de ejecución, run(), y el que lo inicia, start().

Si deseamos que de una clase se puedan crear hilos de ejecución independientes hacemos que extienda a Thread y que tenga un método run(). Veamos el ejemplo de una clase que lanza dos hilos de ejecución independientes en el listado 11.1.

Código 11.1 Objeto que lanza dos hilos de ejecución 1/2

```
1:  public class PingPong extends Thread  {
2:      private String word;
3:      private int delay;
4:
5:      public PingPong(String whatToSay, int delayTime)    {
6:          word = whatToSay;
7:          delay = delayTime;
8:      }
9:
10:     public void run()       {
11:         System.err.println("Empezó:"+Thread.currentThread().getName());
12:         try    {
13:             while (true) {
14:                 System.out.print(word + "␣");
15:                 Thread.sleep(delay);
16:             }
17:         } catch  (InterruptedException e) {
18:             return;   // Termina este hilo de ejecución
19:         }
20:     }
```

Código 11.1 Objeto que lanza dos hilos de ejecución 2/2

```
21:     public static void main(String[] args)    {
22:         System.out.println("Nombre:␣" + currentThread().getName());
23:         new PingPong("ping", 33).start();  // 1/30 de segundo
24:         new PingPong("pong", 100).start();    // 1/10 de segundo
25:     }
26: }
```

En el listado 11.1 en la página opuesta la clase PingPong extiende a la clase Thread, por lo que tendrá que redefinir su propio método run() – el de Thread no hace nada – lo que hace en las líneas 10: a 20:. Esto es lo que va a hacer el hilo de ejecución, cuando sea lanzado. Se inicia un nuevo hilo de ejecución cuando se invoca al método start() de un objeto, cuya clase extiende a Thread – líneas 23:2 y 24: del listado 11.1 en la página opuesta.

En la línea 11: aparecen dos invocaciones a dos métodos de Thread. El método currentThread() es un método estático de la clase Thread y regresa un objeto de tipo Thread – podía haber sido llamada sin precederlo del nombre de la clase. El método getName() regresa una cadena, que es el nombre del hilo de ejecución que se lanzó. El nombre del hilo de ejecución es un atributo del mismo. También en la línea 15: se encuentra un método estático de la clase Thread, el método sleep(delay). Este método suspende la ejecución del hilo durante el tiempo estipulado.

Figura 11.1 Salida de PingPong.

```
java PingPong
Nombre: main
Empezó:Thread-0
ping Empezó:Thread-1
pong ping ping pong ping ping pong ping ping ping pong ping ping
pong ping ping pong ping ping pong ping ping pong ping ping ping
pong ping ping pong ping ping pong ping ping pong ping ping pong
ping ping ping pong ping ping pong ping ping pong ping ping pong
ping ping pong ping ping ping pong ping ping pong ping ping pong
ping ping pong ping ping pong ping ping ping pong ping ping pong
ping ping pong ping ping pong ping ping pong ping ping ping pong
ping ping pong ping ping pong ping ping pong ping ping pong ping
ping ping pong ping ping pong ping
```

La clase PingPong podría no tener método main. Lo ponemos para hacer una demostración de esta clase. En la línea 22: nuevamente invocamos a los métodos

currentThread y getName para que nos indiquen el nombre del hilo donde aparece esa solicitud. A continuación creamos dos objetos anónimos y les solicitamos que cada uno de ellos inicie su hilo de ejecución con el método start().

La ejecución del método main de esta clase produce la salida que se observa en la figura 11.1 en la página anterior.

Como se puede observar en la figura 11.1 en la página anterior, mientras que el nombre del proceso que está ejecutándose inmediatamente antes de lanzar los hilos de ejecución se llama main, que es el que tiene, los hilos de ejecución lanzados tienen nombres asignados por el sistema. Podíamos haberles asignado nosotros un nombre al construir a los objetos. Para eso debemos modificar al constructor de la clase PingPong e invocar a un constructor de Thread que acepte como argumento a una cadena, para que ésta sea el nombre del hilo de ejecución. Los cambios a realizar se pueden observar en el listado 11.2. La salida que produce se puede ver en la figura 11.2 en la página opuesta.

Código 11.2 Asignación de nombres a los hilos de ejecución

```
1:  public class NombrePingPong extends Thread    {
2:      private String word;
3:      private int delay;
4:
5:      public NombrePingPong(String whatToSay, int delayTime,
6:                            String name)    {
7:          super(name);
8:          word = whatToSay;
9:          delay = delayTime;
10:     }

        ...// El método run() queda exactamente igual

20:     public static void main(String[] args)      {
21:         System.out.println("Nombre:␣"+ currentThread().getName());
22:             /* 1/30 de segundo:  */
23:         new NombrePingPong("ping", 33,"Ping").start();
24:             /* 1/10 de segundo:  */
25:         new NombrePingPong("pong", 100,"Pong").start();
26:     }
27: }
```

Ejecución del método run()

Una vez iniciado el hilo de ejecución, el método run() va a proseguir su ejecución hasta que una de tres cosas sucedan:

1. El método se encuentra un return, en cuyo caso simplemente termina su

ejecución.

II. El método llega al final de su definición.

III. Se presenta una excepción en el método.

Debe quedar claro que en el caso de hilos de ejecución, cuando uno termina su ejecución no es que "regrese" al lugar desde el que se le lanzó, sino que simplemente se acaba. El hilo de ejecución, una vez lanzado, adquiere una vida independiente, con acceso a los recursos del objeto desde el cual fue lanzado.

Figura 11.2 Salida con asignación de nombres.

```
java NombrePingPong
Nombre: main
Empezó:Ping
ping Empezó:Pong
pong ping ping pong ping ping pong ping ping ping pong
ping ping pong ping ping pong ping ping pong ping ping
pong ping ping ping pong ping ping pong ping ping pong
ping ping pong ping ping pong ping ping ping pong ping
ping pong ping ping pong ping ping pong ping ping pong
ping ping ping pong ping ping pong ping ping pong ping
ping pong ping ping pong ping ping ping pong ping ping
pong ping ping pong ping ping pong ping ping pong ping
ping ping pong ping ping pong ping ping pong ping ping
pong ping ping pong ping
```

11.3 La interfaz Runnable

Java proporciona otra manera de lanzar hilos de ejecución, que es mediante clases que implementan a la interfaz Runnable. Esta interfaz sólo declara un método, run(), por lo que una clase que implemente a esta interfaz, sus objetos también pueden lanzar hilos de ejecución independientes. Veamos la clase del PingPong programada implementando a la interfaz Runnable en el listado 11.3 en la siguiente página.

Código 11.3 Hilos de ejecución con la interfaz **Runnable**

```
1: class RunPingPong implements Runnable  {
2:     private String word;
3:     private int delay;
4:
5:     RunPingPong(String whatToSay, int delayTime)    {
6:         word = whatToSay;
7:         delay = delayTime;
8:     }
9:
10:    public void run()    {
11:        System.out.println("El nombre del hilo de ejecución es: " +
12:                           Thread.currentThread().getName());
13:        try   {
14:            for( ; ; )  {
15:                System.out.print(word + " ");
16:                Thread.sleep(delay);
17:            }
18:        }  catch (InterruptedException e)  {
19:            return;
20:        }
21:    }
22:
23:    public static void main(String[] args)    {
24:        Runnable ping = new RunPingPong("ping", 33);
25:        Runnable pong = new RunPingPong("pong", 100);
26:        new Thread(ping, "Ping").start();
27:        new Thread(pong, "Pong").start();
28:    }
29: }
```

La clase RunPingPong se declara implementando Runnable. Al igual que cuando se extiende a la clase Thread, RunPingPong debe definir su propio método run(), que es lo que va a hacer el hilo de ejecución una vez lanzado. Básicamente hace lo mismo que hacía la clase PingPong, que era una extensión de Thread. Donde cambia un poco el asunto es al crear los hilos de ejecución. Si vemos las líneas que corresponden a main, primero se crean objetos de tipo Runnable – se pueden crear objetos de este tipo, ya que Runnable está implementada en RunPingPong – y se asignan en esas referencias a objetos tipo RunPingPong. Una vez hecho esto, se construyen hilos de ejecución – líneas 24: y 25: – para cada uno de los objetos y se lanzan mediante el método start. En esta ocasión, al construir el hilo de ejecución le estamos asignando un nombre, mediante el segundo argumento al constructor de la clase Thread.

Una ejecución de este programa produce lo que se ve en la figura 11.3. Es importante aclarar que dos ejecuciones de RunPingPong pueden o no dar la misma salida. La máquina virtual de Java (JVM) no puede garantizar el orden en que llegan los procesos a la pantalla. Nótese también que no se mezclan las palabras "ping" y "pong" en la misma. Esto es porque una vez que System.out empieza a escribir desde uno de los hilos de ejecución, no suelta la pantalla hasta que termina.

Figura 11.3 Salida de RunPingPong.

```
java RunPingPong
El nombre del hilo de ejecución es: Ping
ping El nombre del hilo de ejecución es: Pong
pong ping ping pong ping ping pong ping ping pong ping ping ping
pong ping ping pong ping ping pong ping ping pong ping ping pong
ping ping ping pong ping ping pong ping ping pong ping ping pong
ping ping pong ping ping ping pong ping ping pong ping ping pong
ping ping pong ping ping pong ping ping ping pong ping ping pong
ping ping pong ping ping ping pong ping ping pong ping ping ping pong
ping ping pong ping ping ping pong ping ping pong ping ping pong ping
ping ping pong ping ping ping pong ping ping pong ping ping pong ping
ping pong ping ping ping pong ping ping pong ping ping pong ping
ping pong ping ping pong ping ping ping pong ping ping pong ping
ping pong ping ping pong ping
```

11.3.1. Constructores de la clase Thread

Hasta ahora hemos visto cuatro constructores para objetos de la clase Thread:

public Thread()
: Es el constructor por omisión. Construye un hilo de ejecución anónimo, al que el sistema le asignará nombre. Cuando una clase que extiende a Thread no invoca a ningún constructor de la superclase, éste es el constructor que se ejecuta.

public Thread(String name)
: Éste construye un hilo de ejecución y la asigna el nombre dado como argumento.

public Thread(Runnable target)
Usa el método run() de la clase que implementa a Runnable para ejecutarse.

public Thread(Runnable target, String name)
Construye un Thread con el nombre especificado y usa el método run() de la clase que implementa a Runnable.

El problema con implementar a través de Runnable es que el método run() es público y entonces lo podría ejecutar cualquiera que tuviera acceso a la clase. Una manera de evitar esto es declarando una clase interna anónima para que no se tenga acceso al método run().

Cuando hay más de un hilo de ejecución en un programa se corre el riesgo de que uno de ellos corrompa la información del otro. A los puntos en los que esto puede suceder se les llama *regiones críticas* y se evita la interferencia *sincronizando* las actividades de esas regiones críticas.

Se sincroniza el acceso a datos comunes a través de *candados* (*lock*s). El proceso que quiere trabajar sobre algún dato común, adquiere el candado de los objetos sobre los que va a trabajar. Todo objeto tiene un candado. Para trabajar con ese objeto, cualquier hilo de ejecución debe encontrar el candado disponible. Se usa el término *código sincronizado* para aquel código que se encuentra en un método o enunciado sincronizado.

11.4 Sincronización de hilos de ejecución

11.4.1. Métodos sincronizados

Un objeto que va a ser usado simultáneamente por más de un hilo de ejecución declara a los métodos "peligrosos" como *sincronizados* (synchronized). Cuando se solicita la ejecución de un método sincronizado, el hilo de ejecución primero adquiere el candado sobre el objeto. Realiza la ejecución solicitada, y al terminar el método libera el candado del objeto. Si otro hilo de ejecución intenta ejecutar cualquiera de los métodos sincronizados del objeto, se bloqueará y esperará hasta que el candado del objeto esté disponible. Si los métodos no son sincronizados podrán trabajar sobre el objeto independientemente del estado del candado.

Para evitar que algún hilo de ejecución se bloquee a sí mismo, el "dueño" del candado es el hilo de ejecución mismo. Por lo tanto, si se encuentra ejecutando algún método sincronizado sobre cierto objeto, la ejecución de más métodos, sin-

cronizados o no, sobre el mismo objeto procederá sin problemas, ya que el hilo de ejecución posee el candado respectivo. El candado se libera cuando el hilo de ejecución termina de ejecutar el primer método sincronizado que invocó, que es el que le proporcionó el candado. Si esto no se hiciera así, cualquier método recursivo o de herencia, por ejemplo, bloquearía la ejecución llegando a una situación en la que el hilo de ejecución no puede continuar porque el candado está comprometido (aunque sea consigo mismo), y no puede liberar el candado porque no puede continuar.

El candado se libera automáticamente en cualquiera de estos tres casos:

- El método termina normalmente porque se encuentra un enunciado return.
- El método termina normalmente porque llegó al final de su cuerpo.
- El método termina anormalmente lanzando una excepción.

El hecho de que la liberación del candado sea automática libera al programador de la responsabilidad de adquirir y liberar el candado, evitando así que algún objeto quede congelado porque se olvidaron de liberar su candado.

Para manejar una cuenta de cheques desde varias posibles terminales, el código se haría como se muestra en el listado 11.4.

Código 11.4 Manejo sincronizado de una cuenta de cheques 1/2

```
1: class SobregiradoException extends Exception  {
2:     public SobregiradoException(String msg)    {
3:         super(msg);
4:     }
5: }
6:
7: class Cuenta   {
8:     private double balance;
9:
10:    public Cuenta(double depositoInicial)    {
11:        balance = depositoInicial;
12:    }
13:
14:    public synchronized double getBalance()    {
15:        return balance;
16:    }
17:
18:    public synchronized void deposito(double cantidad)    {
19:        balance += cantidad;
20:    }
```

Código 11.4 Manejo sincronizado de una cuenta de cheques 2/2

```
21:     public synchronized void cheque(double cantidad)
22:                             throws SobregiradoException     {
23:         if (balance >= cantidad)  {
24:             balance -= cantidad;
25:         }
26:         else  {
27:             throw new SobregiradoException("Cuenta sobregirada");
28:         }
29:     }
30: }
```

Revisemos con cuidado la sincronización que pedimos en la clase Cuenta. El constructor no está sincronizado, porque la creación de una nueva cuenta puede hacerse únicamente desde un solo hilo. Además, los constructores no pueden estar sincronizados. Pero debemos proteger al atributo balance, ya que no puede suceder al mismo tiempo que sea modificado y se solicite su valor. Tampoco puede suceder que sea modificado en direcciones opuestas.

En estos momentos debe quedar claro por qué se ha elegido declarar los campos de datos como privados (o en el mejor de los casos, protegidos) y definir métodos de acceso y modificación. Si los campos fueran declarados públicos o de paquete, no habría manera de protegerlos de accesos inválidos desde múltiples hilos de ejecución.

Con la declaración de synchronized se garantiza que dos o más hilos de ejecución no van a interferirse entre sí. Lo que no se puede garantizar es el orden en que se van a ejecutar los distintos hilos de ejecución. Si dos hilos presentan solicitudes simultáneas, es impredecible cuál de ellas va a ser atendida primero. Para garantizar el orden de ejecución frente a procesos simultáneos, los hilos de ejecución tienen que coordinarse entre sí de alguna manera que va a depender de la aplicación.

11.4.2. Sincronización de métodos y la herencia

Cuando el método sincronizado de una superclase se extiende mediante herencia, el método de la subclase puede o no estar sincronizado. El programador podría programar la subclase de tal manera que ya no hubiera la posibilidad de interferir o ser interferido por algún otro hilo de ejecución. Sin embargo, si el método no sincronizado de la subclase invoca al método sincronizado de la su-

perclase, esta invocación deberá estar sincronizada. A la inversa también puede suceder: un método no sincronizado en la superclase puede extenderse a un método sincronizado en la subclase.

11.4.3. Métodos estáticos sincronizados

También los datos estáticos de las clases se pueden sincronizar, utilizando para ello métodos estáticos. Cuando algún hilo de control ejecuta alguno de estos métodos, ningún otro hilo de control podrá invocarlos, exactamente de la misma manera que con los métodos que pertenecen a objetos. Sin embargo, los objetos de la clase si podrán ser usados por otros hilos de control, ya que el candado se referirá únicamente al objeto que representa a la clase, no a los objetos de ese tipo.

11.4.4. Enunciados sincronizados

Puede suceder que queramos sincronizar el acceso no únicamente al objeto vigente (*this*), sino de cualquier otro objeto involucrado en nuestro código, y no queremos que la sincronización dure todo el método, sino únicamente mientras se ejecutan ciertos enunciados. Podemos entonces sincronizar un pedazo de código con la siguiente sintaxis:

Sintaxis:

⟨*enunciado de sincronización*⟩::= **synchronized(** ⟨*expr*⟩**)** {
⟨*enunciados*⟩
}

Semántica:

La ⟨*expr*⟩ tiene que regresar una referencia a un objeto. Mientras se ejecutan los ⟨*enunciados*⟩, el objeto referido en la ⟨*expr*⟩ queda sincronizado. Si se desea sincronizar más de un objeto, se recurrirá a enunciados de estos anidados. El candado quedará liberado cuando se alcance el final de los enunciados, o si se lanza alguna excepción dentro del grupo de enunciados.

Veamos un ejemplo de sincronización de un grupo de enunciados en el listado 11.5 en la siguiente página.

Código 11.5 Sincronización selectiva sobre objetos

```
1:     //------ Cambiar a los elementos de un arreglo
2:     //------ por su valor absoluto -------
3:     public static void abs(int[] valores)    {
4:       synchronized(valores)   {
5:         for(int i = 0; i < valores.length; i ++)
6:           if ( valores[i] < 0)
7:               valores[i] = -valores[i];
8:       }
9:     }
```

A este tipo de sincronización se le conoce como *sincronización del cliente*, porque son los usuarios del objeto los que se ponen de acuerdo para sincronizar su uso. Por ejemplo, en el caso de los arreglos, no hay otra manera de sincronizarlos, ya que no tenemos acceso a los métodos para sincronizarlos desde el "servidor". Los enunciados de sincronización permiten:

- Mantener con candado a un objeto el menor tiempo posible (únicamente el indispensable).
- Adquirir el candado de otros objetos diferentes de this.

Veamos un ejemplo de sincronización con objetos distintos de this en el listado 11.6.

Código 11.6 Sincronización de variables primitivas en enunciados 1/2

```
1: class GruposSeparados  {
2:     private double aVal = 0.0;
3:     private double bVal = 1.1;
4:     protected Object lockA = new Object();
5:     protected Object lockB = new Object();
6:
7:     public double getA()    {
8:         synchronized (lockA)  {
9:             return aVal;
10:        }
11:    }
12:
13:    public void setA(double val)    {
14:        synchronized (lockA) {
15:            aVal = val;
16:        }
17:    }
```

Código 11.6 Sincronización de variables primitivas en enunciados 2/2

```
18:     public double getB()    {
19:         synchronized (lockB)  {
20:             return bVal;
21:         }
22:     }
23:
24:     public void setB(double val)    {
25:         synchronized (lockB)  {
26:             bVal = val;
27:         }
28:     }
29:
30:     public void reset()    {
31:         synchronized (lockA) {
32:             synchronized(lockB) {
33:                 aVal = bVal = 0.0;
34:             }
35:         }
36:     }
37: }
```

Este tipo de sincronización la hace el servidor. Elige sincronizar únicamente a los datos primitivos del objeto, y cada uno de estos datos independiente del otro. Lo consigue restringiendo el acceso a los mismos únicamente a través de métodos de la clase, donde asocia a un objeto arbitrario para que maneje el candado de cada uno de los datos. De esta manera, como el hilo de ejecución que llega al proceso adquiere el candado del objeto, ningún otro hilo de ejecución puede entrar a la región crítica hasta que el primero que llegó libere el candado. En el método reset() se adquieren ambos candados antes de entrar a la región crítica, que es la que modifica el valor de los datos primitivos. Como ya habíamos mencionado, si queremos sincronizar con más de un objeto simplemente anidamos los enunciados de sincronización. Cualquier hilo de ejecución que intente ejecutar la región crítica tendrá que librar ambos obstáculos.

Lo que se pretende con la sincronización es que un método (o grupo de enunciados) se ejecuten como si fueran atómicos, esto es, que nada los interrumpa desde que empiezan a ejecutarse hasta que llegan al final. Por supuesto que estamos hablando de la ejecución del bytecode en la JVM, donde cada enunciado escrito en Java se traduce a varios enunciados del lenguaje de máquina. Por lo tanto, algún otro proceso concurrente podría ejecutarse antes de que se termine de ejecutar el enunciado peligroso.

11.4.5. Modelos de sincronización

Regresemos al esquema de *cliente-servidor* en el cual el objeto que da el servicio es el servidor mientras que el que lo solicita es el cliente. La pregunta importante en la sincronización es ¿quién deberá ser el responsable de la misma? Trataremos de dar algunas ideas al respecto.

En general en la orientación a objetos, como cada objeto se conoce a sí mismo debe encargarse de su propia sincronización, esto es, de protegerse en regiones críticas de manera adecuada. Esto respondería a que sea el servidor, el proveedor de servicios, el encargado de manejar la sincronía. Pero pudiera suceder que el diseñador de la clase que va a dar los servicios no pensó en que sus objetos fueran usados concurrentemente. En ese caso, será la responsabilidad del cliente el manejar la sincronización entre los distintos procesos que quieran trabajar sobre algún objeto.

El problema de sincronización es complejo y en algunos casos puede llevar a lo que se conoce como un *abrazo mortal* – *deadlock* – en el que un proceso está esperando adquirir un candado que no es soltado por otro procesa que espera adquirir el candado que tiene el primer proceso. De esta manera, ambos procesos están esperando a que el otro proceso libere el candado que necesitan para proseguir.

11.5 Comunicación entre hilos de ejecución

Si bien el mecanismo de sincronización evita que los procesos interfieran unos con otros, no tenemos hasta el momento ninguna manera de que los hilos de ejecución se comuniquen entre sí. Por ejemplo, en el caso del servidor de impresora debe suceder que cuando la cola de impresión esté vacía, el servidor de impresora libere el recurso y espere hasta que algún trabajo de impresión ingrese a la misma. Si no es así, el servidor está trabajando inútilmente, revisando la cola continuamente hasta que haya algo, pero manteniendo en su poder el candado, lo que no permitiría que ningún objeto ingresara a la cola. Asimismo, para que esto funcione adecuadamente, se requiere que el proceso que coloca elementos en la cola avise a los procesos que están esperando que ya hay algo que sacar. Tenemos, entonces, dos aspectos en este tipo de comunicación: por un lado, el proceso que está dispuesto a esperar a que algo suceda, y por el otro el proceso que logra que algo suceda y lo notifica.

La forma general para que un proceso espere a que una condición se cumpla es

```
synchronized void doWhenCondition()  {
  while (! condicion)
      wait();
  //... Se hace lo que se tiene que hacer cuando la
  //    condición es verdadera
}
```

Cuando se usa el enunciado wait se deben considerar los siguientes puntos:

- Como hay varios hilos de ejecución, la condición puede cambiar de estado aún cuando este hilo de ejecución no esté haciendo nada. Sin embargo, es importante que estos enunciados estén sincronizados. Si estos enunciados no están sincronizados, pudiera suceder que el estado de la condición se hiciera verdadero, pero entre que este código sigue adelante, algún otro proceso hace que vuelva a cambiar el estado de la condición. Esto haría que este proceso trabajara sin que la condición sea verdadera.

- La definición de wait pide que la suspensión del hilo de ejecución y la liberación del candado sea una operación *atómica*, esto es, que nada pueda suceder ni ejecutarse entre estas dos operaciones. Si no fuera así podría suceder que hubiera un cambio de estado entre que se libera el candado y se suspende el proceso, pero como el proceso ya no tiene el candado, la notificación se podría perder.

- La condición que se está probando debe estar *siempre* en una iteración, para no correr el riesgo de que entre que se prueba la condición una única vez y se decide qué hacer, la condición podría cambiar otra vez.

Por otro lado, la notificación de que se cambió el estado para una condición generalmente toma la siguiente forma:

```
synchronized voif changeCondition() {
    // ... Cambia algún valor que se usa en la prueba
    //     de la condición
    notifyAll();     // o bien notify()
}
```

Si se usa notifyAll() todos los procesos que están en estado de espera – que ejecutaron un wait() – se van a despertar, mientras que notify() únicamente despierta a un hilo de ejecución. Si los procesos están esperando distintas condiciones se debe usar notifyAll(), ya que si se usa notify() se corre el riesgo de que despierte un proceso que está esperando otra condición. En cambio, si todos los procesos están esperando la misma condición y sólo uno de ellos puede reiniciar su ejecución, no es importante cuál de ellos se despierta.

Volvamos al ejemplo del servidor de impresora, pero incluyendo ahora un hilo de ejecución que produce trabajos de impresión.

11.5.1. Ejemplo: un servidor de impresora

Como sabemos a estas alturas, en los sistemas Unix las impresoras están conectadas a los usuarios a través de un servidor de impresión. Cada vez que un usuario solicita la impresión de algún trabajo, lo que en realidad sucede es que se arma la solicitud del usuario y se forma en una *cola de impresión*, que es atendida uno a la vez. En una estructura de datos tipo *cola*, el primero que llega es el primero que es atendido. Por supuesto que el manejo de la cola de impresión se tiene que hacer en al menos dos hilos de ejecución: uno que forma a los trabajos que solicitan ser impresos y otra que los toma de la cola y los imprime. En el listado 11.7 vemos la implementación de una cola genérica o abstracta (sus elementos son del tipo Object) usando para ello listas ligadas. Como los elementos son objetos, se puede meter a la cola cualquier tipo de objeto, ya que todos heredan de Object. Se usa para la implementación listas ligadas, ya que siendo genérica no hay una idea de cuantos elementos podrán introducirse a la cola. Cuando se introduce algún elemento a la cola en el método add, se notifica que la cola ya no está vacía. Cuando se intenta sacar a un elemento de la cola, si la cola está vacía, el método take entra a una espera, que rompe hasta que es notificado de que la cola ya no está vacía.

Código 11.7 Cola genérica con operaciones sincronizadas 1/2

```
1:  class Queue   {
2:     /* Primero y último elementos de la cola */
3:     private Cell head, tail;
4:
5:     public synchronized void add(Object o)    {
6:         Cell p = new Cell(o);      // Envolver a o en una Cell
7:
8:         if (tail == null)  {
9:             head = p;
10:        }
11:        else  {
12:            tail.setNext(p);
13:        }
14:        p.setNext( null);
```

Código 11.7 Cola genérica con operaciones sincronizadas 2/2

```
15:         tail = p;
16:         notifyAll();  // Avísale a los que esperan,
17:                       // el objeto fue añadido
18:     }
19:     public synchronized Object take()
20:                 throws InterruptedException    {
21:         while (head == null)
22:             wait();              // Espera un objeto en la cola
23:         Cell p = head;           // Recuerda al primer objeto
24:         head = head.getNext();   //  Quítalo de la cola
25:         if (head == null)
26:             tail = null;
27:         return p.getElement();
28:     }
29:   }
```

La definición de cada uno de los elementos de la cola se encuentra en el listado 11.8.

Código 11.8 Definición de elementos de la cola

```
1:  class Cell  {
2:     private Cell next;
3:     private Object element;
4:     public Cell(Object element)    {
5:         this.element = element;
6:     }
7:     public Cell(Object element, Cell next)    {
8:         this.element = element;
9:         this.next = next;
10:    }
11:    public Object getElement()    {
12:        return element;
13:    }
14:    public void setElement(Object Element)    {
15:        this.element = element;
16:    }
17:    public Cell getNext()    {
18:        return next;
19:    }
20:    public void setNext(Cell next)    {
21:        this.next = next;
22:    }
23: }
```

En el listado 11.9 tenemos la definición concreta de lo que es un trabajo de impresión. Se redefine el método toString() de Object para poder escribir el trabajo directamente en la impresora.

Código 11.9 PrintJob: trabajo de impresión

```
1: public class PrintJob   {
2:     private int numLines;
3:     private String name;
4:     public PrintJob(String name, int numLines)    {
5:         this.name = name;
6:         this.numLines = numLines;
7:     }
8:     public String toString()    {
9:         return name.trim() + "␣" + numLines + "␣lines␣printed";
10:    }
11: }
```

Con todas las clases que usa el servidor de impresión ya definidas, en el listado 11.10 mostramos el servidor de impresión propiamente dicho.

Código 11.10 Servidor de impresión que corre en un hilo propio de ejecución

```
1: import java.util.*;
2:
3: class PrintServer implements Runnable  {
4:     private Queue requests = new Queue();
5:     public PrintServer()      {
6:         new Thread(this).start();
7:     }
8:     public void print(PrintJob job)    {
9:         requests.add(job);
10:    }
11:    public void run()    {
12:        try  {
13:            for( ; ; )
14:                realPrint((PrintJob)requests.take());
15:        } catch(InterruptedException e)   {
16:            System.out.println("La␣impresora␣quedó␣fuera␣de␣línea");
17:            System.exit(0);
18:        }
19:    }
20:    private void realPrint(PrintJob job)    {
21:        System.out.println("Imprimiendo␣" + job);
22:    }
23: }
```

Finalmente, para simular un servidor de impresión que recibe y procesa trabajos de impresión construimos la clase SistOperativo, que proporciona un entorno en el cual el servidor de impresión se arranca para que empiece a funcionar. Esta clase se muestra en el listado 11.11.

Código 11.11 Entorno en el que funciona un servidor de impresión

```
1: class SistOperativo extends Thread   {
2:     private PrintServer pr = new PrintServer();
3:     private  int numJob =0;
4:     public void run()    {
5:         try  {
6:             while(true)  {
7:                 numJob++;
8:             // Se generan trabajos con un número arbitrario
9:             // de líneas por imprimir
10:                pr.print(new PrintJob("Job\#" + numJob + ",␣",
11:                                ((int)Math.floor(Math.random()
12:                                   * 10000)) + 1));
13:                sleep(300);
14:            }
15:        } catch (InterruptedException e) {
16:            System.out.println("Terminó␣el␣trabajo");
17:            System.exit(0);
18:        }
19:    }
20:    public static void main(String[] args)    {
21:        new SistOperativo().start();
22:    }
23: }
```

Una ejecución breve de este programa produce se puede ver en la figura 11.4 en la siguiente página.

11.5.2. Más detalles sobre la espera y la notificación

Los métodos wait – que tienen tres variantes – y los métodos que corresponden a la notificación – notify() y notifyAll – son métodos de la clase Object y por lo tanto todas las clases los pueden invocar. Todos estos métodos están declarados final por lo que no pueden ser modificados. Veamos las variantes que presentan:

Métodos de espera:

```
public final void wait(long timeout) throws InterruptedException
```

El hilo de control en el que aparece el wait se duerme hasta que sucede una de cuatro cosas:

Figura 11.4 Salida que produce el servidor de impresión.

```
% java SistOperativo
Imprimiendo Job#1, 920 lines printed
Imprimiendo Job#2, 8714 lines printed
Imprimiendo Job#3, 2297 lines printed
Imprimiendo Job#4, 799 lines printed
Imprimiendo Job#5, 1966 lines printed
Imprimiendo Job#6, 2439 lines printed
Imprimiendo Job#7, 6808 lines printed
Imprimiendo Job#8, 8270 lines printed
Imprimiendo Job#9, 7093 lines printed
Imprimiendo Job#10, 63 lines printed
Imprimiendo Job#11, 2904 lines printed
Imprimiendo Job#12, 5396 lines printed
Imprimiendo Job#13, 8980 lines printed
Imprimiendo Job#14, 6519 lines printed
Imprimiendo Job#15, 4359 lines printed
Imprimiendo Job#16, 2179 lines printed
Imprimiendo Job#17, 9404 lines printed
Imprimiendo Job#18, 8575 lines printed
Imprimiendo Job#19, 2095 lines printed
Imprimiendo Job#20, 6245 lines printed
Imprimiendo Job#21, 4016 lines printed
Imprimiendo Job#22, 9531 lines printed
```

Métodos de espera: (continúa...)

i. Se invoca notify sobre este objeto y se selecciona al hilo de ejecución como ejecutable.
ii. Se invoca notifyAll sobre este objeto.
iii. El tiempo especificado para esperar expira.
iv. Se invoca al método interrupt – después veremos con más detalle este método – del hilo de ejecución.

Si timeout es cero, el proceso esperará indefinidamente hasta que sea notificado que puede continuar. Durante la espera, se libera el candado del objeto, y éste es readquirido antes de que termine wait, no importa cómo o por qué termine. Si wait termina porque el hilo de ejecución fue interrumpido,

el método lanza una excepción de InterruptedException.

```
public final void wait(long timeout, int nanos)
```

Da la posibilidad de esperar tiempo en nanosegundos también. El proceso dormirá el número de timeout de milisegundos más nanos nanosegundos.

```
public final void wait() throws InterruptedException
```

Equivalente a wait(0).

Métodos de notificación:

```
public final void notifyAll()
```

Notifica a *todos* los procesos que están en estado de espera que alguna condición cambió. Despertarán los procesos que puedan readquirir el candado que están esperando.

```
public final void notify()
```

Notifica *a lo más* a un hilo esperando notificación, pero pudiera suceder que el hilo notificado no estuviera esperando por la condición que cambió de estado. Esta forma de notificación sólo se debe usar cuando hay seguridad de quién está esperando notificación y por qué.

Los métodos de notificación lo que hacen, de hecho, es "ofrecer" el candado que poseen a los métodos que están esperando. Si ningún método está esperando, o el que recibió la notificación no está esperando ese candado en particular, la notificación se pierde.

Tanto los métodos de espera como los de notificación tienen que invocarse desde código sincronizado, usando el candado del objeto sobre el cual son invocados. La invocación se puede hacer directamente desde el código sincronizado, o bien desde métodos invocados desde el código sincronizado. Si se intenta invocar a cualquiera de estos dos métodos cuando no se posee el candado del objeto, se presentará una excepción IllegalMonitorStateException.

No hay manera de saber el porqué un objeto despierta, si es porque transcurrió el tiempo especificado, o porque fue notificado de un cambio de estado. Esto se puede averiguar si se obtiene el tiempo transcurrido durante la espera. Esto último se puede lograr con el método estático currentTimeMillis() de la clase System, como mostramos en el listado 11.12 en la siguiente página.

Código 11.12 Para verificar tiempo transcurrido.

```
......
1: public synchronized Object take()
2:              throws InterruptedException     {
3:   while (head == null)  {
4:     long tiempo = System.currentTimeMillis();
5:     wait(300);       // Espera un objeto en la cola
6:     long transcurrido = System.currentTimeMillis() - tiempo;
7:     if (transcurrido > 300)
8:       System.err.println("Se despertó sin que hubiera cambios");
9:   }

     ......
```

11.6 Alternativas para la programación de procesos

Hasta ahora, si hay varios procesos esperando para reanudar su ejecución, es impredecible el orden en que van a reanudar su ejecución, ya que es la JVM la que decide a cuál proceso atender. Pero dentro de nuestros programas, algunas tareas van a ser más importantes que otras, o vamos a desear influir, aunque sea un poco, en que esas tareas entren a ejecución antes que otras.

Se puede pensar en varios procesos ejecutándose simultáneamente aún cuando la máquina tenga un solo procesador. Para ello, el sistema operativo (o en nuestro caso, la JVM) tiene políticas de distribución del tiempo de procesador. Cada proceso tiene una *prioridad* asociada, que da una indicación de la importancia del mismo. La política del sistema puede ser que va a llevar a ejecución a los procesos con más altas prioridad de entre los que están en posibilidades de ejecutarse. El proceso seleccionado seguirá ejecutándose hasta que desee ejecutar una operación que bloque la ejecución (un wait, sleep o alguna operación de entrada y salida). En ese momento se le quita el uso del procesador y se le da a otro proceso, poniéndolo a esperar. El sistema también pudiera tener la política de repartir el tiempo de procesador de manera equitativa, permitiendo a cada proceso ejecutarse un cierto lapso. En este caso, los procesos son desalojados del procesador aún cuando están en estado ejecutable, y simplemente esperarán hasta que les vuelva a tocar otro lapso de ejecución.

El que el sistema desaloje a un proceso depende más de las políticas del sis-

tema que del tipo de proceso. En general es aceptable asumir que el sistema le dará preferencia a los procesos con más alta prioridad y que estén en estado de poder ser ejecutados, pero no hay ninguna garantía al respecto. La única manera de modificar estas políticas de desalojo es mediante la comunicación de procesos.

La prioridad de un hilo de ejecución es, en principio, la misma que la del proceso que lo creó. Esta prioridad se puede cambiar con el método public final void setPriority(int newPriority) que asigna una nueva prioridad al hilo de ejecución. Esta prioridad es un valor entero tal que cumple

$$\textbf{MIN\_PRIORITY} \leqslant \textbf{newPriority} \leqslant \textbf{MAX\_PRIORITY}$$

La prioridad de un hilo que se está ejecutando puede cambiarse en cualquier momento. En general, aquellas partes que se ejecutan continuamente en un sistema deben correr con prioridad menor que los que detectan situaciones más raras, como la alimentación de datos. Por ejemplo, cuando un usuario oprime el botón de cancelar para un proceso, quiere que éste termine lo antes posible. Sin embargo, si se están ejecutando con la misma prioridad, puede pasar mucho tiempo antes de que el proceso que se encarga de cancelar tome el control.

En general es preferible usar prioridades cercanas a NORM_PRIORITY para evitar comportamientos extremos en un sistema. Si un proceso tiene la prioridad más alta posible, puede suceder que evite que cualquier otro proceso se ejecute, una situación que se conoce como *hambruna* (*starvation*).

11.6.1. Desalojo voluntario

Hay varios métodos mediante los cuales un programa puede desalojarse a sí mismo del procesador. Todos estos métodos de la clase Thread son estáticos. Veamos cuáles son.

public static void sleep(long millis) throws InterruptedException
: Pone a dormir al hilo de ejecución en cuestión al menos por el tiempo especificado. Sin embargo, no hay garantía de que sea exactamente este tiempo, ya que las políticas de programación pudieran no permitir que el proceso regrese al procesador en cuando despierta. Si el proceso es interrumpido mientras está durmiendo, se termina y lanza una excepción InterruptedException.

public static void sleep(long millis, int nanos) throws InterruptedException
: Lo mismo que el anterior, pero se pueden agregar nanosegundos, que estarán en el rango 0-999999.

public static void yield()

Le indica al programador de procesos que está dispuesto a ser desalojado, ya que la tarea que está realizando no tiene que continuar en este momento. El programador de procesos decide si lo desaloja o no.

Ya hemos utilizado el método sleep. En la clase del listado 11.13 mostramos un ejemplo de yield. Esta clase se ejecuta dándole en la línea de comandos varios argumentos.

Código 11.13 Desalojo voluntario

```
1: class Babble extends Thread  {
2:     static boolean declina;     // Declinar a otros procesos?
3:     static int howOften;          // cuantas veces imprimir
4:     private String word;
5:
6:     Babble(String whatToSay)     {
7:         word = whatToSay;
8:     }
9:
10:    public void run()      {
11:        for(int i = 0; i < howOften; i++) {
12:            System.out.println(word);
13:            if (declina)
14:                Thread.yield();        // Deja que otros procesos
15:                                       // se ejecuten
16:        }
17:    }
18:
19:    public static void main(String[] args)     {
20:        declina = new Boolean(args[0]).booleanValue();
21:        howOften = Integer.parseInt(args[1]);
22:
23:          // Crea un hilo de ejecución para cada palabra
24:        for (int i = 2; i < args.length; i++)
25:            new Babble(args[i]).start();
26:    }
27: }
```

Podemos ver dos ejecuciones con líneas de comandos distintas, una que provoca el desalojo y la otra no. Sin embargo, en el sistema en que probamos este pequeño programa, el resultado es el mismo. Deberíamos ejecutarlo en varios sistemas para ver si el resultado cambia de alguna manera. Veamos los resultados en la figura 11.5 en la página opuesta.

Figura 11.5 Ejecución con desalojo voluntario.

```
% java Babble false 2 Did DidNot
Did
Did
DidNot
DidNot
% java Babble true 2 Did DidNot
Did
Did
DidNot
DidNot
```

11.7 Abrazo mortal (deadlock)

Siempre que se tienen dos hilos de ejecución y dos objetos con candados se puede llegar a una situación conocida como *abrazo mortal*, en la cual cada proceso posee el candado de uno de los objetos y está esperando adquirir el candado del otro proceso. Si el objeto X tiene un método sincronizado que invoca a un método sincronizado del objeto Y, quien a su vez tiene a un método sincronizado invocando a un método sincronizado de X, cada proceso se encontrará esperando a que el otro termine para poder continuar, y ninguno de los dos va a poder hacerlo.

En el listado 11.14 mostramos una clase Apapachosa en la cual un amigo, al ser apapachado, insiste en apapachar de regreso a su compañero.

Código 11.14 Posibilidad de abrazo mortal 1/2

```
1: class Apapachosa {
2:     private Apapachosa amigo;
3:     private String nombre;
4:     public Apapachosa(String nombre) {
5:         this.nombre = nombre;
6:     }
7:     public synchronized void apapacha() {
8:         System.out.println(Thread.currentThread().getName() +
9:                 "␣en␣" + nombre + ".apapacha()␣tratando␣de␣" +
10:                "invocar␣a␣" + amigo.nombre + ".reApapacha()");
11:        amigo.reApapacha();
12:    }
```

Código 11.14 Posibilidad de abrazo mortal 2/2

```
13:
14:     private synchronized void reApapacha()    {
15:         System.out.println(Thread.currentThread().getName()+
16:                     "␣en␣" + nombre + ".reApapacha()");
17:     }
18:
19:     public void seAmigo(Apapachosa amiga)    {
20:         this.amigo = amiga;
21:     }
22:
23:     public static void main(String[] args)    {
24:         final Apapachosa lupe = new Apapachosa("Lupe");
25:         final Apapachosa juana = new Apapachosa("Juana");
26:         lupe.seAmigo(juana);
27:         juana.seAmigo(lupe);
28:
29:         new Thread(new Runnable() {
30:              public void run() { lupe.apapacha(); }
31:         }, "Hilo1").start();
32:             new Thread(new Runnable() {
33:                 public void run() {    juana.apapacha();    }
34:         }, "Hilo2").start();
35:     }
36: }
```

Dependiendo del encargado de la programación, pudiera suceder que lupe termine antes que empiece juana. Pero esto no lo podemos garantizar. Una ejecución en nuestra máquina logró lo que acabamos de mencionar, y produjo la salida que se muestra en la figura 11.6.

Figura 11.6 Ejecución con la posibilidad de abrazo mortal.

```
% java Apapachosa
Hilo1 en Lupe.apapacha() tratando de invocar a Juana.reApapacha()
Hilo1 en Juana.reApapacha()
Hilo2 en Juana.apapacha() tratando de invocar a Lupe.reApapacha()
Hilo2 en Lupe.reApapacha()
```

En las corridas que hicimos en nuestra máquina siempre pudo Lupe terminar su intercambio antes de que Juana intentara el suyo. Sin embargo, pudiera suceder

que en otro sistema, o si la JVM tuviera que atender a más hilos de ejecución, ese programita bloqueara el sistema cayendo en abrazo mortal.

Introduciendo un enunciado wait dentro del método reaApapacha conseguimos que el proceso caiga en un abrazo mortal. Se lanza el hilo de ejecución con lupe, pero como tiene que esperar cuando llega a reApapacha(), esto permite que también el hilo de ejecución con juana se inicie. Esto hace que ambos procesos se queden esperando a que el otro libere el candado del objeto, pero para liberarlo tienen que terminar la ejecución de reApapacha, lo ninguno de los dos puede hacer – cada uno está esperando a que termine el otro. La aplicación modificada se muestra en el listado 11.15 y en la figura 11.7 en la siguiente página se muestra cómo se queda pasmado el programa, sin avanzar ni terminar, hasta que se teclea un ^C.

Código 11.15 Abrazo mortal entre hilos de ejecución 1/2

```
1: class Apapachosa2   {
2:     private Apapachosa2 amigo;
3:     private String nombre;
4:     private Object candado = new Object();
5:
6:     public Apapachosa2(String nombre)    {
7:         this.nombre = nombre;
8:     }
9:
10:    public  synchronized void apapacha()    {
11:                System.out.println(Thread.currentThread().getName() +
12:                            + "␣en␣" + nombre
13:                            + ".apapacha()␣tratando"
14:                            +  "␣de␣invocar␣" + amigo.nombre
15:                            + ".reApapacha()");
16:                amigo.reApapacha();
17:    }
18:
19:    private  synchronized void reApapacha()    {
20:        try {
21:             wait(3);
22:             System.out.println(Thread.currentThread().getName()
23:                                  + "␣en␣" + nombre + ".reApapacha()");
24:        } catch(InterruptedException e) { }
25:    }
26:
27:    public void hazteAmigo(Apapachosa2 amigo)    {
28:        this.amigo = amigo;
29:    }
```

Código 11.15 Abrazo mortal entre hilos de ejecución 2/2

```
30:
31:     public static void main(String[] args)    {
32:         final Apapachosa2 lupe = new Apapachosa2("Lupe");
33:         final Apapachosa2 juana = new Apapachosa2("Juana");
34:         lupe.hazteAmigo(juana);
35:         juana.hazteAmigo(lupe);
36:         new Thread(new Runnable()  {
37:                 public void run() {    lupe.apapacha();    }
38:             }, "Hilo1").start();
39:         new Thread(new Runnable() {
40:                 public void run() { juana.apapacha(); }
41:             }, "Hilo2").start();
42:     }
43: }
```

Como se puede ver en las líneas 22, 23 y 26, lo único que se le agregó a esta clase es que una vez dentro del método reApapacha espere 3 milisegundos, liberando el candado. Este es un tiempo suficiente para que el otro hilo de ejecución se apodere del candado. Acá es cuando se da el abrazo mortal, porque el primer hilo no puede terminar ya que está esperando a que se libere el candado, pero el segundo hilo tampoco puede continuar porque el primero no ha terminado.

Figura 11.7 Ejecución de Apapachosa2 con abrazo mortal

```
elisa@lambda ...ICC1/progs/threads  % java Apapachosa2
Hilo1 en Lupe.apapacha() tratando de invocar Juana.reApapacha()
Hilo2 en Juana.apapacha() tratando de invocar Lupe.reApapacha()

elisa@lambda ...ICC1/progs/threads %
```

Si bien nos costó trabajo lograr el abrazo mortal, en un entorno donde se están ejecutando múltiples aplicaciones a la vez no podríamos predecir que el segundo hilo de ejecución no se apoderara del candado del objeto antes de que el primer hilo lograra llegar a ejecutar el método reApapacha, por lo que tendríamos que hacer algo al respecto.

Es responsabilidad del cliente evitar que haya abrazos mortales, asegurándose del orden en que se ejecutan los distintos métodos y procesos. Esto lo conseguirá el programador usando enunciados wait, yield, notify, notifyAll y sincronizando alrededor de uno o más objetos, para conseguir que un proceso espere siempre a otro.

El tema de abrazos mortales es de gran importancia e interés, y proporciona un campo de estudio e investigación muy fértil. Siendo éste un curso introductorio no creemos necesario verlo más a fondo y nos conformamos con haber creado la conciencia de situaciones anómalas que se pueden presentar.

11.8 Cómo se termina la ejecución de un proceso

Un proceso en general puede presentar los siguientes estados:

vivo: Un método está vivo desde el momento que es iniciado con start() y hasta que termina su ejecución.

activo: Un método está activo si está siendo ejecutado por el procesador.

ejecutable: Na hay ningún obstáculo para que sea ejecutado, pero el procesador no lo ha atendido.

esperando: Se encuentra esperando a que alguna condición se cumpla o se le notifique de algún evento, por lo que el procesador no lo puede ejecutar.

Un método deja de estar vivo, como ya dijimos, por cualquiera de las causas que siguen:

- El método run ejecuta un return o llega al final de su definición.
- El método run termina abruptamente.
- Se invoca al método destroy sobre ese proceso.

Es posible que se detecte que algo mal está sucediendo en cierto proceso y se desee terminar su ejecución. Sin embargo, destroy debería ser un último recurso, ya que si bien se consigue que el proceso termine, pudiera suceder que no se liberen los candados que posee el proceso en cuestión y se queden en el sistema otros procesos bloqueados para siempre esperando candados que ya nunca van a poder ser liberados. Es tan drástico y peligroso este método para el sistema en general que muchas máquinas virtuales han decidido no implementarlo y simplemente lanzan una excepción NoSuchMethodError, que termina la ejecución del hilo. Esto sucede si modificamos el método apapacha como se muestra en el listado 11.16 en la siguiente página.

Código 11.16 Uso de **destroy** para terminar un hilo de ejecución

```
1:  public synchronized void apapacha() {
2:     System.out.println(Thread.currentThread().getName()
3:                        + "␣en␣" + nombre + ".apapacha()␣tratando"
4:                        + "␣de␣invocar␣" + amigo.nombre
5:                        + ".reApapacha()");
6:     if (Thread.currentThread().getName().equals("Hilo2")) {
7:         System.out.println("Entré␣a␣destroy");
8:         Thread.currentThread().destroy();
9:     }
10:    amigo.reApapacha();
11: }
```

Figura 11.8 Implementación de **destroy** en la máquina virtual de Java.

```
elisa@lambda ...ICC1/progs/threads % java Apapachosa2
Hilo1 en Lupe.apapacha() tratando de invocar Juana.reApapacha()
Hilo2 en Juana.apapacha() tratando de invocar Lupe.reApapacha()
Entré a destroy
java.lang.NoSuchMethodErrorHilo1 en Juana.reApapacha()
        at java.lang.Thread.destroy(Thread.java:709)
        at Apapachosa2.apapacha(Apapachosa2.java:20)
        at Apapachosa2$2.run(Apapachosa2.java:49)
        at java.lang.Thread.run(Thread.java:484)
```

La ejecución de Apapachosa2 con la redefinición de este método produce la salida que se observa en la figura 11.8.

Cuando un programa termina por cualquier razón, en ese momento terminan todos los procesos lanzados por este programa.

11.8.1. Cancelación de un proceso

En ocasiones es deseable poder cancelar un proceso que está trabajando. Por ejemplo, si estamos trabajando con interfaces gráficas y el usuario oprime el botón de "cancelar" deseamos poder terminar el proceso de una manera adecuada. Para poder pedir la cancelación de un proceso debemos poder interrumpirlo para que "escuche" la orden de terminación.

La clase Thread tiene dos métodos asociados a la interrupción de un proceso, la solicitud de interrupción y la consulta de si el proceso ha sido o no interrumpido.

La solicitud de interrupción se hace mediante el enunciado

```
// En el hilo 1
                    thread2.interrupt()
```

desde un método distinto que el que se desea interrumpir. En el método por interrumpir deberemos tener una iteración que en cada vuelta esté preguntando si ha sido o no interrumpido:

```
// En el hilo 2
                    while (! interrupted()) {
                          // haz el trabajo planeado
                    }
```

Si tenemos más de un proceso ejecutándose y se le envía una señal de interrupción a uno de los procesos, el otro proceso proseguirá, adueñándose del procesador. Al interrumpir a un proceso, los candados que posea el objeto se liberan, por lo que éste puede ser un mecanismo necesario para terminar con un abrazo mortal, aún a costa de uno de los procesos.

Veamos una aplicación de dos hilos de ejecución encargados de escribir un punto en la pantalla cada determinado tiempo. En un momento dado el programa principal decide interrumpir a uno de las procesos, y en ese momento el otro continúa su ejecución ya sin ceder el procesador a nadie más. Podemos ver este programa en el listado 11.17.

Código 11.17 Interrupción de procesos 1/2

```
1: class Tick extends Thread {
2:     int count;
3:     long pause;
4:
5:     public Tick(int count, long pauseTime, String name)    {
6:         super(name);
7:         this.count = count;
8:         pause = pauseTime;
9:     }
10:
11:    public void run()    {
12:        tick(count, pause);
13:    }
```

Código 11.17 Interrupción de procesos 2/2

```
14:     synchronized void  tick(int count, long pauseTime)    {
15:         System.out.println(Thread.currentThread());
16:         try   {
17:             for (int i = 0; i < count; i++)  {
18:                 System.out.println(currentThread().getName()
19:                                    + ">.");
20:                 System.out.flush();
21:                 Thread.sleep(pauseTime);
22:             }
23:         } catch(InterruptedException e)  {
24:             System.out.println("Se interrumpió " +
25:                                currentThread().getName());
26:         }
27:     }
28:
29:     public static void main(String[] args)    {
30:         Tick hilo1 = new Tick(20,100,"Hilo 1");
31:         Tick hilo2 = new Tick(10,100,"Hilo2");
32:         hilo1.start();
33:         hilo2.start();
34:         System.out.println("Desde el programa principal");
35:         try {
36:             sleep(300); // Deja que trabajen un rato
37:         }  catch(InterruptedException e)  {
38:             hilo1.interrupt();
39:         }
40:     }
41: }
```

La diferencia principal entre este método de interrupción o interrumpir con un Ĉ, por ejemplo, es que en este último caso estamos interrumpiendo al programa principal, por lo que todos los procesos que fueron lanzados desde él se dan por aludidos, mientras que si mandamos un mensaje de interrupción a un solo proceso, éste es el único que cachará la excepción y suspenderá su ejecución. La salida que produce esta aplicación se puede ver en la figura 11.9 en la página opuesta.

La ejecución del proceso hilo 1 se interrumpe porque al estar dormido (ejecutando un sleep) el proceso sale de este estado al recibir la señal de interrupción. Entonces, el proceso terminará.

Un enunciado interrupt no va a tener ningún efecto si el proceso correspondiente no está en un estado de espera (wait o sleep). Tiene efecto en estos estados porque estos métodos lanzan la excepción InterruptedException, pero no porque de hecho se suspenda inmediatamente el mismo.

Figura 11.9 Interrupción de un hilo de ejecución desde el programa principal.

```
elisa@lambda ...ICC1/progs/threads % java Tick
Thread[Hilo 1,5,main]
Hilo 1>.
Desde el programa principal
Thread[Hilo2,5,main]
Hilo2>.
Hilo 1>.
Hilo2>.
Hilo 1>.
Hilo2>.
Se interrumpió Hilo 1
Hilo2>.
Hilo2>.
Hilo2>.
Hilo2>.
Hilo2>.
Hilo2>.
Hilo2>.
elisa@lambda ...ICC1/progs/threads %
```

Si el programa solicita la interrupción de un proceso es muy peligroso hacer caso omiso de esta solicitud, cachando la excepción y no haciendo nada al respecto. En algunas ocasiones se usa interrupt() para desbloquear a un proceso que está esperando algo, y al no responder en ninguna forma a la excepción de hecho el proceso no se interrumpe.

Código 11.18 Significado de la interrupción en un proceso

```
1:    synchronized void  tick(int count, long pauseTime)    {
2:        System.out.println(currentThread());
3:        for (int i = 0; i < count; i++)  {
4:            try  {
5:                System.out.println(currentThread().getName() +">.");
6:                System.out.flush();
7:                sleep(pauseTime);
8:            } catch(InterruptedException e) {
9:                System.out.println("Se interrumpió "
10:                                   + currentThread().getName());
11:           }
12:       }
13:   }
```

Si en el método tick del ejemplo que estamos manejando intercambiamos el try con la iteración, como se muestra en el listado 11.18 en la página anterior, el proceso recibe la señal de interrupción en una de las iteraciones, pero como no hace nada al momento de cacharla respecto al hilo de ejecución, al regresar una vez más a la iteración resulta que el hilo supuestamente suspendido no fue suspendido y sigue compartiendo el procesador con el otro hilo.

11.8.2. Terminación de la ejecución de un proceso

Una de las razones que puede tener un programa para lanzar un proceso paralelo es la necesidad de hacer cálculos complejos, que pudieran hacerse simultáneamente. Sin embargo, muchas veces no se sabe cuál de los dos cálculos se va a tardar más, por lo que el proceso principal tendrá que esperar a que termine, en su caso, el proceso que lanzó antes de poder utilizar los resultados del mismo.

Java tiene el método join() de la clase Thread que espera a que el proceso con el que se invoca el método termine antes de que se proceda a ejecutar la siguiente línea del programa. Veamos en los listados 11.19 y 11.20 en la página opuesta un ejemplo en el quesupuestamente se desean hacer dos cálculos que pueden llevarse a cabo de manera simultánea. El método principal (main) crea un proceso paralelo para que se efectúe uno de los cálculos mientras él ejecuta el otro. Una vez que termina de ejecutar su propio cálculo, "se sienta" a esperar hasta que el proceso que lanzó termine.

Código 11.19 Espera para la terminación de un coproceso

```
1: class CalcThread extends Thread  {
2:     private double result;
3:     public void run()     {
4:         result = calculate();
5:     }
6:     public double getResult()     {
7:         return result;
8:     }
9:     public double calculate()     {
10:        int tope =((int)Math.floor(Math.random()*100000));
11:        for (int i = 0; i < tope; i++);
12:        return ((double)System.currentTimeMillis());
13:    }
14: }
```

En el programa principal, desde el que se lanza el coproceso del cálculo, una

vez hecha la tarea que se podía realizar de manera simultánea, se espera para garantizar que el coproceso terminó. Si esto no se hace así pudiera suceder que el valor que se desea calcule el coproceso no estuviera listo al terminar el proceso principal con su propio trabajo.

Código 11.20 Programa principal para ejemplificar la espera

```
1: class ShowJoins  {
2:     public static void main(String[] args)    {
3:         long antes, despues;
4:         CalcThread calc = new CalcThread();
5:         calc.start();
6:         int tope =((int)Math.floor(Math.random()*10000));
7:         for(int i=0; i < tope; i++);
8:         try {
9:             antes = System.currentTimeMillis();
10:            calc.join();
11:            despues = System.currentTimeMillis();
12:            System.out.println("Esperé␣"+(despues - antes) +
13:                                        "␣milisegundos");
14:            System.out.println("result␣vale␣" +
15:                        calc.getResult());
16:        } catch (InterruptedException e) {
17:            System.out.println("No␣hay␣respuesta:␣fue␣interrumpido");
18:        }
19:    }
20: }
```

Otra manera de esperar a que un proceso termine antes de continuar con un cierto hilo de ejecución es mediante el método isAlive() de la clase Thread, que nos indica si el método todavía no ha llegado a su final, ya sea porque fue interrumpido o porque terminó su tarea. Si eligiéramos usar esta forma de sincronización, el programa del listado 11.20 se vería como se muestra en el listado 11.21 en la siguiente página, con la parte que cambió en negritas.

En la línea 11 del listado 11.21 en la siguiente página todo lo que hace el programa es estar en un ciclo hasta que la condición del ciclo se vuelva falsa. Como la condición es que el hilo de ejecución calc esté vivo, el programa podrá salir del ciclo cuando el proceso calc deje de estar vivo, o sea cuando termine.

Código 11.21 Verificación de terminación con **isAlive()**

```
1: class ShowAlive  {
2:     public static void main(String[] args)    {
3:         long antes, despues;
4:         CalcThread calc = new CalcThread();
5:         calc.start();
6:         int tope =((int)Math.floor(Math.random()*10000));
7:         for(int i=0; i < tope; i++); // espera
8:         antes = System.currentTimeMillis();
9:         while (calc.isAlive());
10:        despues = System.currentTimeMillis();
11:        System.out.println("Esperé␣"+(despues - antes)
12:                                    + "␣milisegundos");
13:        System.out.println("result␣vale␣" + calc.getResult());
14:    }
15: }
```

Que un proceso no esté ejecutándose no quiere decir que el proceso no esté vivo. Un proceso está vivo si está en espera, ya sea de alguna notificación, de que transcurra algún tiempo, o de que la JVM continúe ejecutándolo. Por lo tanto, aún cuando un proceso sea interrumpido por alguna de las condiciones que acabamos de mencionar, el proceso seguirá vivo mientras no termine su ejecución.

11.9 Terminación de la aplicación

Nos surge una pregunta natural respecto a la relación que existe entre los procesos lanzados desde una aplicación, y el proceso que corresponde a la aplicación misma: ¿qué pasa si el proceso principal termina antes de que terminen los procesos que fueron lanzados por él?

La respuesta es más simple de lo que se piensa: el proceso principal no termina hasta que hayan terminado todos los procesos que fueron lanzados por él. Bajo terminar nos referimos a dejar de estar vivo, no forzosamente a estar ejecutando algo. Esta situación, pensándolo bien, es lo natural. El proceso principal es aquel cuyo método main se invocó. El resto de los procesos fueron invocados utilizando start(). Esa es la única diferencia que hay entre ellos. Al lanzar procesos se genera una estructura como de cacto, donde cada proceso tiene su stack de ejecución, que es una continuación del stack de ejecución del proceso que lo lanzó. Por lo tanto,

no puede terminar un proceso hasta en tanto todos los procesos que dependen de él (que montaron su stack de ejecución encima) hayan terminado.

En Java hay otro tipo de procesos llamados *demonios*. Un *demonio* es un proceso lanzado desde otro y cuya tarea es realizar acciones que no forzosamente tienen mucho que ver con el proceso que los lanzó. En el caso de los demonios, cuando termina el proceso que los lanzó, en ese momento y abruptamente se interrumpen todos los demonios lanzados por ese proceso. Es como si aplicáramos un destroy() a todos los demonios lanzados por el proceso que termina.

Resumiendo, hay dos tipos de procesos que se pueden lanzar desde otro proceso cualquiera: los hilos de ejecución "normales" y los demonios. El proceso lanzador no puede terminar hasta en tanto los hilos de ejecución iniciados por él no terminen, mientras que los demonios se suspenden abruptamente cuando el proceso que los lanzó termina.

Un proceso se puede hacer demonio simplemente aplicándole el método setDaemon(true) al hilo de ejecución antes de que éste inicie su ejecución. Los procesos iniciados por un demonio son, a su vez, demonios. Veamos un ejemplo de la diferencia entre procesos normales y demonios en el listado 11.22.

Código 11.22 Diferencia entre procesos normales y demonios 1/2

```
1: import java.io.*;
2:
3: class Daemon extends Thread  {
4:     private static final int SIZE = 10;
5:     private Thread[] t = new Thread[SIZE];
6:
7:     public Daemon()     {
8:         setDaemon(true);
9:         start();
10:    }
11:
12:    public void run()     {
13:        for(int i = 0; i < SIZE; i++)
14:            t[i] = new DaemonSpawn(i);
15:        for(int i = 0; i < SIZE; i++)
16:            System.out.println("t[" + i +"].isDaemon()␣=␣" +
17:                                        t[i].isDaemon());
18:        while(true)
19:            yield();
20:    }
21: }
```

Código 11.22 Diferencia entre procesos normales y demonios 2/2

```
22: class DaemonSpawn extends Thread {
23:     public DaemonSpawn(int i)    {
24:         System.out.println("DaemonSpawn␣" + i + "␣lanzado");
25:         start();
26:     }
27:
28:     public void run()    {
29:         while(true) yield();
30:     }
31: }
32:
33:
34: public class Daemons   {
35:     public static void main(String[] args)}
36:                         throws IOException    {
37:         Thread d = new Daemon();
38:         System.out.println("d.isDaemon()␣=␣" + d.isDaemon());
39:     }
40: }
```

Para ver los distintos efectos que tiene el que los procesos lanzados sean o no demonios, vamos a ejecutar la clase Daemon como está. La teoría es que al terminar de ejecutarse el método main de esta clase, los demonios suspenderán también su funcionamiento. Y en efecto así es. La ejecución la podemos ver en la figura 11.10 en la página opuesta.

Mostramos dos ejecuciones de la misma aplicación. En la primera ejecución, se alcanzó a lanzar 5 demonios antes de que el proceso principal terminara. En la segunda ejecución, únicamente se alcanzaron a lanzar 2 demonios. El número de demonios que se logre lanzar dependerá de las políticas de atención de la JVM. Como el proceso Daemon es, a su vez, un demonio, en cuanto llega el proceso principal al final, se suspende abruptamente la ejecución de todos los demonios que se hayan iniciado desde él, o desde procesos o demonios iniciados por él.

Si cambiamos la línea 8: de esta aplicación para que Daemon sea un proceso común y corriente con la línea

```
8:   setDaemon(false);
```

obtenemos la salida que se muestra en la figura 11.10 en la página opuesta.

Figura 11.10 Terminación de procesos que son demonios

```
elisa@lambda ...ICC1/progs/threads % java Daemons
d.isDaemon() = true
DaemonSpawn 0 iniciado
DaemonSpawn 1 iniciado
DaemonSpawn 2 iniciado
DaemonSpawn 3 iniciado
DaemonSpawn 4 iniciado
elisa@lambda ...ICC1/progs/threads % java Daemons
d.isDaemon() = true
DaemonSpawn 0 iniciado
DaemonSpawn 1 iniciado
elisa@lambda ...ICC1/progs/threads %
```

Si además agregamos la línea que sigue

```
setDaemon(true);
```

entre las líneas 37: y 38: de la clase DaemonSpawn, la ejecución de Daemon tiene que terminar antes de que la aplicación pueda hacerlo, por lo que se alcanzan a lanzar todos los demonios. Como no hay ninguna manera de que termine el proceso Daemon, también el proceso principal nunca termina, hasta que tecleamos ctrl-C. Si en las líneas 18: y 19: de la clase Daemon quitamos el ciclo y únicamente tenemos yield(), el proceso termina normalmente en el momento en que termina el proceso de la clase Daemon. Veamos la salida en la figura 11.11 en la siguiente página.

Dependiendo de la programación particular que dé la JVM, podría suceder que alguno de los hilos de ejecución causara una excepción, o que al suspenderse alguno de los demonios abruptamente lanzara una excepción. Esto resulta en que ejecuciones de la misma aplicación puedan dar distintos resultados cuando estamos trabajando con hilos de ejecución, sobre todo si no se ejerce ninguna sincronización, como es el caso del ejemplo con el que estamos trabajando.

11.10 Depuración en hilos de ejecución

Java proporciona dos métodos a utilizar cuando hay problemas con la ejecución de coprocesos. Éstos son:

public String toString

Regresa la representación en cadena del hilo de ejecución. Esta representación tiene el nombre del hilo de ejecución, su prioridad y el grupo al que pertenece.

public static void dumpStack()

Imprime el stack de ejecución para el coproceso en cuestión. Manda la salida a System.out.

Figura 11.11 Terminación de procesos

```
elisa@lambda ...ICC1/progs/threads % java Daemons
d.isDaemon() = false
DaemonSpawn 0 iniciado
DaemonSpawn 1 iniciado
DaemonSpawn 2 iniciado
DaemonSpawn 3 iniciado
DaemonSpawn 4 iniciado
DaemonSpawn 5 iniciado
DaemonSpawn 6 iniciado
DaemonSpawn 7 iniciado
DaemonSpawn 8 iniciado
DaemonSpawn 9 iniciado
t[0].isDaemon() = true
t[1].isDaemon() = true
t[2].isDaemon() = true
t[3].isDaemon() = true
t[4].isDaemon() = true
t[5].isDaemon() = true
t[6].isDaemon() = true
t[7].isDaemon() = true
t[8].isDaemon() = true
t[9].isDaemon() = true
elisa@lambda ...ICC1/progs/threads
```

11.11 Otros temas relacionados con hilos de ejecución

Entre los temas que ya no veremos en este capítulo por considerarlos más apropiados para cursos posteriores, podemos mencionar:

Grupos: Cada hilo de ejecución pertenece a un grupo de hilos de ejecución con sus características particulares. Java proporciona casi un método para el trabajo en grupo por cada método para el trabajo con un hilo de ejecución. Los grupos pueden modificar la prioridad con la que se ejecuta un proceso, a qué otros hilos de ejecución tiene acceso, etc.

Excepciones en hilos de ejecución: Al igual que en una aplicación común y corriente, se pueden dar excepciones dentro de un hilo de ejecución. Si la excepción no es manejada a ese nivel, se propagará a niveles superiores. La gran diferencia es que una vez que hay una excepción en un hilo de ejecución, ese hilo se aborta y no se va a poder obtener información sobre él desde fuera, ya que ya no va a existir. Bajo este tema se revisan bien los mecanismos para poder saber con más certeza que está pasando y el por qué de las excepciones.

variables volatile: Esto se refiere a variables que para el compilador pudieran parecer como que no se modifican de un enunciado a otro, pero con la existencia de hilos de ejecución paralelos, pudieran ser modificadas por otros procesos. Se marcan las variables como `volatile` para obligar al compilador a consultar el valor de la variable, y no asumir que no ha cambiado desde la última vez que la vio; el efecto es que no permite al compilador hacer optimizaciones que pudieran falsear el verdadero valor de una variable en un momento dado.

Esperamos haber proporcionado una visión del potencial que tienen los hilos de ejecución en Java. Una vez que estén realizando programación en serio, y en el ambiente actual de procesos en red, tendrán que usarlos.

Bibliografía

[1] Ken Arnold and James Gosling. *The Java Programming Language Third Edition*. Addison-Wesley, 2001.

[2] José Galaviz Casas. *Elogio a la pereza*. Vínculos Matemáticos, Núm. 8, 2001.

[3] Sun Corporation. The source for java technology. web page.

[4] Elliotte Rusty Harold. *Java I/O*. O'Reilly, 1999.

[5] Henry F. Ledgard. *The Little Book of* Object-Oriented Programming. Prentice Hall, 1996.

[6] Canek Peláez and Elisa Viso. *Prácticas para el curso de Introducción a Ciencias de la Computación I*. Vínculos Matemáticos, por publicarse, 2002.

[7] Danny C. C. Poo and Dereck B. K. Kiong. *Object-Oriented Programming and Java*. Springer, 1998.

[8] Phil Sully. *Modeling the world with objects*. Prentice hall, 1993.

Esta obra terminó de imprimirse en
abril de 2007 en los talleres de
Publidisa Mexicana S. A. de C. V.
Calzada de Chabacano 69, planta alta.
México 06850, D. F.

El tiro fue de 500 ejemplares