

Empezar a programar usando Java



Natividad Prieto
Assumpció Casanova
Francisco Marqués
Marisa Llorens
Isabel Galiano
Jon Ander Gómez
Jorge González
Carlos Herrero
Carlos Martínez-Hinarejos
Germán Moltó
Javier Piris

Natividad Prieto (coordinadora)
Assumpció Casanova
Francisco Marqués
Marisa Llorens
Isabel Galiano
Jon Ander Gómez
Jorge González
Carlos Herrero
Carlos Martínez-Hinarejos
Germán Moltó
Javier Piris

EMPEZAR A PROGRAMAR USANDO JAVA

Primera edición, 2012 (versión impresa)
Primera edición, 2012 (versión electrónica)

- © de la presente edición:
Editorial Universitat Politècnica de València
www.editorial.upv.es
- © Todos los nombres comerciales, marcas o signos distintivos de cualquier clase contenidos en la obra están protegidos por la Ley.
<http://java.sun.com/docs/redist.html>
- © Natividad Prieto (coordinadora y autora)
Assumpció Casanova
Francisco Marqués
Marisa Llorens
Isabel Galiano
Jon Ander Gómez
Jorge González
Carlos Herrero
Carlos Martínez-Hinarejos
Germán Molto
Javier Piris
- © de las fotografías: su autor

ISBN: 978-84-8363-903-0 (versión impresa)
ISBN: 978-84-8363-935-1 (versión electrónica)

Queda prohibida la reproducción, distribución, comercialización, transformación, y en general, cualquier otra forma de explotación, por cualquier procedimiento, de todo o parte de los contenidos de esta obra sin autorización expresa y por escrito de sus autores.

Prólogo

El lector, o más bien usuario, de este libro tiene en sus manos el esfuerzo de un grupo de profesores con amplia experiencia universitaria en la docencia de asignaturas de introducción a la Programación y Estructuras de Datos. Los primeros pasos que dan los estudiantes en estas disciplinas deben estar cuidadosamente guiados para asegurar la atención en lo relevante y la construcción ordenada de los conocimientos, que posteriormente deben aplicar al desarrollo de programas. Otro proceder lleva a la confusión de ideas y a la incertidumbre en su aplicación, ya que las posibilidades que ofrecen los lenguajes de programación son tan amplias que su uso desordenado, o mal aprendido, genera importantes limitaciones en los futuros graduados.

Un nuevo libro de introducción a la Programación es un reto importante en la medida que se requiere seleccionar, ordenar, o crear contenidos propios de la enseñanza de esta materia, de modo que se facilite la capacidad de aprendizaje de los alumnos, a la vez que se cubran todos los objetivos. Todo ello añadiendo aportaciones originales que hagan verdaderamente útil este modo de plantear la enseñanza. Con estas premisas se ha elaborado este libro, dirigido a los profesores y estudiantes de los primeros cursos de Programación. El libro plantea el objetivo de enseñar a programar utilizando Java como lenguaje vehicular. Es cuidadoso en el equilibrio entre enseñar a pensar algoritmos y su correspondiente implementación en un lenguaje. Se ha procurado que la estructura del libro sea clara y con una ordenación de los contenidos que permite una sencilla utilización como libro de texto de una asignatura. Este enfoque, junto a los numerosos ejemplos ilustrativos, lo hacen ideal para su uso en los primeros cursos de la universidad.

No me queda más en este prólogo que agradecer a los autores el trabajo realizado y felicitarles por la capacidad de aunar, filtrar, o componer ideas, venciendo la dificultad que este proceso plantea cuando son varias las personas participantes en un proyecto. Por eso tiene más valor este trabajo que ha generado un texto homogéneo y claro, que seguro que servirá a muchos profesores y estudiantes para el aprendizaje de la Programación en los próximos años.

Emilio Sanchis Arnal
Catedrático de Lenguajes y Sistemas Informáticos
DSIC - UPV

Agradecimientos

Este libro compila una gran cantidad de material docente (apuntes, transparencias, código, ejercicios, etc.) desarrollado a lo largo de muchos años y planes de estudio por los profesores de las primeras asignaturas de Programación de los estudios de Informática de la Universitat Politècnica de València. Así que, de una forma u otra, en este libro se pueden reconocer no solo las aportaciones e ideas de sus autores, sino también las de los distintos compañeros que, durante ese tiempo, han compartido con nosotros la tarea docente de estas asignaturas: el uso del lenguaje de Programación, el enfoque y metodología expositiva seguida en sus temas, los ejercicios y ejemplos en él planteados, ... Por todo ello, los autores no podemos menos que agradecer a estos compañeros su inestimable ayuda y apoyo a la hora de plantear en este libro y en nuestro día a día docente la Programación como una actividad de resolución de problemas por ordenador.

Índice

Índice

Capítulo 1. Problemas, algoritmos y programas

1.1. Programas y la actividad de la programación	4
1.2. Lenguajes y modelos de programación	5
1.3. La programación orientada a objetos. El lenguaje Java	9
1.4. Un mismo ejemplo en diferentes lenguajes	11

Capítulo 2. Objetos, clases y programas

2.1. Estructura básica de una clase: atributos y métodos	20
2.2. Creación y uso de objetos: operadores <i>new</i> y <i>“.”</i>	23
2.3. La organización en paquetes del lenguaje Java	24
2.4. La herencia. Jerarquía de clases, la clase <i>Object</i>	26
2.5. Edición, compilación y ejecución en Java	28
2.5.1. Errores en los programas. Excepciones	29
2.6. Uso de comentarios. Documentación de programas	30
2.7. Problemas propuestos	34

Capítulo 3. Variables y asignación. Tipos de datos elementales. Bloques

3.1. Tipos de datos	39
3.2. Variables	40
3.3. Expresiones y asignación. Compatibilidad de tipos	42
3.4. Constantes. Modificador <i>final</i>	43
3.5. Algunas consideraciones sintácticas sobre identificadores	44
3.6. Tipos numéricos	45
3.6.1. Tipos enteros	45
3.6.2. Tipos reales	46
3.6.3. Compatibilidad y conversión de tipos	47
3.6.4. Operadores aritméticos	49
3.6.5. Desbordamiento	53
3.7. Tipo carácter	54
3.8. Tipo lógico	58
3.8.1. Operadores relacionales	59
3.8.2. Operadores lógicos	59
3.9. Precedencia de operadores	60
3.10. Bloques de instrucciones	61
3.11. Problemas propuestos	63

Capítulo 4. Tipos de datos: clases y referencias

4.1. Un nuevo ejemplo de definición de una clase	68
4.2. Inicialización de los atributos	70
4.3. Representación en memoria de los objetos. Variables referencia	71
4.3.1. Declaración de variables. Operador <i>new</i>	72
4.3.2. El operador de acceso <i>".</i>	74
4.3.3. La asignación	74

4.3.4. Copia de objetos	76
4.3.5. El operador == y el método <i>equals</i>	78
4.3.6. El <i>garbage collector</i>	79
4.4. Información de clase	79
4.5. Problemas propuestos	82

1er parcial hasta acá

Capítulo 5. Métodos

5.1. Definición y uso de métodos	86
5.1.1. Definición de métodos: métodos de clase y de objeto	86
5.1.2. Llamadas a métodos: perfil y sobrecarga	90
5.2. Declaración de métodos	94
5.2.1. Modificadores de visibilidad o acceso	94
5.2.2. Tipo de retorno. Instrucción <i>return</i>	95
5.2.3. Lista de parámetros	95
5.2.4. Cuerpo del método. Acceso a variables. Referencia <i>this</i>	96
5.3. Clases programa: el método <i>main</i>	100
5.4. Ejecución de una llamada	102
5.4.1. Registro de activación. Pila de llamadas	102
5.4.2. Paso de parámetros por valor	105
5.5. Clases Tipo de Dato	106
5.5.1. Funcionalidad básica de una clase	106
5.5.2. Sobreescritura de los métodos implementados en <i>Object</i>	109
5.6. Clases de utilidades	114
5.7. Documentación de métodos: <i>javadoc</i>	115
5.8. Problemas propuestos	120

Capítulo 6. Algunas clases predefinidas: String, Math. Clases envolventes

6.1. La clase <i>String</i>	127
6.1.1. Aspectos básicos	128
6.1.2. Concatenación	128
6.1.3. Formación de literales	130
6.1.4. Comparación	131
6.1.5. Algunos métodos	132
6.2. La clase <i>Math</i>	134
6.2.1. Constantes y métodos	134
6.2.2. Algunos ejemplos	137
6.3. Clases envolventes	139
6.4. Problemas propuestos	142

Capítulo 7. Entrada y salida elemental

7.1. Salida por pantalla	148
7.1.1. <i>System.out.println</i> y <i>System.out.print</i>	148
7.1.2. Salida formateada con <i>printf</i>	150
7.2. Entrada desde teclado	153
7.2.1. La clase <i>Scanner</i>	153
7.3. Problemas propuestos	161

Capítulo 8. Estructuras de control: selección

8.1. Instrucciones condicionales	165
8.1.1. Instrucción <i>if...else</i>	166
8.1.2. Instrucción <i>switch</i>	173
8.2. El operador ternario	178
8.3. Algunos ejemplos	179
8.4. Problemas propuestos	185

Capítulo 9. Estructuras de control: iteración

9.1. Iteraciones. El bucle <i>while</i>	195
9.2. Diseño de iteraciones	199
9.2.1. Estructura iterativa del problema	199
9.2.2. Terminación de la iteración	202
9.3. La instrucción <i>for</i>	206
9.4. La instrucción <i>do...while</i>	209
9.5. Algunos ejemplos	211
9.6. Problemas propuestos	214

Capítulo 10. Arrays: definición y aplicaciones

10.1. Arrays unidimensionales	222
10.1.1. Declaración y creación. Atributo <i>length</i>	222
10.1.2. Acceso a las componentes	225
10.1.3. Uso	226
10.2. Arrays multidimensionales	229
10.2.1. Declaración y creación	233
10.2.2. Acceso a las componentes	235
10.3. Tratamiento secuencial y directo de un array	237
10.3.1. Acceso secuencial: recorrido y búsqueda	237
10.3.2. Acceso directo	250
10.4. Representación de una secuencia de datos dinámica usando un array	254
10.5. Problemas propuestos	259

Capítulo 11. Recursión

11.1. Diseño de un método recursivo	273
11.2. Tipos de recursión	275
11.3. Recursividad y pila de llamadas	277
11.4. Algunos ejemplos	280
11.5. Recursión con arrays: recorrido y búsqueda	285
11.5.1. Esquemas recursivos de recorrido	287
11.5.2. Esquemas recursivos de búsqueda	292
11.6. Recursión versus iteración	295
11.7. Problemas propuestos	297

Capítulo 12. Análisis de algoritmos

12.1. Análisis de algoritmos	306
12.2. El coste temporal y espacial de los programas	307
12.2.1. El coste temporal medido en función de los tiempos de las operaciones elementales	308
12.2.2. El coste como una función del tamaño del problema. Talla del problema	310
12.2.3. Paso de programa. El coste temporal definido por conteo de pasos	311
12.3. Complejidad asintótica	313
12.3.1. Comparación de los costes de los algoritmos	314
12.3.2. Introducción a la notación asintótica	317
12.3.3. Algunas propiedades de los conjuntos Θ , O y Ω	319
12.3.4. La jerarquía de complejidades	320
12.3.5. Uso de la anotación asintótica	321
12.4. Análisis por casos	322
12.4.1. Caso mejor, caso peor y coste promedio	322
12.4.2. Ejemplos: algoritmos de recorrido y búsqueda	323

12.5. Análisis del coste de los algoritmos	324
12.6. Análisis del coste de los algoritmos iterativos	325
12.6.1. Otra unidad de medida temporal: la instrucción crítica	325
12.6.2. Eficiencia de los algoritmos de recorrido	325
12.6.3. Eficiencia de los algoritmos de búsqueda secuencial	326
12.6.4. Estudio del coste promedio del algoritmo de búsqueda secuencial	328
12.7. Análisis del coste de los algoritmos recursivos	328
12.7.1. Planteamiento de la función de coste. Ecuaciones de recurrencia	329
12.7.2. Resolución de las ecuaciones de recurrencia. Teoremas	332
12.7.3. Coste espacial de la recursión	335
12.8. Complejidad de algunos algoritmos numéricos recursivos	336
12.8.1. La multiplicación de números naturales	336
12.8.2. Exponenciación modular	340
12.9. Problemas propuestos	342

Capítulo 13. Ordenación y otros algoritmos sobre arrays

13.1. Selección directa	350
13.2. Inserción directa	353
13.3. Intercambio directo o algoritmo de la burbuja	355
13.4. Ordenación por mezcla o <i>mergesort</i>	357
13.5. Otros algoritmos sobre arrays	360
13.5.1. El algoritmo de mezcla natural	360
13.5.2. El algoritmo de búsqueda binaria	363
13.6. Problemas propuestos	367

Capítulo 14. Extensión del comportamiento de una clase. Herencia

14.1. Jerarquía de clases. Clases base y derivadas	372
14.2. Diseño de clases base y derivadas: <i>extends</i> , <i>protected</i> y <i>super</i>	374
14.3. Uso de una jerarquía de clases. Polimorfismo	380
14.3.1. Tipos estáticos y dinámicos	382
14.3.2. Ejemplo de uso del polimorfismo	383
14.4. Más herencia en Java: control de la sobreescritura	389
14.4.1. Métodos y clases finales	389
14.4.2. Métodos y clases abstractos	390
14.4.3. Interfaces y herencia múltiple	392
14.5. Organización de las clases en Java	393
14.5.1. La librería de clases del Java	393
14.5.2. Uso de <i>packages</i>	395
14.6. Problemas propuestos	397

Capítulo 15. Tratamiento de errores

15.1. Fallos de ejecución y su modelo Java	404
15.1.1. La jerarquía <i>Throwable</i>	404
15.1.2. Ampliación de la jerarquía <i>Throwable</i> con excepciones de usuario	410
15.2. Tratamiento de excepciones	411
15.2.1. Captura de excepciones: <i>try/catch/finally</i>	412
15.2.2. Propagación de excepciones: <i>throw</i> versus <i>throws</i>	415
15.2.3. Excepciones <i>checked/unchecked</i>	418
15.3. Problemas propuestos	419

Capítulo 16. Entrada y salida: ficheros y flujos

16.1. La clase <i>File</i>	427
16.2. Ficheros de texto	430
16.2.1. Escritura en un fichero de texto	430
16.2.2. Lectura de un fichero de texto	431
16.3. Ficheros binarios	436
16.3.1. Escritura en un fichero binario	436
16.3.2. Lectura de un fichero binario	437
16.3.3. Ficheros binarios de acceso aleatorio	439
16.4. Flujos	441
16.4.1. Flujos de bytes	442
16.4.2. Flujos de caracteres	444
16.5. E/S de objetos	444
16.6. Excepción <i>E0FException</i> . Determinación del final de un fichero binario	450
16.7. Problemas propuestos	453

Capítulo 17. Tipos lineales. Estructuras enlazadas

17.1. Representación enlazada de secuencias	462
17.1.1. Definición recursiva de secuencias. La clase <i>Nodo</i>	462
17.1.2. Recorrido y búsqueda en secuencias enlazadas	468
17.1.3. Inserción y borrado en secuencias enlazadas	471
17.2. Tipos lineales	477
17.2.1. Pilas	477
17.2.2. Colas	483
17.2.3. Listas con punto de interés	488
17.3. Problemas propuestos	497

Bibliografía

Índice de Figuras

Índice de Tablas



**Contenidos complementarios
(ejercicios, etc.)**

Capítulo 1

Problemas, algoritmos y programas

Los conceptos que se desarrollarán a continuación son fundamentales en la mecanización del cálculo, objetivo de gran importancia en el desarrollo cultural humano que, además, ha adquirido una relevancia extraordinaria con la aparición y posterior universalización de los computadores. Problemas, algoritmos y programas forman el tronco principal en que se fundamentan los estudios de computación.

Dado un *problema* P , un *algoritmo* A es un conjunto de reglas, o instrucciones, que definen cómo resolver P en un tiempo finito.

Aunque “cambiar una rueda pinchada a un coche” es un problema que incluso puede estudiarse y resolverse en el ámbito informático, no es el tipo de problema que habitualmente se resuelve utilizando un computador. Por su misma estructura, y por las unidades de entrada/salida que utilizan, los ordenadores están especializados en el tratamiento de secuencias de información (codificada) como, por ejemplo, series de números, de caracteres, de puntos de una imagen, muestras de una señal, etc.

Ejemplos más habituales de las clases de problemas que se plantearán en el ámbito de la programación a pequeña escala y, por lo tanto, en el de este libro, se pueden encontrar en el campo del cálculo numérico, del tratamiento de palabras y de la representación gráfica, entre muchos otros. Algunos ejemplos de ese tipo de problemas son los siguientes:

- Determinar el producto de dos números multidígito a y b .
- Determinar la raíz cuadrada positiva del número 2.
- Determinar la raíz cuadrada positiva de un número n cualquiera.
- Determinar si el número n , entero mayor que 1, es primo.

- Dada la lista de palabras l , determinar las palabras repetidas.
- Determinar si la palabra p es del idioma castellano.
- Separar siládicamente la palabra p .
- Ordenar y listar alfabéticamente todas las palabras del castellano.
- Dibujar en la pantalla del ordenador un círculo de radio r .

Como se puede observar, en la mayoría de las ocasiones, los problemas se definen de forma general, haciendo uso de identificadores o *parámetros* (en los ejemplos esto es así excepto en el segundo problema, que es un caso particular del tercero). Las soluciones proporcionadas a esos problemas (algoritmos) tendrán también esa característica.

A veces los problemas están definidos de forma imprecisa puesto que los seres humanos podemos, o bien recabar nueva información sobre ellos, o bien realizar presunciones sobre los mismos. Cuando un problema se encuentra definido de forma imprecisa introduce una ambigüedad indeseable, por ello, siempre que esto ocurra, se deberá precisar el problema, eliminando en lo posible su ambigüedad. Así, por ejemplo, cuando en el problema tercero se desea determinar la raíz cuadrada positiva de un número n , se puede presuponer que dicho número n es real y no negativo, por ello, redefiniremos el problema del modo siguiente: determinar la raíz cuadrada positiva de un número n , entero no negativo, cualquiera.

Ejemplos de algoritmos pueden encontrarse en las secuencias de reglas aprendidas en nuestra niñez, mediante las cuales realizamos operaciones básicas de números multidígito como, por ejemplo, sumas, restas, productos y divisiones. Son algoritmos ya que definen de forma precisa la resolución en tiempo finito de un problema de índole general.

Como ejemplos adicionales, se muestran a continuación algunos algoritmos para la solución de problemas de la lista anterior:

Ejemplo 1.1. Considérese el problema: ¿es n , entero mayor que uno, un número primo?

Como se recordará un número primo es aquel que sólo es divisible por el mismo o por la unidad. Los siguientes son posibles algoritmos para resolver este problema:

El primer algoritmo (que se muestra en la figura 1.1) consiste en la descripción de una enumeración de los números anteriores a n comprobando, para cada uno, la divisibilidad del propio n por el número considerado.

El algoritmo siguiente, en la figura 1.2, es similar al anterior, ya que la secuencia de cálculos que define para resolver el problema es idéntica a la expresada por

Algoritmo 1.-

Considerar todos los números comprendidos entre 2 y n (excluido).

Para cada número de dicha sucesión comprobar si dicho número divide al número n .

Si ningún número divide a n , entonces n es primo.

Figura 1.1: Algoritmo 1 para determinar si n es primo.

el algoritmo primero; sin embargo, se ha escrito utilizando una notación algo más detallada, en la que se han hecho explícitos, enumerándolos, los pasos que se siguen y permitiendo con ello la referencia a un paso determinado del propio algoritmo.

Algoritmo 2.- Seguir los pasos siguientes en orden ascendente:

Paso 1. Sea i un número entero de valor igual a 2.

Paso 2. Si i es igual a n parar, n es primo.

Paso 3. Comprobar si i divide a n , entonces parar, n no es primo.

Paso 4. Reemplazar el valor de i por $i+1$, volver al Paso 2.

Figura 1.2: Algoritmo 2 para determinar si n es primo.

El tercer algoritmo, en la figura 1.3, mantiene una estrategia similar a la utilizada por los dos primeros: comprobaciones sucesivas de divisibilidad por números anteriores; sin embargo, haciendo uso de propiedades básicas de los números, mejora a los algoritmos anteriores al reducir mucho la cantidad de comprobaciones de divisibilidad efectuadas.

Algoritmo 3.- Seguir los pasos siguientes en orden ascendente:

Paso 1. Si n vale 2 entonces parar, n es primo.

Paso 2. Si n es múltiplo de 2 acabar, n no es primo.

Paso 3. Sea i un número entero de valor igual a 3.

Paso 4. Si i es mayor que la raíz cuadrada positiva de n parar, n es primo.

Paso 5. Comprobar si i divide a n , entonces parar, n no es primo.

Paso 6. Reemplazar el valor de i por $i+2$, volver al Paso 4.

Figura 1.3: Algoritmo 3 para determinar si n es primo.

Ejemplo 1.2. Considérese el problema de encontrar las palabras repetidas de cierta lista l ; dos posibles algoritmos para resolver el problema aparecen a continuación en la figura 1.4. En este caso, se puede considerar que el primer algoritmo es mejor (*más eficiente*) que el segundo por que, en condiciones normales, su ejecución supondrá un menor número de operaciones de comparación.

Algoritmo 1.-

- Ordenar la lista alfabéticamente.
- Recorrer la lista,
 - si dos elementos consecutivos son iguales, entonces estaban repetidos, escribirlos.

Algoritmo 2.-

- Recorrer la lista,
 - para cada elemento comprobar (recorriendo de nuevo la lista)
 - si está repetido y entonces escribirlo.

Figura 1.4: Dos algoritmos que permiten escribir los elementos repetidos de una lista.

En cualquier caso, como es fácil ver, la descripción o nivel de detalle de la solución de un problema en términos algorítmicos depende de qué o quién debe entenderlo, resolverlo e interpretarlo.

Para facilitar la discusión se introduce el término genérico *procesador*. Se denomina *procesador* a cualquier entidad capaz de interpretar y ejecutar un cierto repertorio de instrucciones.

Un *programa* es un algoritmo escrito con una notación precisa para que pueda ser ejecutado por un procesador. Habitualmente, los procesadores que se utilizarán serán computadores con otros programas para facilitar el manejo de la máquina subyacente.

Cada instrucción al ejecutarse en el procesador supone cierto cambio o transformación, de duración finita, y de resultados definidos y predecibles. Dicho cambio se produce en los valores de los elementos que manipula el programa. En un instante dado el conjunto de dichos valores se denomina el *estado del programa*.

Denominamos *cómputo* a la transformación de estado que tiene lugar al ejecutarse una o varias instrucciones de un programa.

1.1 Programas y la actividad de la programación

Como se ve, un programa es la definición precisa de una tarea de computación; siendo el propósito de un programa su ejecución en un procesador; y suponiendo dicha ejecución cierto cómputo o transformación.

Para poder escribir programas de forma precisa y no ambigua es necesario definir reglas que determinen tanto lo que se puede escribir en un programa (y el procesador podrá interpretar) como el resultado de la ejecución de dicho programa por el procesador. Dicha notación, conjunto de reglas y definiciones, es lo que se deno-

mina un *lenguaje de programación*. Más adelante se estudiarán las características de algunos de ellos.

Como es lógico, el propósito principal de la programación consiste en describir la solución computacional (eficiente) de clases de problemas. Aunque hay que destacar que se ha demostrado la existencia de problemas para los que no puede existir solución computacional alguna, lo que implica una limitación importante a las posibilidades de la mecanización del cálculo.

Adicionalmente, los programas son objetos complejos que habitualmente necesitan modificaciones y adaptaciones. De esta complejidad es posible hacerse una idea si se piensa que algunos programas (la antigua iniciativa de defensa estratégica de los EEUU, por ejemplo) pueden contener millones de líneas y que, por otro lado, un error en un único carácter de una sola línea puede suponer el malfuncionamiento de un programa (así, por ejemplo, el Apollo XIII tuvo que cancelar, durante el trayecto, una misión a la luna debido a que en un programa se había sustituido erróneamente una coma por un punto decimal, o al telescopio espacial Hubble se le corrigió de forma indebida las aberraciones de su espejo, al cambiarse en un programa un símbolo + por un -, con lo que el telescopio acabó "miope" y, por ello, inutilizable durante un periodo de tiempo considerable).

Debido a la complejidad mencionada, se considera que los programas tienen un ciclo de existencia que está formado, a grandes rasgos, por las dos etapas siguientes:

- *Desarrollo*: creación inicial y validación del programa.
- *Mantenimiento*: correcciones y cambios posteriores al desarrollo.

También se puede establecer la siguiente subdivisión en función de la envergadura del problema a resolver (y del tamaño del programa necesario para resolverlo):

- *Programación a pequeña escala*: número reducido de líneas de programa, intervención de una sola persona, por ejemplo: un programa de ordenación.
- *Programación a gran escala*: muchas líneas de programa, equipo de programadores, por ejemplo: desarrollo de un sistema operativo.

1.2 Lenguajes y modelos de programación

Los orígenes de los lenguajes de programación se encuentran en las máquinas. La llamada máquina original de Von Neumann se diseñó a finales de los años 1940 en Princeton (aunque su diseño coincide en gran medida con el de la máquina creada con elementos exclusivamente mecánicos por Charles Babbage y programada por Ada Byron en Londres hacia 1880).

La mayoría de los ordenadores modernos tienen tanto en común con la máquina original de Von Neumann que se les denomina precisamente máquinas con arquitectura “Von Neumann”.

La característica fundamental de dicha arquitectura es la *banalización de la memoria*, esto es, la existencia de un espacio de memoria único y direccionable individualmente, que sirve para mantener tanto datos como instrucciones; existiendo unidades especializadas para el tratamiento de los datos, Unidad Aritmético Lógica (ALU) y de las instrucciones, Unidad de Control (UC). Ésta es también, a grandes rasgos, la estructura del procesador central de casi cualquier computador moderno significativo. Véase la figura 1.5, en la que se puede observar que en el mismo espacio de memoria coexisten tanto datos como instrucciones para la manipulación de los mismos.

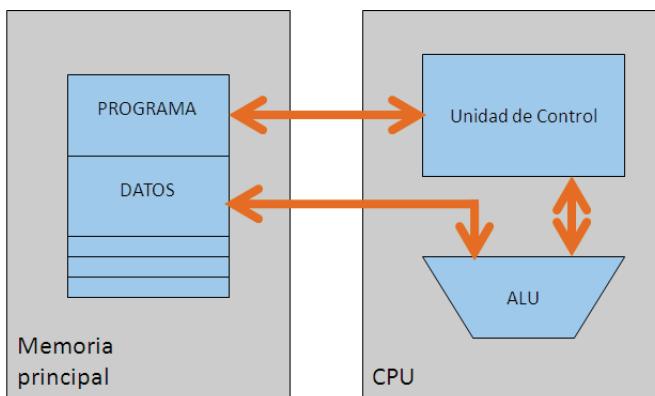


Figura 1.5: Estructura de un procesador con arquitectura Von Neumann.

Al nivel de la máquina, un programa es una sucesión de palabras (compuestas de bits), habitualmente en posiciones consecutivas de memoria que representan instrucciones o datos. El lenguaje con el que se expresa es el *lenguaje máquina*.

Por ejemplo, el fragmento siguiente, muestra en su parte derecha una secuencia de código en lenguaje máquina.

Instrucciones en ensamblador y código máquina		
Load 24,	# a está en la dir. 24h	10111100 00100100
Multiply 33,	# mult. por b en la dir. 33h	10111111 00110011
Store 3C,	# almacenar en c en la dir. 3Ch	11001110 00111100

Obviamente, los programas en lenguaje máquina son ininteligibles, tal y como puede verse en el ejemplo.

Aunque no tanto, también son muy difíciles de entender los denominados *lenguajes ensambladores* (fragmento anterior, columna primera a la izquierda) en los que ya se utilizan mnémicos e identificadores para las instrucciones y datos.

Estos lenguajes se conocen como de *bajo nivel*. Los problemas principales de dichos lenguajes son el bajo nivel de las operaciones que aportan, así como la posibilidad de efectuar todo tipo de operaciones (de entre las posibles) sobre los datos que manipulan. Así, por ejemplo, es habitual disponer tan solo de operaciones de carácter aritmético, de comparación y de desplazamiento, ello permite interpretar cualquier posición de memoria exclusivamente como un número. Un carácter se representará mediante un código numérico, aunque será visto a nivel máquina como un número (con lo que pueden multiplicarse entre sí, por ejemplo, dos caracteres, lo que posiblemente no tiene sentido).

Hacia finales de la década de los años 50 aparecieron lenguajes de programación orientados a hacer los programas más potentes, inteligibles y seguros; estos lenguajes serían denominados, en contraposición a los anteriores, *lenguajes de alto nivel*. En ellos, un segmento como el anterior, para multiplicar ciertos valores *a* y *b*, dando como resultado *c*, podría ser simplemente:

```
c = a * b;
```

que, además de más legible, es bastante más seguro puesto que implica que para poderse ejecutar, típicamente se comprueba que los datos implicados deben de ser numéricos. Por ejemplo, si *a*, *b* o *c* se hubiesen definido previamente como caracteres, la operación anterior puede no tener sentido y el programa detenerse antes de su ejecución, advirtiendo de ello al programador, que podrá subsanar el error.

Así, por ejemplo, la motivación fundamental del primer lenguaje de alto nivel, el FORTRAN (FORmula TRANslator), desarrollado en 1957, era la de disponer de un lenguaje conciso para poder escribir programas de índole numérica y traducirlos automáticamente a lenguaje máquina.

Esta forma de trabajo es la utilizada hoy en día de forma habitual. A los programas que traducen las instrucciones de un lenguaje de alto nivel a un lenguaje máquina se les denomina *compiladores*, siendo el proceso seguido para poder traducir y ejecutar un programa en un lenguaje determinado el que se muestra en la figura 1.6.

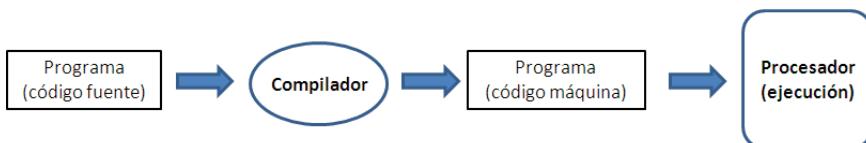


Figura 1.6: Proceso de compilación y ejecución de un programa.

Otros lenguajes de programación que aparecieron en la década de los 60, poco tiempo después del FORTRAN son el APL, el Algol, el Cobol, el LISP, el Basic y el PL1.

Algunas características comunes a todos ellos y, en general, a todos los lenguajes de alto nivel son:

- Tienen operadores y estructuras más cercanas a las utilizadas por las personas.
- Son más seguros que el código máquina y protegen de errores evidentes.
- El código que proporcionan es transportable y, por lo tanto, independiente de la máquina en que se tenga que ejecutar.
- El código que proporcionan es más legible.

En la década de los 70, como reacción a la falta de claridad y de estructuración introducida en los programas por los abusos que permitían los primeros lenguajes de programación, se originó la, así denominada, *programación estructurada*, que consiste en el uso de un conjunto de modos de declaración y constructores en los lenguajes, reducido para que sea fácilmente abarcable y, al mismo tiempo, suficiente para expresar la solución algorítmica de cualquier problema resoluble.

Ejemplos de dichos lenguajes son los conocidos Pascal, C y Módula-2.

El modelo introducido por la *programación estructurada* tiene aún hoy en día una gran importancia para el desarrollo de programas. De hecho, se asumirá de forma implícita a lo largo del libro aunque, como se verá, enmarcándolo dentro de la programación orientada a objetos.

Otro aspecto significativo de los lenguajes de programación de alto nivel que hay que destacar es el de que los mismos representan un procesador o *máquina extendida*: esto es, aquélla que puede ejecutar las instrucciones de dicho lenguaje. Consideraremos, en general, que un lenguaje de programación es una extensión de la máquina en que se apoya, del mismo modo que un programa es una extensión del lenguaje de programación en que se construye.

Un lenguaje de programación proporciona un *modelo de computación* que no tiene por qué ser igual al de la máquina que lo sustenta, pudiendo ser de hecho completamente diferente. Por ejemplo, un lenguaje puede hacer parecer que un programa se está ejecutando en varias máquinas distintas, aun cuando sólo existe una; o, por el contrario, puede hacer parecer que se está ejecutando en una sola máquina (muy rápidamente) cuando realmente ha subdividido la computación que realiza entre varias máquinas diferentes.

A lo largo de la historia los seres humanos hemos desarrollado varios modelos de computación posibles (unos basados en una máquina universal, otros en las funciones recursivas, otros en la noción de inferencia, etc). Se ha demostrado que todos estos modelos son computacionalmente equivalentes, esto es: si existe una solución algorítmica para un problema utilizando uno de los modelos, también existe una solución utilizando cualquiera de los otros.

El modelo más extendido de computación hace uso de una máquina universal bastante similar en su esencia a los procesadores actuales denominada, en honor a su inventor, *Máquina de Turing*. En este modelo, una computación es una transformación de estados y un programa representa una sucesión de computaciones, o transformaciones, del estado inicial del problema al final o solución del mismo. Este modelo es el que seguiremos a lo largo del presente libro. En él, la solución de un problema se define dando una secuencia de pasos que indican la secuencia de computaciones para resolverlo. Este modelo de programación recibe el nombre de *modelo* o *paradigma imperativo*.

Diagramas y listas bastante completos con la evolución de los lenguajes, pueden encontrarse, si se efectúa una búsqueda, en muchas URLs; entre ellas:

<http://www.levenez.com/lang/>

<http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>

1.3 La programación orientada a objetos. El lenguaje Java

Aunque la *programación orientada a objetos* tuvo sus inicios en la década de los 70, es sólo más recientemente cuando ha adquirido relevancia, siendo en la actualidad uno de los modelos de desarrollo de programas predominante. Así, presenta mejoras para el desarrollo de programas en comparación a lo que aporta la programación estructurada que, como se ha mencionado, fue el modelo de desarrollo fundamental durante la década de los 70.

El elemento central de un programa orientado a objetos es la *clase*. Una clase determina completamente el comportamiento y las características propias de sus componentes. A los casos particulares de una clase se les denomina *objetos*. Un programa se entiende como un conjunto de objetos que interactúan entre sí.

Una de las principales ventajas de la programación orientada a objetos es que facilita la reutilización del código ya realizado (*reusabilidad*), al tiempo que permite ocultar detalles (*ocultación*) no relevantes (*abstracción*), aspectos fundamentales en la gestión de proyectos de programación complejos.

El lenguaje Java (1991) es un lenguaje orientado a objetos, de aparición relativamente reciente. En ese sentido, un programa en Java consta de una o más clases

interdependientes. Las clases permiten describir las propiedades y habilidades de los objetos de la vida real con los que el programa tiene que tratar.

El lenguaje Java presenta, además, algunas características que lo diferencian, a veces significativamente, de otros lenguajes. En particular está diseñado para facilitar el trabajo en la WWW, mediante el uso de los programas navegadores de uso completamente difundido hoy en día. Los programas de Java que se ejecutan a través de la red se denominan *applets* (aplicación pequeña).

Otras de sus características son: la inclusión en el lenguaje de un entorno para la programación gráfica (**AWT** y **Swing**) y el hecho de que su ejecución es *independiente de la plataforma*, lo que significa que un mismo programa se ejecutará exactamente igual en diferentes sistemas.

Para la consecución de las características anteriores, el Java hace uso de lo que se denomina *Máquina Virtual Java* (Java Virtual Machine, *JVM*). La *JVM* es una extensión (mediante un programa) del sistema real en el que se trabaja, que permite ejecutar el código resultante de un programa Java ya compilado independientemente de la plataforma en que se esté utilizando. En particular, todo navegador dispone (o puede disponer) de una *JVM*; de ahí la universalidad de su uso.

El procedimiento necesario para la ejecución un programa en Java puede verse, de forma resumida, en la figura 1.7.

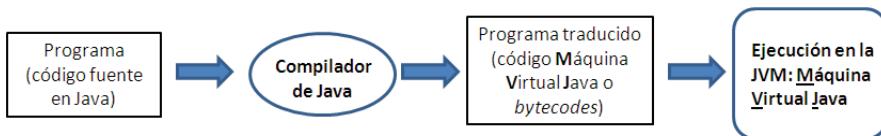


Figura 1.7: Proceso de compilación y ejecución de un programa en Java.

Es interesante comparar dicho proceso con el que aparece en la figura 1.6, donde se muestra un proceso similar pero para un programa escrito en otros lenguajes de programación. La diferencia, como puede observarse, consiste en el uso de la, ya mencionada, máquina virtual, en el caso del Java (*JVM*).

Una de las ventajas de este modelo, es que permite utilizar el mismo código Java virtual, ya compilado, siempre que en el sistema se disponga de una máquina virtual Java.

Uno de los inconvenientes de un modelo así, estriba en que puede penalizar el tiempo de ejecución del programa final ya que introduce un elemento intermedio, la máquina virtual, para permitir la ejecución.

1.4 Un mismo ejemplo en diferentes lenguajes

Como ejemplo final de este capítulo, se muestra a continuación el algoritmo ya visto para determinar si un número n entero y positivo es o no un número primo (Algoritmo 3, figura 1.3), implementado en diferentes lenguajes de programación:

- Pascal, en la figura 1.8.
- C/C++, en la figura 1.9.
- Python, en la figura 1.10.
- Java, en la figura 1.11.
- C#, en la figura 1.12.

La similitud que se puede observar en los ejemplos, entre los distintos lenguajes, se debe principalmente a que en la evolución de los mismos, muchos de ellos heredan, mejorándolas, características de los lenguajes anteriores.

En particular, el lenguaje C++ es una ampliación del C hacia la Programación orientada a objetos, mientras que el Java es una evolución de los dos anteriores, que presenta mejoras con respecto a ellos en cuanto a la gestión de la memoria, así como un modelo de ejecución, diferente, basado, como ya se ha mencionado, en una máquina virtual.

También están basados en un modelo de máquina virtual el C# y el Python. Se puede decir que el C# es un heredero directo del Java; mientras que el Python, aunque toma características de los anteriores, presenta también bastantes elementos innovadores.

```
function es_primo(n:integer):boolean;
(* determina si n, entero mayor que uno, es un número primo *)
var i,integer; primo:boolean; raiz:real;
begin
  if n = 2 then primo:=true
  else if n mod 2 = 0 then primo:=false
  else begin
    primo:=true;
    i:=3; raiz:=sqrt(n);
    while (i<=raiz) and primo do
    begin
      primo:=((n mod i) <> 0);
      i:=i+2;
    end;
  end;
  es_primo:= primo;
end;
```

Figura 1.8: ¿Es n primo? Algoritmo 3, versión en Pascal.

```
int es_primo(int n) {
    /* determina si n, entero mayor que uno, es un número primo */
    int i, primo; float raiz;
    if (n==2) primo = 0;
    else if (n%2) primo = 1;
    else {
        i = 3; raiz = sqrt(n);
        while ((i<=raiz) && !(n%i)) {i += 2;}
        primo = !(n%i);
    }
    return primo;
}
```

Figura 1.9: ¿Es n primo? Algoritmo 3, versión en C/C++.

```
from math import sqrt
def es_primo(n):
    # determina si n, entero mayor que uno, es un número primo
    if n==2: primo = True
    elif n%2==0: primo = False
    else:
        i = 3
        raíz = sqrt(n)
        while i<=raíz and n%i!=0: i += 2
        primo = (n%i!=0)
    return primo
```

Figura 1.10: ¿Es n primo? Algoritmo 3, versión en Python.

```
/**
 * Determina si n, entero mayor que uno, es un número primo
 */
static boolean es_primo(int n) {
    int i; double raíz; boolean primo;
    if (n==2) primo = true;
    else if (n%2==0) primo = false;
    else {
        i = 3; raíz = Math.sqrt(n);
        while ((i<=raíz) && (n%i!=0)) {i += 2;}
        primo = (n%i!=0);
    }
    return primo;
}
```

Figura 1.11: ¿Es n primo? Algoritmo 3, versión en Java.

```
/* Determina si n, entero mayor que uno, es un número primo */
static bool es_primo(int n) {
    int i; double raíz; bool primo;
    if (n==2) primo = true;
    else if (n%2==0) primo = false;
    else {
        i = 3; raíz = Math.Sqrt(n);
        while ((i<=raíz) && (n%i!=0)) {i += 2;}
        primo = (n%i!=0);
    }
    return primo;
}
```

Figura 1.12: ¿Es n primo? Algoritmo 3, versión en C#.

Más información

[Pyl75] Z.W. (selec.) Pylyshyn. *Perspectivas de la revolución de los computadores/Selec., comentarios e introd. de Z.W. Pylyshyn; tr. por Luis García Llorente; rev. de Eva Sánchez.* Alianza, 1975. Incluye textos de H. Aiken, Ch. Babbage, J. von Neumann, C. Shannon, A.M. Turing y otros.

[Tra77] B.A. Trajtenbrot. *Los algoritmos y la resolución automática de problemas.* MIR, 1977.

Capítulo 2

Objetos, clases y programas

La *programación orientada a objetos* (POO) es el modelo de construcción de programas predominante en la actualidad debido a que presenta un sistema basado fuertemente en la representación de la realidad y que, al mismo tiempo, refuerza el uso de buenos criterios aplicables al desarrollo de programas, como son la *abstracción*, la *ocultación de información* y la *reusabilidad*, entre otros.

El objetivo de este capítulo es introducir de manera superficial las nociones básicas de la POO. Un estudio en profundidad de todas ellas se abordará en el resto de capítulos del libro.

El elemento fundamental en la POO es, por supuesto, el *objeto*. Un *objeto* se puede definir como una agrupación o colección de datos y operaciones que poseen determinada estructura y mediante los cuales se modelan aspectos relevantes de un problema.

Los objetos que comparten cierto comportamiento se pueden agrupar en diferentes categorías llamadas *clases*. Una *clase* es, por lo tanto, una descripción de cuál es el comportamiento de cada uno de los objetos de la clase. Se dice entonces que el *objeto* es una *instancia* de la *clase*.

El lenguaje Java es un lenguaje orientado a objetos, por lo que se puede decir que programar en Java consiste en *escribir las definiciones de las clases y utilizar esas clases para crear objetos* de forma que, mediante los mismos, se represente adecuadamente el problema que se desea resolver.

El lenguaje Java posee un gran número de clases predefinidas, por lo que no es necesario reinventarlas, basta con utilizarlas cuando se necesiten.

Atendiendo a la estructura de la clase y al uso que se va a hacer de ella se pueden distinguir tres tipos básicos de clases:

Clase Tipo de Dato: es aquélla que define el conjunto de valores posibles que pueden tomar los objetos y las operaciones que sobre estos objetos se pueden realizar.

Clase Programa: son éstas las que realmente inician la ejecución del código.

Clase de Utilidades: es un repositorio de operaciones que pueden utilizarse desde otras clases.

En la figura 2.1 se muestra, como ejemplo, el código completo en Java de la clase **Círculo**.

Antes de estudiarlo con detalle, conviene señalar que todas las líneas precedidas por los símbolos `//` o encerradas en un bloque `/** ... */` son consideradas por el Java como comentarios, cuyo único fin es documentar la clase. Estos comentarios no afectan al comportamiento (o ejecución) de dicha clase.

En esencia, la clase que se presenta define el tipo de datos **Círculo** que tiene un radio que es un número real, un color que se expresa con su nombre (verde, rojo, etc.) y cierta posición de su centro, representada por sus coordenadas *x* e *y* que son dos números enteros (véase las líneas 8 y 9); asimismo, define que sobre un **Círculo** sólo se pueden hacer las operaciones que se describen en la clase como, por ejemplo, consultar su radio (`getRadio` en línea 19), modificarlo (`setRadio` en línea 28), calcular su área (`área` en línea 41), etc.

La clase gráfica **Pizarra** es otro ejemplo de Clase Tipo de Dato; como el código asociado a esta clase queda fuera de los propósitos del libro, en la figura 2.2 se muestra únicamente la documentación esencial asociada a esta clase.

Sirva este ejemplo para destacar la importancia de documentar de forma correcta las clases que se diseñan. Nótese que para saber utilizar esta clase basta con su documentación; no es necesario conocer los detalles de cómo esta implementada. Dada la importancia de este tema, será tratado con más detalle en un apartado posterior.

En concreto, de la documentación de la clase **Pizarra** se tiene que una **Pizarra** se puede construir (véase la parte denominada **Constructor Summary**) dándole, si se desea, un título y un tamaño inicial. Además, sobre objetos de tipo **Pizarra**, se pueden realizar dos operaciones (véase la parte **Method Summary**): `add` para añadir un objeto gráfico, dibujándolo, como por ejemplo un **Círculo**, y `dibujaTodo` para dibujar todos los elementos gráficos que se hayan añadido hasta el momento a la **Pizarra**. Adicionalmente, un programa puede utilizar tantos objetos de tipo **Pizarra** como se deseé.

```

1  /**
2  * Clase Circulo: define un círculo de un determinado radio, color y
3  * posición de su centro, con la funcionalidad que aparece a continuación.
4  * @author Libro IIP-PRG
5  * @version 2011
6  */
7 public class Circulo {
8     private double radio; private String color;
9     private int centroX, centroY;
10
11    /** crea un Circulo de radio 50, negro y centro en (100,100). */
12    public Circulo() {
13        radio = 50; color = "negro"; centroX = 100; centroY = 100; }
14    /** crea un Circulo de radio r, color c y centro en (px,py). */
15    public Circulo(double r, String c, int px, int py) {
16        radio = r; color = c; centroX = px; centroY = py; }
17
18    /** consulta el radio del Circulo. */
19    public double getRadio() { return radio; }
20    /** consulta el color del Circulo. */
21    public String getColor() { return color; }
22    /** consulta la abscisa del centro del Circulo. */
23    public int getCentroX() { return centroX; }
24    /** consulta la ordenada del centro del Circulo. */
25    public int getCentroY() { return centroY; }
26
27    /** actualiza el radio del Circulo a nuevoRadio. */
28    public void setRadio(double nuevoRadio) { radio = nuevoRadio; }
29    /** actualiza el color del Circulo a nuevoColor. */
30    public void setColor(String nuevoColor) { color = nuevoColor; }
31    /** actualiza el centro del Circulo a la posición (px,py). */
32    public void setCentro(int px, int py) { centroX=px; centroY=py; }
33    /** desplaza un poco a la derecha el Circulo. */
34    public void aLaDerecha() { centroX += 10; }
35    /** incrementa el radio del Circulo. */
36    public void crece() { radio = radio * 1.3; }
37    /** decrementa el radio del Circulo. */
38    public void decrece() { radio = radio / 1.3; }
39
40    /** calcula el área del Circulo. */
41    public double area() { return 3.14 * radio * radio; }
42    /** calcula el perímetro del Circulo. */
43    public double perimetro() { return 2 * 3.14 * radio; }
44
45    /** obtiene un String con las componentes del Circulo. */
46    public String toString() { String res = "Circulo de radio "+radio;
47        res += ", color "+color+" y centro ("+centroX+","+centroY+"); "
48        return res; }
49}

```

Figura 2.1: Clase Circulo.

Class Pizarra

```
java.lang.Object
└ java.awt.Component
   └ java.awt.Container
      └ java.awt.Window
         └ java.awt.Frame
            └ javax.swing.JFrame
               └ Pizarra
```

```
public class Pizarra
extends JFrame
```

Clase Pizarra: define una Pizarra sobre la que se pueden dibujar elementos de tipo Circulo, Rectangulo y Cuadrado

Version:

2011

Author:

Libro-IIP-PRG

Constructor Summary

[Pizarra\(\)](#)

Construye una Pizarra por defecto en la que es posible situar elementos gráficos.

[Pizarra\(String titulo, int dimX, int dimY\)](#)

Construye una Pizarra con cierto título y tamaño en la que es posible situar elementos gráficos.

Method Summary

void [add\(Object o\)](#)

Añade un objeto gráfico a la Pizarra y lo dibuja.

void [dibujaTodo\(\)](#)

Redibuja todos los elementos gráficos que se han añadido a la Pizarra.

Figura 2.2: Parte de la documentación de la clase Pizarra.

```

1  /**
2   * Programa de prueba de las clases Circulo, Rectangulo y Pizarra
3   * @author Libro IIP-PRG
4   * @version 2011
5   */
6  public class PrimerPrograma {
7      public static void main(String[] args) {
8          // Iniciar el espacio para dibujar dándole nombre y dimensión
9          Pizarra miPizarra = new Pizarra("ESPACIO DIBUJO",300,300);
10
11         // Crear un Circulo de radio 50, amarillo, con centro en (100,100)
12         Circulo c1 = new Circulo(50,"amarillo",100,100);
13         // Añadirlo a la Pizarra y dibujarlo
14         miPizarra.add(c1);
15
16         // Crear un Rectangulo de 30 por 30, azul, con centro en (125,125)
17         Rectangulo r1 = new Rectangulo(30,30,"azul",125,125);
18         // Añadirlo a la Pizarra y dibujarlo
19         miPizarra.add(r1);
20
21         // Crear un Rectangulo de 100 por 10, rojo, con centro en (50,155)
22         Rectangulo r2 = new Rectangulo(100,10,"rojo",50,155);
23         // Añadirlo a la Pizarra y dibujarlo
24         miPizarra.add(r2);
25     }
26 }
```

Figura 2.3: Clase PrimerPrograma.

En la figura 2.3 se muestra el código de una Clase Programa, denominada **PrimerPrograma**, que utiliza las dos clases anteriores.

En las líneas 9 a 24 se incluyen las instrucciones (o elementos de ejecución) que se efectuarán a medida que se ejecute el propio programa. El orden de ejecución, también denominado *flujo del programa*, sigue, una tras otra, la secuencia escrita de las instrucciones:

1. Se crea una **Pizarra** con el título “ESPACIO DIBUJO” con tamaño 300x300 píxeles.
2. Se crea un **Circulo** de radio 50, color amarillo y con centro en (100,100).
3. Se añade a la **Pizarra**, dibujándolo.
4. Se crea un cuadrado de lado 30, como un **Rectangulo** de base y altura 30, color azul y con centro en (125,125).
5. Se añade a la **Pizarra**, dibujándolo.

6. Se crea un Rectangulo de base 100 y altura 10, color rojo y con centro en (50,155).
7. Se añade a la Pizarra, dibujándolo.

El resultado del programa se muestra en la figura 2.4.

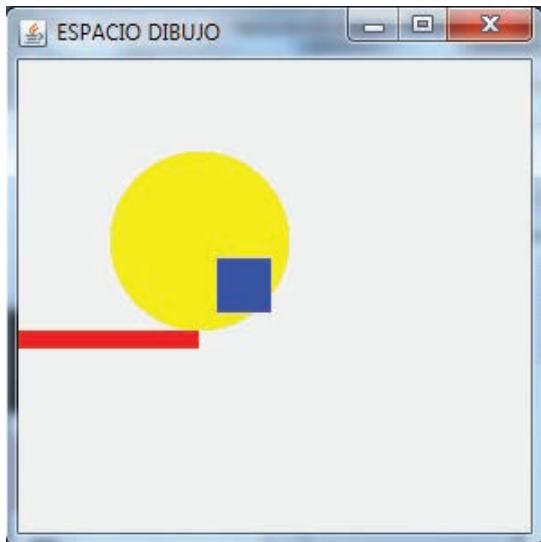


Figura 2.4: Ejecución de la clase PrimerPrograma.

2.1 Estructura básica de una clase: atributos y métodos

Los elementos de una clase Java se escriben en un fichero cuya extensión es .java. Desde un punto de vista estructural, la forma general de una clase es la que se muestra a continuación:

```
[modificadores] class NombreDeLaClase [ extends OtraClase ] {  
    [[modificadores] tipo nomVar1;  
    [[modificadores] tipo nomVar2;  
    ...  
    [[modificadores] tipo nomVarN; ]  
  
    [[modificadores] tipo nomMetodo1 ([listaParams]) { cuerpo }  
    [[modificadores] tipo nomMetodo2 ([listaParams]) { cuerpo }  
    ...  
    [[modificadores] tipo nomMetodoM ([listaParams]) { cuerpo } ]  
}
```

En esencia, tal y como aparece en el esquema, la definición de una clase es una descripción detallada de sus dos componentes básicos: *atributos* y *métodos*. Los ítems que aparecen entre corchetes son opcionales y pueden, por lo tanto, existir o no en alguna clase en particular.

Como puede verse, aparece el nombre que el programador da a la clase (**NombreDeLaClase**), sus posibles *atributos* (**nomVar1**, **nomVar2**, ..., **nomVarN**) y sus posibles *métodos* (**nomMetodo1**, **nomMetodo2**, ..., **nomMetodoM**).

Con respecto a los *modificadores*, los únicos que se utilizarán por el momento son los correspondientes al *ámbito de la declaración* (**private** y **public**), así como un modificador especial, denominado **static** que se describirá más adelante.

Mediante los modificadores correspondientes al *ámbito de declaración* se indica en qué otras clases puede el programador utilizar o no los elementos calificados. De forma más detallada, se tiene que:

- Toda la información declarada **private** es exclusiva del objeto e inaccesible desde fuera de la clase. Por ello, cualquier intento de acceso a las variables de instancia **radio** o **color** que se realice fuera de la clase **Circulo** (por ejemplo, en la clase **PrimerPrograma**) dará lugar a un error de compilación.
- Toda la información declarada **public** es accesible desde fuera de la clase. Así, en otras clases se podrá acceder a cualquiera de los métodos así definidos; es el caso de los métodos **getRadio()** o **area()** de la clase **Circulo**.

Atributos

Los *atributos* o *variables de instancia* (**nomVar1**, **nomVar2**, ..., **nomVarN**) representan información propia de cada objeto de la clase y se declaran de un *tipo de datos* determinado, siendo definidos habitualmente de acceso privado. El tipo de datos define los valores que el atributo puede tomar y las operaciones que sobre él se pueden realizar. Este tipo puede ser primitivo o una clase. En la clase ejemplo **Circulo** se definen los siguientes atributos:

1. **radio** de tipo real (**double** en Java), que puede tomar valores reales, por ejemplo 2.57 y puede formar parte de expresiones aritméticas como, por ejemplo, para el cálculo del perímetro **2*3.14*radio**.
2. **color** de tipo **String**, clase predefinida en Java y cuyos valores posibles son las frases que se pueden formar con los símbolos aceptados en el lenguaje.¹

¹Aunque el lenguaje Java tiene maneras más precisas de representar el color de un objeto, en este primer ejemplo se ha optado por esta versión sencilla, pero limitada, de los colores representables

3. **centroX** y **centroY**, de tipo entero (`int` en Java) que se corresponden con valores numéricos enteros que, al igual que lo que ocurre con valores de otros tipos numéricos, pueden formar parte de expresiones aritméticas.

Mediante estos dos atributos se mantiene el centro del objeto, definido en un espacio de representación en píxeles que se corresponde con la pantalla y que tiene su origen $(0,0)$ en la esquina superior izquierda de la misma.

En ocasiones es necesario definir *atributos* o *variables de clase* que en lugar de estar asociadas a cada objeto individual, instancia de la clase, supongan información común, idéntica en todos los objetos de la clase; para ello se utiliza el modificador `static` con dichos atributos.

Métodos

Los *métodos* definen las operaciones que se pueden aplicar sobre los objetos de la clase y se describen indicando:

1. Su *cabecera* o *perfil*, en la que se detalla el nombre del método, por ejemplo `perímetro` de la clase `Círculo`, el tipo del resultado que devuelve el método, `int` en el caso del método `perímetro` de la clase `Círculo` y la lista de parámetros que se requieren para el cálculo si fuera necesario, el método `perímetro` no tiene parámetros. Nótese que es posible que un método no devuelva un valor, circunstancia que se representa indicando que el tipo del resultado del método es `void`. Éste es el caso del método `setRadio` de la clase `Círculo` que, nótese, tiene un parámetro de tipo `double`.
2. Su *cuerpo*, que contiene la secuencia de instrucciones que se deben efectuar cuando el método se ejecute. Podrán formar parte de esta secuencia de instrucciones cualesquiera de las que constituyen el repertorio del lenguaje y que se estudiarán en capítulos sucesivos: asignación, composición, instrucciones condicionales, de repetición y combinadas. A menos que el tipo del resultado del método sea `void`, la instrucción `return` es de aparición obligada y su efecto es devolver el resultado calculado. Por ejemplo el cuerpo del método `perímetro` de la clase `Círculo` tiene una única instrucción que es `return 2*3.14*radio.`

Los métodos se pueden clasificar, atendiendo a su función con respecto al objeto del modo siguiente:

- *Constructores*: Son métodos que permiten crear el objeto. En el ejemplo es el método `Círculo(double, String, int, int)`. Pueden tener o no argumentos y se utilizan para inicializar el objeto de una forma dada.

- *Modificadores:* Son métodos que permiten alterar el estado (valores de las variables de instancia) del objeto. El método `setRadio(double)` es ejemplo de uno de ellos.
- *Consultores:* Son métodos que permiten conocer, sin alterar, el estado del objeto. En el ejemplo, son métodos como: `getRadio()`, `getCentroX()` o `perímetro()`.

En Java existe un método especial denominado `main` que indica el punto de inicio de ejecución del código. Su cabecera se define como sigue y se tiene un ejemplo en la Clase Programa `PrimerPrograma`.

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

Aunque en general los identificadores de atributos y métodos deben ser diferentes, el lenguaje Java permite explícitamente la, así denominada, *sobrecarga de métodos*. Se denomina *sobrecarga* a la definición de un mismo ítem (símbolo, identificador, etc.) con distintos significados, de forma que, en función del modo en que se utilice, pueda interpretarse su significado de una u otra forma.

Un ejemplo habitual de sobrecarga en Java es la del operador `+` que tanto puede utilizarse para expresar la suma de valores numéricos como la concatenación de elementos de tipo `String`.

Así pues, dos métodos cualesquiera, existentes en el mismo ámbito, con el mismo nombre y con diferente lista de argumentos se dice que están *sobrecargados*.

La *sobrecarga* explícita de los métodos tiene una gran importancia en la POO, especialmente debido a su uso cuando hay *herencia*, como se estudiará más adelante. Sin embargo, es prácticamente inexistente en lenguajes de programación tradicionales tales como C y Pascal.

Como se puede comprobar en la clase `Círculo` hay dos métodos constructores, denominados ambos `Círculo` que difieren entre sí por sus parámetros. Durante la ejecución de un programa el lenguaje Java seleccionará, según que argumentos se utilicen, un método u otro.

2.2 Creación y uso de objetos: operadores `new` y “.”

Para poder utilizar un objeto de una clase determinada hay que crearlo y declararlo, dándole previamente un nombre. Esto se hace mediante el operador `new`. Una descripción más detallada se encuentra en el capítulo 4.

Considérese, por ejemplo la siguiente secuencia en Java mediante la que se declara y crea un objeto de tipo **Círculo**:

```
// Crear un Círculo "c1" con los valores definidos por defecto  
Círculo c1 = new Círculo();  
// Crear un Círculo "c2" de radio 50, amarillo y centro (100,100)  
Círculo c2 = new Círculo(50,"amarillo",100,100);
```

o la siguiente, en el programa de la figura 2.3, para la declaración de un objeto de tipo **Pizarra**:

```
// Iniciar el espacio para dibujar, dándole nombre y dimensión  
Pizarra miPizarra = new Pizarra("ESPACIO DIBUJO",300,300);
```

Esto es, cuando se desea utilizar un nuevo objeto de cierto tipo, es necesario crearlo explícitamente. Hasta el momento de su creación el objeto no existe y cualquier intento de utilizarlo antes de dicho momento provocará un error durante la ejecución.

Asociados a los objetos, definidos en la clase a la que pertenecen, pueden existir atributos pertenecientes a ellos o métodos que se podrán aplicar a los mismos. El *operador punto “.”* se emplea en dichos casos para seleccionar el atributo deseado o el método específico que se desee utilizar sobre el objeto.

Véase como ejemplo el uso del método **add** en la clase **PrimerPrograma** sobre el objeto **miPizarra** de la clase **Pizarra**:

```
// Añadirlo a la Pizarra y dibujarlo  
miPizarra.add(c1);
```

Hay que notar que si se intenta aplicar un método asociado a un objeto, cuando este último no existe (porque, por ejemplo, el objeto no ha sido creado anteriormente), se producirá una condición de error que en Java se denomina una *Excepción* (en particular, la denominada **NullPointerException**). El tratamiento de errores en Java se estudia en el capítulo 15.

2.3 La organización en paquetes del lenguaje Java

Como se verá, un programa escrito en Java consistirá frecuentemente en un número amplio de clases que, de una forma u otra, estarán relacionadas entre ellas a la hora de establecer la solución a un problema.

En muchas ocasiones, los programas que se construyan utilizarán elementos previamente definidos, existentes muchos de ellos en el propio lenguaje Java, formando parte de las, así denominadas, *librerías del lenguaje*.

Por ejemplo, si se desea utilizar la capacidad gráfica del lenguaje Java en la resolución de un problema, convendrá usar las características de manipulación gráfica ya definidas en el lenguaje. Estos elementos del Java ya existentes, serán en la práctica un grupo de clases y métodos predefinidos que el programador podrá utilizar en el desarrollo de su solución.

Para facilitar la organización y uso de los elementos ya definidos y permitir la definición y uso de otros nuevos, el lenguaje Java hace uso del concepto de *paquete* (*package*). Un *package* del Java consiste en un grupo de clases cuyas definiciones y operaciones pueden ser *importadas* y, tras ello, utilizadas en un programa. Por ejemplo, considérese el segmento de código siguiente que forma parte del comienzo de la clase **Pizarra**:

```
import javax.swing.*;
import java.awt.*;

/**
 * Clase Pizarra: define una Pizarra sobre la que se pueden
 * dibujar elementos de tipo: Circulo, Rectangulo y Cuadrado
 *
 * @author Libro IIP-PRG
 * @version 2011
 */
public class Pizarra extends JFrame {
    .....
    .....
}
```

Mediante las dos primeras líneas se indica que en la clase **Pizarra** se importan y por lo tanto se pueden utilizar, todos los elementos existentes en los paquetes predefinidos en el Java: **awt** y **swing**. Además, en la declaración de la cabecera de la clase **Pizarra** figura una referencia a la clase **JFrame** que, precisamente, se encuentra definida en el paquete **swing**.

Más adelante, en la clase **Pizarra**, figura el método constructor:

```
public Pizarra(String titulo, int dimX, int dimY) {
    super(titulo);
    setSize(dimX, dimY);
    setContentPane(initPanel());
    setVisible(true);
}
```

en el que se utilizan algunas operaciones (**setContentPane**, **setVisible**, **setSize**) cuyo uso es posible por haberse importado en los paquetes mencionados.

En general, mediante la estructura de paquetes inherente al Java, es posible tanto importar paquetes predefinidos para su uso posterior; como definir nuevos paque-

tes, incorporando en los mismos las clases que se deseen. Estos paquetes, definidos por el programador, pueden luego ser utilizados en el desarrollo de nuevos programas del mismo modo que se hace con los paquetes predefinidos en el lenguaje.

Como ejemplo de esto último, obsérvese la siguiente variación del código inicial de la clase **Pizarra**, en el que aparece una nueva línea, al comienzo de la clase, mediante la que se informa de que ahora **Pizarra** se encuentra definida en un paquete denominado **libUtil**:

```
package libUtil;

import javax.swing.*;
import java.awt.*;

/**
 * Clase Pizarra: define una Pizarra sobre la que se pueden
 * dibujar elementos de tipo: Circulo, Rectangulo y Cuadrado
 *
 * @author Libro IIP-PRG
 * @version 2011
 */
public class Pizarra extends JFrame {
    ....
```

Hecho esto, podrían utilizarse a continuación los elementos públicos de la clase **Pizarra** importando dicho paquete cuando sea necesario.

En Java, las clases se estructuran siempre dentro de paquetes, cuando no se referencia a qué paquete pertenece una clase, se supone implícitamente que está en uno especial, sin nombre, que se denomina *anonymous* y que comprende todas las clases existentes en el directorio del sistema en el que se está trabajando. Todos los ejemplos de clases Java vistos hasta ahora, tales como, **Circulo**, **PrimerPrograma** y **HolaATodos**, se han definido, por simplicidad, de dicha manera.

Cabe señalar que el paquete **java.lang** se importa por defecto en cualquier clase y sus métodos públicos son accesibles de forma directa. Forman parte de este paquete, por ejemplo, las clases **Object**, **String** y **Math**.

Dada la relevancia que tienen los aspectos organizativos en la construcción sistemática de programas, la definición y uso de paquetes en Java se abordará en capítulos sucesivos con bastante más detalle.

2.4 La herencia. Jerarquía de clases, la clase **Object**

Como se ha mencionado en el capítulo anterior, uno de los objetivos fundamentales de la POO es la de facilitar la *reutilización del código* y precisamente éste ha sido

uno de los objetivos buscado con la creación de nuevos lenguajes de programación. En particular, en los lenguajes de programación orientados a objetos el mecanismo básico para el reuso del código es la *herencia*. Mediante ella es posible definir nuevas clases extendiendo o restringiendo las funcionalidades de otras clases ya existentes.

La *herencia* es un mecanismo que permite modelar relaciones jerárquicas entre elementos, del tipo *is-a* (*es un(a)*), como por ejemplo en la relación que se da entre las clases **Pizarra** y **JFrame**, en la que una **Pizarra** *es un* **JFrame**. En una relación así un elemento, el *heredero*, tiene las características de otro elemento pero, tal vez, refinándolas para definirlo como un caso especial del primero.

Desde el punto de vista del lenguaje Java hay dos puntos donde la herencia es particularmente relevante. Por una parte la herencia se emplea exhaustivamente en el propio lenguaje a lo largo del conjunto de librerías de clases que posee. Por otra parte el lenguaje, como cabía prever, da soporte a la definición de nuevas clases *herederas* de las características de otras ya definidas. Todo esto se estudiará con detalle en el capítulo 14.

La librería de clases del lenguaje se encuentra organizada de forma jerárquica, pudiéndose representar la jerarquía de clases del lenguaje mediante un árbol de bastante profundidad. Véase, por ejemplo, la jerarquía correspondiente a la clase **JFrame**, a partir de la cual se ha definido la clase **Pizarra**, tal y como aparece en la ayuda *on line* del lenguaje [Ora11c]:

```
java.lang.Object
|
+--java.awt.Component
|
+--java.awt.Container
|
+--java.awt.Window
|
+--java.awt.Frame
|
+- javax.swing.JFrame
```

Cada una de las clases del ejemplo: **Component**, **Container**, **Window**, **Frame**, **JFrame**, heredan en sus definiciones los atributos y métodos de las clases precedentes, sobrescribiéndolos cuando así lo necesitan, de forma que las funcionalidades de una clase quedan definidas por las de las clases que extienden, junto con las aportadas por ella misma. Nótese que los nombres de las clases del ejemplo anterior vienen antepuestos por los nombres de los paquetes `java.lang`, `java.awt` y `javax.swing`.

Existe una clase inicial, primera en el árbol de la jerarquía de clases, denominada `Object`. Todas las clases de Java son, de una forma u otra, descendientes de la clase `Object`. Igualmente todas las clases del Java heredan, a veces redefiniéndolos, los métodos de dicha superclase. Tres de estos métodos que por su relevancia se examinan con detalle en el capítulo 5 son: `clone()`, para poder duplicar un objeto; `equals()`, para determinar la igualdad de dos objetos y `toString()`, mediante el que se obtiene una representación imprimible, como `String`, del objeto sobre el que se aplique.

2.5 Edición, compilación y ejecución en Java

Una vez escrito y guardado el contenido de un programa en un fichero del sistema, hay que traducirlo del lenguaje en que ha sido escrito (Java), a una secuencia de instrucciones reconocibles por el sistema operativo y el procesador; esto es, su *compilación*.

El programa más sencillo en Java se define como una clase sin atributos y con un único método, el `main`, con una instrucción para mostrar por pantalla un saludo. El código es el que se muestra en la figura 2.5.

```
/**  
 * Ejemplo de programa que muestra por la salida estándar  
 * el mensaje "Hola a todos".  
 * @author Libro IIP-PRG  
 * @version 2011  
 */  
public class HolaATodos {  
    public static void main(String[] args) {  
        System.out.println("Hola a todos");  
    }  
}
```

Figura 2.5: El programa `HolaATodos`.

Este programa muestra por pantalla el saludo `Hola a todos`. El `main` consta de una instrucción que escribe en pantalla la cadena de caracteres delimitada entre comillas dobles. Esta instrucción es una llamada a un método (`println`) de la clase que representa la salida estándar (`System.out`).

Para compilar el programa que ha sido editado, se debe invocar al programa que realiza esta traducción, indicándole el fichero que contiene el código fuente, como en el siguiente ejemplo:

```
javac HolaATodos.java
```

en donde `javac` significa *java compiler*. Si el programa está escrito correctamente, y no aparecen errores de compilación, se genera un fichero con la extensión `.class` y con el mismo nombre que el de la clase que está contenida en el fichero donde está escrito el programa. Este fichero de extensión `.class` contiene los denominados *bytecodes*, o instrucciones para la máquina virtual Java (*JVM*), resultantes del proceso de compilación.

Finalmente se puede ejecutar el programa, mediante la ejecución de la *JVM*, con el siguiente comando:

```
java HolaATodos
```

en donde `java` es la invocación a la máquina virtual, y `HolaATodos` es el nombre de la clase que se ha editado en el fichero `HolaATodos.java`, y que aparece compilada en el fichero `HolaATodos.class`.

2.5.1 Errores en los programas. Excepciones

En la construcción de programas es bastante posible que puedan aparecer errores que imposibiliten su ejecución o, lo que puede ser incluso peor, que alteren su comportamiento con respecto a lo pretendido. Básicamente hay tres tipos de errores que se deben considerar y que se describen a continuación por orden de su posible aparición:

1. *Errores de compilación* que, como su nombre indica, surgen en esa fase de la realización de un programa. Estos errores se deben a que el programa incumple alguna de las características de la definición del lenguaje, detectándose el hecho por el compilador.

El compilador del Java es del tipo *múltiple pasada* o, lo que es lo mismo, está organizado en fases que se ejecutan sólo si se ha pasado correctamente las fases previas; de forma que en cada una de ellas se vigila una característica determinada del código. En las primeras fases se detectan *errores léxicos* y *sintácticos*, mientras que en fases posteriores se determinan otros errores, tales como los que pueden aparecer en la declaración incorrecta de elementos o en la realización de operaciones no permitidas, etc.

Generalmente estos errores son sencillos de corregir gracias a la ayuda proporcionada por el compilador y al uso de la documentación del lenguaje.

2. *Errores de ejecución* que provocan un malfuncionamiento del programa. Se suelen subdividir en los denominados *errores en tiempo de ejecución* y *errores lógicos*. Provocando los primeros la detención de la ejecución, mientras que los segundos, los más difíciles de descubrir, consisten en que los resultados obtenidos, o los procesos realizados, por el programa o una parte del

mismo no son correctos aunque el programa puede parecer que funciona correctamente.

En general, los errores de ejecución pueden ser difíciles de detectar y resolver ya que pueden darse de forma esporádica cuando se den determinadas condiciones especiales. Por ejemplo, un error que se da sólo cuando un elemento tiene un valor dentro de un rango reducido o el que pueda aparecer cuando se intente manipular un elemento que circunstancialmente no exista, etc.

Para abordar y resolver los errores de ejecución, es necesario probar exhaustivamente los programas para comprobar que su comportamiento se corresponde con el pretendido. Muchas veces se elaboran sistemáticamente *bancos de pruebas* que son conjuntos de tests que tratan de someter a los programas a todas las posibles combinaciones de uso o, si eso es muy difícil, se intenta probar al menos un subconjunto relevante de las mismas.

Por otra parte, existen programas especializados, denominados *depuradores*, en inglés *debuggers*, que permiten la ejecución controlada de un programa o segmento del mismo. Con ellos es posible ejecutar, por ejemplo, instrucción a instrucción un segmento de código, examinando mientras tanto el valor de las variables implicadas, el estado de la memoria, las ejecuciones realizadas junto con sus características, etc.

Los depuradores son en muchas ocasiones una herramienta imprescindible para determinar con precisión el funcionamiento de un segmento de código y el porqué de un determinado error.

A veces, los errores en tiempo de ejecución provocan un error del sistema que en la mayoría de las ocasiones implica la detención del programa. En Java, se denominan *Excepciones* a dicho tipo de errores. Una *Excepción* es en Java cierto tipo de objeto que, si se desea, puede ser manipulado para así gestionar debidamente el error que la provoca. Por ejemplo, se puede conseguir que un acceso a un ordenador remoto inexistente sea detectado por el programa y resuelto con un aviso al usuario del programa, en lugar de que se detenga por completo la ejecución del mismo.

En este libro, las excepciones y su tratamiento se tratan con detalle en el capítulo 15.

2.6 Uso de comentarios. Documentación de programas

Como se ha comentado en alguno de los ejemplos anteriores, cualquier parte de un programa englobada en una secuencia: /* ... */ o que forme parte de una línea que vaya precedida en la misma por la doble barra (//), son considerados comentarios, de los que el compilador hará caso omiso y que serán irrelevantes durante

la ejecución del programa, tal como se muestra en los ejemplos siguientes, en los que los puntos suspensivos representan secuencias de instrucciones cualesquiera.

```
...
/*
  Todo lo que se encuentra entre la barra estrella anterior
  y la siguiente estrella barra es un comentario para el Java.
*/
...
int d = 10; // y ahora lo que queda de línea es un comentario
...
```

Este tipo de comentarios se utiliza por lo general para documentar aquellos aspectos del código de los programas que se considere relevante.

Además de este tipo de comentarios, existe en Java otra clase especial que sí que puede ser procesada por el lenguaje para generar de forma automática, a partir de los comentarios hechos en el código y escritos de una forma determinada, la documentación de las clases siguiendo un formato estandarizado. Para dicha generación se utiliza una herramienta denominada `javadoc` [Ora11b] que viene incluida en la instalación estándar del Java.

Para poder generar este tipo de documentación, hay que incluir en el programa comentarios que especifiquen los métodos, precediendo cada uno de ellos. Además, estos comentarios de documentación irán escritos en un formato similar al del que se puede ver en la figura 2.1 que se muestra de nuevo, parcialmente, a continuación:

```
/** consulta el radio del Circulo. */
public double getRadio() { return radio; }

/** actualiza el radio del Circulo a nuevoRadio. */
public void setRadio(double nuevoRadio) { radio = nuevoRadio; }
```

La documentación de los métodos de una clase sirve para indicar cómo usarlos, es decir, cuál es su cabecera, qué condiciones especiales deben cumplir los parámetros, si las hubiera, y cuál es el resultado que se puede esperar en cada caso de los datos.

Es por ello que, en general, se recomienda, al incluir este tipo de documentación en la cabecera de los métodos, evitar cualquier referencia a cómo se han implementado. En concreto, los comentarios que anoten aspectos de implementación deberán aparecer en el cuerpo de los métodos, para uso exclusivo del implementador. Una descripción más detallada de la documentación de los métodos aparece en el capítulo 5.

Además, cuando se crea una clase nueva el código debe venir precedido por un comentario de documentación que incluye la descripción de la clase y, precedidas

por las etiquetas `@author` y `@version` respectivamente, el nombre del autor o autores y el número de versión o fecha de creación de la clase; por ejemplo, en la clase `Circulo` se tiene:

```
/**  
 * Clase Circulo: define un círculo de un determinado radio, color y  
 * posición de su centro, con la funcionalidad que aparece a  
 * continuación. <br>  
 * @author Libro IIP-PRG  
 * @version 2011  
 */
```

Nótese que en los comentarios de documentación en Java[Oracle] también se puede usar código `html` (por ejemplo, para resaltar texto en negrita ` ` o para incluir un cambio de línea `
`).

Para producir automáticamente la documentación `html` de un código comentado de esta forma se utiliza el comando `javadoc Circulo`, bien desde el sistema, bien mediante el uso de alguna herramienta de desarrollo de código.

En cualquier caso, el resultado es un fichero `html` que se puede abrir con cualquier navegador y cuyo resultado para el ejemplo anterior se puede ver en la figura 2.6.

Class Circulo

```
java.lang.Object
└ Circulo
```

```
public class Circulo
extends Object
```

Clase Circulo: define un círculo de un determinado radio, color y posición de su centro, con la funcionalidad que aparece a continuación.

Version:

2011

Author:

Libro IIP-PRG

Constructor Summary

<code>Circulo()</code>	crea un Circulo de radio 50, negro y centro en (100,100).
<code>Circulo(double r, String c, int px, int py)</code>	crea un Circulo de radio r, color c y centro en (px,py).

Method Summary

<code>void aLaDerecha()</code>	desplaza un poco a la derecha el Circulo.
<code>double area()</code>	calcula el área del Circulo.
<code>void crece()</code>	incrementa el radio del Circulo.
<code>void decrece()</code>	decrementa el radio del Circulo.
<code>int getCentroX()</code>	consulta la abscisa del centro del Circulo.
<code>int getCentroY()</code>	consulta la ordenada del centro del Circulo.
<code>String getColor()</code>	consulta el color del Circulo.
<code>double getRadio()</code>	consulta el radio del Circulo.
<code>double perimetro()</code>	calcula el perímetro del Circulo.
<code>void setCentro(int px, int py)</code>	actualiza el centro del Circulo a la posición (px,py).
<code>void setColor(String nuevoColor)</code>	actualiza el color del Circulo a nuevoColor.
<code>void setRadio(double nuevoRadio)</code>	actualiza el radio del Circulo a nuevoRadio.
<code>String toString()</code>	obtiene un String con las componentes del Circulo.

Figura 2.6: Parte de la documentación de la clase Circulo.

2.7 Problemas propuestos

1. Dada la clase Punto que se muestra en la figura 2.7 se pide identificar sus elementos y en concreto:
 - a) Indicar sus atributos, de qué tipo son cada uno y cuál es su nivel de visibilidad.
 - b) Escribir el perfil de los métodos constructores. ¿En qué se diferencian del resto de métodos? ¿Para qué se utilizan?
 - c) Identificar los métodos modificadores.
 - d) Identificar los métodos consultores.

```
/**  
 * Clase Punto: define puntos en un espacio bidimensional entero  
 * con la funcionalidad que se indica a continuación. <br>  
 * @author Libro IIP-PRG  
 * @version 2011  
 */  
public class Punto {  
    private int x; // abscisa del punto  
    private int y; // ordenada del punto  
    /** crea un punto (0,0). */  
    public Punto() { x = 0; y = 0; }  
    /** crea un punto (abs, ord). */  
    public Punto(int abs, int ord) { x = abs; y = ord; }  
    /** crea un punto (coord, coord). */  
    public Punto(int coord) { x = coord; y = coord; }  
    /** consulta la abcisa del punto. */  
    public int abscisa() { return x; }  
    /** consulta la ordenada del punto. */  
    public int ordenada() { return y; }  
    /** consulta la distancia al origen del punto. */  
    public double distOrigen() { return Math.sqrt(x*x + y*y); }  
    /** actualiza las componentes del punto a (abs, ord). */  
    public void asignar(int abs, int ord) { x = abs; y = ord; }  
}
```

Figura 2.7: Clase Punto.

2. Dada la clase Punto del ejercicio anterior se pide escribir las instrucciones Java para:
 - a) Declarar y crear un objeto de tipo Punto cuyo nombre sea p1.
 - b) Mostrar por pantalla la distancia al origen de dicho punto.

- c) Escribir la clase **PruebaPunto** en cuyo **main** se deben incluir las instrucciones anteriores. Compilar y ejecutar el programa.
3. ¿Qué error tiene el siguiente programa?
- ```
public class PruebaCirculo {
 public static void main(String[] args) {
 Circulo c = new Circulo(2.5, "rojo", 1, 1);
 System.out.println("El radio del circulo es:" + c.radio);
 }
}
```
4. Se pide completar el código de la clase **Cuadrado** (figura 2.8) para que tenga una funcionalidad similar a la clase **Circulo**.
5. Modificar el programa **PrimerPrograma** (figura 2.3) para usar **Cuadrado** como tipo de **r1** en lugar de **Rectangulo**.
6. Modificar la clase **Circulo** para sustituir los dos atributos **centroX** y **centroY** por un único atributo **centro** de tipo **Punto**.

```
public class Cuadrado {
 private ... lado;
 private ... color;
 private ... centroX;
 private ... centroY;

 /** crea un Cuadrado de lado 50, negro y centro en (100,100).*/
 public Cuadrado() { ... }
 /** crea un Cuadrado de lado l, color c y centro en (px,py).*/
 public Cuadrado(double l, String c, int px, int py) {
 ...
 }

 /** consulta el lado de un Cuadrado. */
 public double getLado() { ... }
 /** consulta el color de un Cuadrado. */
 public String getColor() { ... }
 /** consulta el centro de un Cuadrado. */
 public int getCentroX() { ... }
 /** consulta el centro de un Cuadrado. */
 public int getCentroY() { ... }

 /** actualiza el lado de un Cuadrado a nuevoLado. */
 public void setLado(double nuevoLado) { ... }
 /** actualiza el color de un Cuadrado a nuevoColor. */
 public void setColor(String nuevoColor) { ... }
 /** actualiza el centro de un Cuadrado. */
 public void setCentro(int px, int py) { ... }

 /** desplaza un poco a la derecha el Cuadrado. */
 public void aLaDerecha() { ... }
 /** incrementa el lado de un Cuadrado. */
 public void crece() { ... }
 /** decrementa el lado de un Cuadrado. */
 public void decrece() { ... }

 /** calcula el área de un Cuadrado. */
 public double area() { ... }
 /** calcula el perímetro de un Cuadrado. */
 public double perimetro() { ... }
 /** obtiene el String con las componentes de un Cuadrado. */
 public String toString() { ... }
}
```

Figura 2.8: Clase Cuadrado (incompleta).

## Más información

[BK07] D.J. Barnes and M. Kölking. *Programación Orientada a Objetos con Java: una introducción práctica usando BlueJ*. Pearson Educación, 2007. Capítulos 1 y 2 (2.1 a 2.10).

[Ora11a] Oracle. *How to Write Doc Comments for the Javadoc Tool*, 2011. URL: <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>.

[Ora11b] Oracle. *Javadoc Tool*, 2011. URL: <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>.

[Ora11c] Oracle. *Java<sup>TM</sup> Platform, Standard Edition 6, API Specification*, 2011. URL: <http://download.oracle.com/javase/6/docs/api/>.

[Sch07] H. Schildt. *Fundamentos de Java*. McGraw-Hill, 2007. Capítulo 1 (1.1 a 1.6) y Capítulo 4 (4.1 y 4.2).

# Capítulo 3

## VARIABLES Y ASIGNACIÓN. TIPOS DE DATOS ELEMENTALES. BLOQUES

Como ya se ha comentado, la actividad de programar consiste en escribir programas, es decir, secuencias de instrucciones descritas en un determinado lenguaje de programación que tratan la información para resolver un problema.

La información relativa al problema y su resolución se puede representar mediante sus datos y sus resultados intermedios y finales. Estos datos y resultados pueden ser más o menos complejos y se manejan en los programas mediante lo que se denominan *variables*. Una variable se caracteriza por ser de un *tipo* determinado (numérica, cadena de caracteres, etc.) que determina el conjunto de operaciones que sobre ella se pueden realizar. Las variables, se almacenan en la memoria del computador ocupando más o menos posiciones dependiendo de su tipo.

En este capítulo se introducirán los conceptos de variable y tipo de dato, se estudiará la instrucción básica de la programación imperativa, *la asignación*, y se presentarán los aspectos fundamentales para el uso de los tipos de datos básicos o elementales. Finalmente se introducen las características de los *bloques*, mecanismo mediante el que es posible agrupar, en un contexto común, un conjunto de declaraciones e instrucciones.

### 3.1 Tipos de datos

Si un *dato* es cualquier información dispuesta de manera adecuada para su tratamiento por un ordenador, un *tipo de datos* se refiere a la clase de información de que se trata. Nótese que aunque la información manipulable de forma elemental

por el computador son bits, los lenguajes de programación permiten tratarla a un nivel de abstracción más próximo al planteamiento del problema a resolver. El concepto de tipo de dato ha evolucionado en los últimos años y en la actualidad se puede definir como un conjunto de valores y un conjunto de operaciones permitidas sobre ellos.

Los tipos de datos se pueden clasificar en elementales o primitivos y complejos o estructurados. Son *tipos elementales o primitivos* los que no se definen a partir de otros y su representación y operaciones vienen dadas por el propio lenguaje. Son *tipos complejos o estructurados* aquellos que se construyen por agregación de datos que pueden ser del mismo o de distinto tipo; estos pueden venir *predefinidos* en el lenguaje, normalmente en forma de librerías, o pueden ser *definidos por el programador*. Por ejemplo, el tipo `int` del atributo `radio` de la clase `Circulo` de la figura 2.1 es un tipo elemental mientras que el tipo predefinido `String` del atributo `color` no lo es. Tampoco es un tipo elemental el tipo definido por el programador `Circulo`.

En Java todos los tipos de datos que no son primitivos, predefinidos o no, se consideran clases y se manipulan utilizando el concepto de *referencia*. Esto se trata en el capítulo 4.

## 3.2 Variables

Todos los datos que se manejan en la resolución de un problema mediante un programa se representan mediante *variables*. Según el uso que se vaya a hacer de la variable, éstas se pueden clasificar como:

- *Atributos o variables de instancia y de clase* que se definen en una Clase Tipo de Dato, por ejemplo `radio` es un atributo o variable de instancia de la clase `Circulo` (figura 2.1). Se estudiarán en detalle en el capítulo 4.
- *Variables locales* que son las que se definen en el método `main` de una Clase Programa o en cualquier bloque de instrucciones o método, como se verá en capítulos sucesivos; por ejemplo la variable `miPizarra` del programa `PrimerPrograma` (figura 2.3).
- *Parámetros* de un método, como por ejemplo `nuevoRadio` del método `setRadio` de la clase `Circulo` (figura 2.1). El manejo de parámetros se estudiará con detalle en el capítulo 5.

A la descripción de las características de una variable se la denomina *declaración de variable* y en ella se define el nombre o *identificador* de la variable y el *tipo de datos* que restringe los valores que puede almacenar y las operaciones que sobre ella se pueden realizar. *Java es un lenguaje fuertemente tipado*, lo que significa

que exige la declaración de todas las variables antes de su uso. La sintaxis para declarar variables atributos o locales en Java es básicamente la siguiente:

```
tipo nomvar1, nomvar2, ..., nomvarn;
```

donde **tipo** es el nombre del tipo de datos y **nomvari** los identificadores elegidos para las variables. Los identificadores están separados por comas y finalizan con un punto y coma; las comas no son necesarias cuando sólo se define una variable. Los identificadores deben comenzar por una letra y, a continuación, cualquier combinación de letras, números, el carácter subrayado (\_) y el signo de dólar (\$). Nótese que no hay ningún elemento en la sintaxis de la declaración que permita distinguir entre un atributo y una variable local; sin embargo, se distinguen por el lugar donde se definen ya que las variables locales se definen en los métodos, por ejemplo en el **main**. Además, solo los atributos pueden ir precedidos por los modificadores de visibilidad y ámbito apropiados.

A continuación se muestra un ejemplo de definición de variables: las tres primeras son de tipo entero, la cuarta es de tipo carácter y las dos últimas son reales.

```
int var1, var2, suma;
char c;
double d1, d2;
```

El compilador asigna valores por defecto a los atributos (por ejemplo 0 para las variables numéricas y **null**<sup>1</sup> para las referencias). No ocurre igual para las variables locales; así, acceder a una variable local que no esté inicializada da lugar a un error de compilación. Las variables pueden cambiar de valor durante la resolución del programa y ésta es la diferencia básica con las variables que se utilizan en matemáticas; una vez determinado su valor, no cambia. En este sentido, se denomina *estado de una variable* en un determinado momento de la ejecución al contenido de la variable en ese momento. Desde este punto de vista, la ejecución de un programa se puede ver como una sucesión de cambios de estado que transforman un cierto estado inicial (los datos) en un determinado estado final (solución). El *estado de un programa* es el contenido de sus variables en un momento de la ejecución. Se llama *traza de la ejecución* de un programa al seguimiento de la evolución de los valores de las variables en una ejecución.

---

<sup>1</sup>La constante **null** se trata en el capítulo 4.

### 3.3 Expresiones y asignación. Compatibilidad de tipos

Una variable cambia de valor mediante la instrucción denominada *asignación* que tiene la siguiente sintaxis:

```
identificador = expresión;
```

Nótese el uso del símbolo = como símbolo de asignación. A la izquierda del símbolo de asignación debe aparecer el identificador de la variable sobre la que se realiza la asignación y a la derecha una expresión de *tipo compatible*. En general, una *expresión* es una sucesión (sintácticamente correcta) de valores, variables, operadores y llamadas a métodos que se evalúa a un único valor, siendo el tipo de la expresión el tipo de este valor.

La operación de asignación primero evalúa, es decir, obtiene el valor de la expresión a su derecha (*expresión*) y después guarda el valor resultante de la evaluación en la variable cuyo identificador (*identificador*) tiene a su izquierda. Por ejemplo, en el siguiente código Java se declara la variable **cantidadInicial** de tipo **int** y se le asigna el valor 50.

```
int cantidadInicial;
cantidadInicial = 50;
```

Además, la instrucción de asignación se evalúa a un resultado que, como cualquier otro valor, es susceptible de ser utilizado o no. Por ejemplo, en el código que se muestra a continuación se usa el valor al que se evalúa la operación de asignación para asignarlo, a su vez, a **cantidadReal**. Primero se evalúa la expresión a la derecha de la primera asignación, es decir: **cantidadInicial** = 50 y 50, su valor, se asigna a **cantidadReal**. El efecto final es que las dos variables contienen el mismo valor.

```
int cantidadReal, cantidadInicial;
cantidadReal = cantidadInicial = 50;
```

Como se ha señalado, la instrucción de asignación exige que variable y expresión sean de tipos compatibles; el caso más sencillo de compatibilidad de tipos se tiene cuando variable y expresión son exactamente del mismo tipo. No obstante, como se verá más adelante existe la posibilidad de transformar el tipo de una expresión para hacerla compatible bien implícitamente (de forma automática) bien explícitamente (mediante lo que se conoce como *casting*).

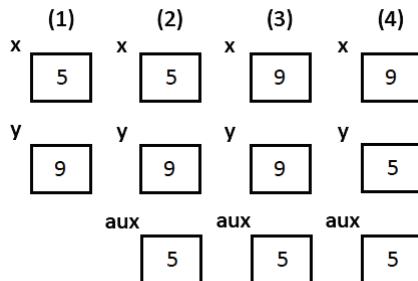
La instrucción de asignación se puede utilizar también en el momento de la declaración de una variable para asignarle un valor inicial, lo que se conoce como su *inicialización*. Por ejemplo:

```
int var1, var2, suma = 5;
char ch1, ch2 = 'u';
double d1 = 2.0, d2 = 3.0 + d1;
var1 = 15;
suma = suma + 2;
```

Nótese que el contenido de una variable se pierde cuando se le asigna uno nuevo. El código que se muestra a continuación permite intercambiar los valores de dos variables:

```
int x = 5, y = 9; // Inicialización de variables
int aux = x; // Se guarda en aux copia del valor en x
 // que se pierde en la próxima instrucción.
x = y; // Se asigna el valor almacenado en y a x
 // perdiendo ésta su valor anterior.
y = aux; // Se asigna a y el valor antiguo de x que
 // está copiado en aux.
```

La evolución de los estados se ilustra en cada una de las columnas numeradas de (1) a (4) en la figura 3.1. Esta estrategia, con variable auxiliar, se puede usar para intercambiar variables de cualquier tipo.



**Figura 3.1:** Cambio de estados al intercambiar el valor de dos variables.

## 3.4 Constantes. Modificador *final*

También se pueden definir variables cuyos valores no se pueden cambiar a lo largo de la ejecución de un programa. A éstas se las llama *constantes*, aunque en realidad

son variables con valor inmutable. Este tipo de constantes se define en dos pasos. Primero se declara la variable precedida del modificador `final` y después se le asigna un valor. El modificador establece que el primer valor asignado no puede ser modificado, como se muestra en el siguiente ejemplo:

```
final int NUM_ALUMNOS;
NUM_ALUMNOS = 25;
```

aunque normalmente se utiliza una única instrucción con asignación inicial:

```
final int NUM_ALUMNOS = 25;
```

El uso de constantes es altamente recomendable para mejorar la fiabilidad y la legibilidad del código.

### 3.5 Algunas consideraciones sintácticas sobre el uso de identificadores

Algunas consideraciones sintácticas relevantes son las siguientes:

- Java es sensible a las mayúsculas, es decir, distingue mayúsculas de minúsculas. Por ejemplo, los identificadores `toString` y `tostring` no son el mismo.
- Es conveniente utilizar identificadores con nombres descriptivos, de manera que cualquier persona que acceda al código pueda conocer el significado de lo que representan. Por ejemplo: `teclado`, `suma`, `toString`, `cantidadInicial`, `cantidadReal`.
- Los identificadores de variables suelen escribirse en minúsculas. Si el identificador está formado por varias palabras, la primera palabra comienza en minúscula y el resto de palabras comienzan por una mayúscula. Por ejemplo: `radioEsfera` y `volumenCubo`.
- Los identificadores de constantes se suelen escribir en mayúsculas. Si el identificador consta de más de una palabra, dichas palabras se separan por el signo de subrayado ‘`_`’ como en el identificador `NUM_ALUMNOS`.
- Las palabras reservadas tienen un significado preestablecido y no pueden usarse como identificadores. Por ejemplo no pueden usarse como identificadores `null`, `true` o `false`. En la tabla 3.1 aparecen algunas palabras reservadas en Java.

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

Tabla 3.1: Palabras reservadas.

## 3.6 Tipos numéricos

Los tipos numéricos que se estudian en esta sección son tipos primitivos (elementales o básicos). Como todos los tipos básicos, éstos tienen los mismos tamaños y capacidades independientemente del entorno en el que se trabaje. Los detalles de la representación interna de los tipos numéricos se pueden encontrar en [For03].

### 3.6.1 Tipos enteros

En Java existen varios tipos de datos enteros con la misma representación interna (complemento a dos) y que se diferencian por la cantidad de memoria utilizada y, consecuentemente, por el rango de valores enteros que permite representar. Estos tipos son: `byte`, `short`, `int` y `long` cuyo tamaño en bits y rango de valores representado aparece en la tabla 3.2.

| Nombre             | Tamaño  | Rango                                       |
|--------------------|---------|---------------------------------------------|
| <code>byte</code>  | 8 bits  | [-128, 127]                                 |
| <code>short</code> | 16 bits | [-32768, 32767]                             |
| <code>int</code>   | 32 bits | [-2147483648, 2147483647]                   |
| <code>long</code>  | 64 bits | [-9223372036854775808, 9223372036854775807] |

Tabla 3.2: Tipos de números enteros.

Los valores o literales enteros pueden expresarse en los siguientes formatos, aunque el más utilizado es el decimal:

- *Formato decimal*: secuencia de dígitos precedida por el signo – para los negativos y, opcionalmente, + para los positivos.

- *Formato octal*: para indicar que una secuencia de dígitos está representada en el sistema octal, se le antepone el carácter cero (0). Por ejemplo, el número en octal 0301 es el equivalente al 193 en el sistema decimal ( $3*8^2 + 0*8^1 + 1*8^0 = 3*64 + 1 = 193$ ).
- *Formato hexadecimal*: para indicar esta representación, se anteponen los caracteres cero y equis (*0x*). Por ejemplo, el número en hexadecimal 0xC1 equivale a 193 en decimal ( $C*16^1 + 1*16^0 = 12*16 + 1 = 193$ ).

Por defecto, los valores enteros son de tipo **int**, así, si se escribe 123+0xA1, tanto el 123 como el 0xA1 (161 en decimal) son números que ocupan 32 bits y el resultado es 284 de tipo **int**. Si se desea forzar que un entero sea tomado como un **long** debemos añadir al final una ‘L’ o una ‘l’. Así, tanto 123L como 0xA1l ocupan 64 bits.

### 3.6.2 Tipos reales

Java dispone de los tipos **float** y **double** para trabajar con los números reales. La representación interna es la de punto flotante, diferente a la utilizada para los enteros. El conjunto de números reales es infinito y es imposible codificarlos todos en binario, teniendo que asumir una cierta imprecisión y trabajar con un número finito de valores.

Los números reales se caracterizan por dos magnitudes: la precisión y el intervalo de representación. La precisión es el número de dígitos significativos con los que se puede representar un número y el intervalo es la diferencia entre el mayor y el menor número que se pueden representar. La precisión de un número real depende del número de bits de su mantisa, mientras que el intervalo depende del número de bits de su exponente.

Así, el tipo **double** logra el doble de precisión que el **float**, entendiendo como precisión la cantidad de decimales. En estos tipos, lo más importante no es lo grande o pequeño que es el número a representar, sino su precisión. Si se evalúa la expresión 1-0.1-0.1-0.1-0.1-0.1 se obtiene el resultado 0.5000000000000001.<sup>2</sup> Otro tanto ocurre con la expresión 1-0.9 que se evalúa a 0.0999999999999998.

En la tabla 3.3 se muestra el tamaño, el rango de valores y la precisión de los dos tipos reales.

Por defecto, los valores reales son de tipo **double**. Podemos forzar un tipo **float** añadiendo al final del número el carácter ‘F’ o ‘f’ (0.1f). Los números reales pue-

---

<sup>2</sup>Sin embargo, si se evalúa la expresión 1+(-0.1-0.1-0.1-0.1-0.1) se obtiene el resultado 0.5, de lo que se deduce que la aritmética de los valores en coma flotante no cumple la propiedad asociativa. Esto es habitual en los lenguajes de programación y es una consecuencia de la representación finita de los valores reales.

| Nombre              | Tamaño  | Rango                              | Precisión    |
|---------------------|---------|------------------------------------|--------------|
| <code>float</code>  | 32 bits | [1.4E-45, 3.4028235E38]            | 7 decimales  |
| <code>double</code> | 64 bits | [4.9E-324, 1.7976931348623157E308] | 15 decimales |

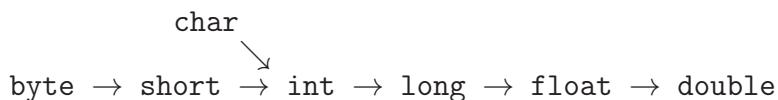
**Tabla 3.3:** Tipos reales.

den representarse con la notación decimal habitual en matemáticas, es decir, se escribe como una secuencia de dígitos que contiene un punto decimal. También se permite la notación científica, en la que  $x \cdot 10^y$  se escribe como `xEy`, como por ejemplo:

|             |         |          |               |
|-------------|---------|----------|---------------|
| decimal:    | -123.05 | 0.2243   | 0.00000000001 |
| científica: | 23.4e2  | -1.9E-18 | +1e-11        |

### 3.6.3 Compatibilidad y conversión de tipos

Como se explicó anteriormente, una variable sólo puede albergar un valor del tamaño y representación del tipo con el que se ha declarado, es decir, un valor del mismo tipo. Aunque Java sea un lenguaje fuertemente tipado, para facilitar el trabajo del programador, se proporcionan conversiones automatizadas entre aquellos tipos en los que no se compromete la representación interna de los datos. Son *conversiones de tipo implícitas* que se realizan de forma automática. Así, como un `byte` es más pequeño que un `short` y éste, más pequeño que un `int` y éste, a su vez, más pequeño que un `long`, el lenguaje convierte los valores de los tipos más pequeños a más grandes adaptándolos de forma automática. Esto ocurre siempre y cuando tengan la misma representación interna como es el caso de todos los tipos enteros o los dos tipos reales `float` y `double`. Java también automatiza la conversión de los tipos enteros a reales. En la figura 3.2 se representan las conversiones automáticas que realiza el lenguaje. El caso del tipo `char` se abordará en la sección 3.7.

**Figura 3.2:** Compatibilidad de tipos básicos.

Algunos ejemplos de conversión implícita o automática entre tipos numéricos son los siguientes:

```
byte e1 = 10; // conversión del int 10 a byte
short e2 = e1; // conversión de byte a short
int e3 = e2; // conversión de short a int
long e4 = e3; // conversión de int a long
float e5 = e4; // conversión de long a float
double e6 = e5; // conversión de float a double
```

Además de la conversión automática de tipos, en Java también existe una *conversión de tipos explícita* que fuerza la conversión entre tipos; es lo que se conoce como *casting* y tiene la siguiente sintaxis:

(tipo)expresión

Esta operación transforma el tipo de la **expresión** al tipo que aparece entre paréntesis sin importar que el tipo del que se trate sea numérico u otro tipo de los existentes en Java. En este caso, es el programador el que asume la responsabilidad de controlar los posibles errores que puedan producirse.

Nótese que cuando se fuerza la conversión de un real a entero, se trunca la parte decimal del real. La instrucción `int x = (int)12.98;` asigna a la variable `x` el valor entero 12 despreciando los decimales. El siguiente ejemplo propone una aplicación del casting explícito.

**Ejemplo 3.1.** Supóngase declaradas e inicializadas las variables siguientes:

```
double inf = 10.0; // cota inferior del rango de valores
double sup = 20.0; // cota superior del rango de valores
int cantInt = 2; // cantidad de intervalos
double valor = 14.9; // valor real
```

La siguiente secuencia de instrucciones permite averiguar el intervalo en el que está `valor`. Considérese que los intervalos se numeran desde cero.

```
// Cálculo y escritura del número del intervalo
double tamInt = (sup - inf)/cantInt;
int numInt = (int)((valor - inf)/tamInt);
System.out.print("Número del intervalo al que pertenece ");
System.out.println(valor + " : " + numInt);
```

```
// Cálculo y escritura del intervalo
double limInfInt = inf + numInt*tamInt,
 limSupInt = inf + (numInt+1)*tamInt;
System.out.println("[" + limInfInt + "," + limSupInt + "]");
```

Nótese el uso del *casting* al tipo entero para calcular el número del intervalo y asignarlo a la variable `numInt`. El resultado de ejecutar estas instrucciones sería:

| Salida Estándar                                               |
|---------------------------------------------------------------|
| Número del intervalo al que pertenece 14.9 : 0<br>[10.0,15.0[ |

### 3.6.4 Operadores aritméticos

Las operaciones, junto con los valores, las variables y los paréntesis forman el conjunto de elementos para construir expresiones en el lenguaje. Una expresión es de algún tipo numérico si al evaluarla se obtiene un valor de tipo numérico. El tipo de una operación es el tipo del valor que devuelve tras ser evaluada. En esta sección se estudian las operaciones aritméticas de los tipos numéricos que aparecen en la tabla 3.4. En las operaciones que se va a considerar se da cierto tipo de polimorfismo, es decir, son operaciones con el mismo nombre (o símbolo) pero que realizan acciones distintas en función de los operandos a los que se apliquen.<sup>3</sup>

| Operador | Descripción     | Operador | Descripción                 |
|----------|-----------------|----------|-----------------------------|
| +        | Suma o signo    | +=       | Suma y asignación           |
| -        | Resta o signo   | -=       | Resta y asignación          |
| *        | Multiplicación  | *=       | Multiplicación y asignación |
| /        | División        | /=       | División y asignación       |
| %        | Módulo          | %=       | Módulo y asignación         |
| ++       | Incremento en 1 | --       | Decremento en 1             |

**Tabla 3.4:** Operadores aritméticos.

#### Operadores aritméticos simples

La suma, resta, multiplicación, división y módulo o resto de la división, están definidas tanto para los tipos enteros como los reales, aunque debido a su representación interna, las acciones internas de cálculo que realizan son distintas. Cuando se evalúan el tipo de su valor resultante es el mismo que el de sus operandos.

---

<sup>3</sup>Por ejemplo, el operador división (/) actúa de forma distinta según que los operandos sean enteros o reales.

**Ejemplo 3.2.** En este ejemplo pueden observarse algunas expresiones simples y el resultado de su evaluación.

|         | Expresión  | Resultado | Expresión  | Resultado |
|---------|------------|-----------|------------|-----------|
| Enteros | $3+5$      | 8         | $2*6$      | 12        |
|         | $7/2$      | 3         | $7\%2$     | 1         |
| Reales  | $3.5+5.6$  | 9.1       | $3.1*2.0$  | 6.2       |
|         | $15.0/2.0$ | 7.5       | $7.0\%2.0$ | 1.0       |

Nótese que el valor de la expresión  $7/2$  es el entero 3, ya que los operandos de la división son enteros y cuando se realiza la división entera evalúa, a su vez, a un entero. Si lo que se desea es efectuar la división real hay que indicar que al menos uno de los operandos es de tipo real. El lenguaje permite escribir expresiones con esta mezcla de tipos y los convierte automáticamente al tipo superior. Así, si se escribe  $7/2.0$ ; el dividendo se convierte al tipo `double` y lo que realmente se evalúa es  $7.0/2.0$  al valor 3.5.

Al evaluar una expresión aritmética, se tiene que prestar especial atención a la división entera (entre enteros) ya que si el divisor es cero, se aborta la ejecución del programa y se emite el siguiente mensaje, correspondiente a una *excepción aritmética*, que normalmente aparecerá en la salida estándar:

```
java.lang.ArithmetricException: / by zero
```

La división real con divisor cero (0.0) no provoca ninguna interrupción brusca del programa y como resultado devuelve el valor  $\infty$ . A continuación se ilustran algunos ejemplos:

| Expresión  | Resultado              |
|------------|------------------------|
| $5.0/0.0$  | <code>Infinity</code>  |
| $-5.0/0.0$ | <code>-Infinity</code> |
| $0.0/0.0$  | <code>NaN</code>       |

donde el resultado `NaN` es el acrónimo de *Not a Number* (*No un Número* en castellano) y se devuelve para indicar que el resultado está indefinido o no se puede representar.

**Ejemplo 3.3.** En este ejemplo se ilustra el comportamiento de la operación que calcula el resto de la división, tanto para números enteros como reales. ¿De qué depende el signo del resultado? ¿Qué expresión se podría usar para saber si un número entero es par o impar? ¿Y para saber si un entero es múltiplo de otro entero? ¿Sería igual de fiable la contestación a la cuestión anterior para números reales? ¿A qué se debe que haya un real con tantos decimales?

| Enteros   |           | Reales      |                    |
|-----------|-----------|-------------|--------------------|
| Expresión | Resultado | Expresión   | Resultado          |
| 5 % 2     | 1         | 6.5 % 2.5   | 1.5                |
| -5 % 2    | -1        | -6.5 % 2.5  | -1.5               |
| 5 %-2     | 1         | 6.5 %-2.5   | 1.5                |
| -5 %-2    | -1        | -6.5 %-2.5  | -1.5               |
| 64 % 8    | 0         | 7.5 % 2.5   | 0.0                |
| 13 % 20   | 13        | 5.66 % 20.0 | 5.66               |
| 13 % 5    | 3         | 60 % 4.2    | 1.1999999999999975 |

El operador unario + se usa de forma opcional delante de los números y el operador unario - delante de los números para obtener su inverso. Por ejemplo, en el real escrito en notación científica  $+31416e-4$ , el símbolo + se usa opcionalmente, mientras que el signo - es necesario para que este número sea equivalente a 3.1416 en representación decimal.

### Operadores aritméticos compuestos

En los lenguajes de programación es muy habitual reutilizar las variables para guardar nuevos valores y reducir la cantidad de memoria usada. Por ejemplo, si se define la variable de tipo entero `numAlumnos` con un valor inicial arbitrario 50 mediante la instrucción `int numAlumnos = 50;` y se quiere decrementar su contenido en siete unidades. La instrucción de asignación `numAlumnos = numAlumnos-7;` se ejecuta evaluando primero la expresión de la derecha `numAlumnos-7`. Lo cual requiere restar al contenido de la variable el número 7. El resultado obtenido es 43 y se guarda en la misma variable. Java proporciona una forma abreviada de guardar en una variable el valor resultante de operar con el valor inicial de la misma sin tener que escribir el identificador de la variable dos veces. La anterior instrucción se puede escribir de forma concisa: `numAlumnos-=7;`. En la columna de la derecha de la tabla 3.4 se muestran los cinco operadores compuestos disponibles para los tipos numéricos.

**Ejemplo 3.4.** La siguiente secuencia de instrucciones transforma cierta cantidad de segundos (`segundos`) en días, horas, minutos y segundos restantes:

```
long segundos = 765432; // cantidad de segundos
long dias = segundos/(24*60*60);
segundos %= 24*60*60;
System.out.println("Días: " + dias);
System.out.println(" (Segundos restantes: " + segundos + ")");
long horas = segundos/(60*60);
segundos %= 60*60;
System.out.println("Horas: " + horas);
System.out.println(" (Segundos restantes: " + segundos + ")");
```

```

long minutos = segundos/60;
segundos %= 60;
System.out.println("Minutos: " + minutos);
System.out.println("Segundos restantes: " + segundos);

```

La cantidad de segundos iniciales se guarda en la variable **segundos** de tipo **long**. La cantidad de días se calcula a continuación evaluando la expresión **segundos/(24\*60\*60)** ya que la cantidad de segundos que hay en un día son  $(24*60*60)$ ; como se realiza la división entera se desprecian los decimales. El siguiente paso consiste en descontar de la variable **segundos** todos aquellos segundos utilizados para los días calculados. Esta cantidad de segundos se corresponde con el número de días multiplicados por los segundos que tiene un día. La expresión utilizada en Java podría ser **segundos-días\*(24\*60\*60)** aunque es preferible su equivalente **segundos%(24\*60\*60)**. En ambas opciones, el resultado se puede guardar reutilizando la variable **segundos**. En la primera opción, la instrucción resultante sería:

```
segundos = segundos-días*(24*60*60);
```

que puede abreviarse utilizando el operador compuesto **-=** quedando la instrucción **segundos-=días\*(24\*60\*60)**. Para la segunda opción, la instrucción es:

```
segundos = segundos%(24*60*60);
```

que también puede abreviarse utilizando el operador **%=**.

Después, se escriben los días calculados y la cantidad de segundos que quedan aún por asignar. Esta última es menor que la cantidad de segundos que tiene un día. El cálculo del número de horas se realiza de la misma forma teniendo en cuenta que una hora tiene  $(60*60)$  segundos. Y finalmente se repite lo mismo con los segundos de un minuto (60). La última instrucción muestra por pantalla los segundos restantes que no se han podido asignar a las otras medidas de tiempo. El resultado de la ejecución de este fragmento de código es como sigue:

| Salida Estándar                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------|
| Días: 8<br>(Segundos restantes: 74232)<br>Horas: 20<br>(Segundos restantes: 2232)<br>Minutos: 37<br>Segundos restantes: 12 |

### Operadores de incremento y decremento en uno

Existen dos operadores unarios (se aplican sobre un único operando) para incrementar en una unidad el valor de una variable de tipo numérico. Estos operadores son `++` para el incremento positivo y `--` para el incremento negativo. Ambos pueden ser prefijos o posfijos, es decir, se escriben delante o detrás del operando, siendo éste último una variable de tipo numérico. Como operadores prefijos, primero realizan la operación y después devuelven el resultado. Como posfijos, primero devuelven el valor sin modificar y después realizan el incremento.

En la tabla 3.5 figuran varios ejemplos de uso de los mismos. Esta tabla es un seguimiento o *traza* de los valores contenidos en las variables tras ejecutarse las instrucciones de la primera columna.

En la segunda y tercera columna aparecen los valores de las variables utilizadas para ilustrar la funcionalidad de los operadores. En la primera instrucción se declara e inicializa la variable `a`. Desde la segunda hasta la quinta, el comportamiento de las instrucciones solo afecta a la variable `a` incrementándola o decrementándola, según el caso, y el valor resultante no es utilizado. Como puede observarse, el hecho de ser prefijo o posfijo no afecta al resultado. En las cuatro últimas instrucciones, el valor resultante se asigna a la variable `b` y puede observarse la diferencia de los valores resultantes según el operador sea prefijo, en cuyo caso es el valor de la variable `a` después de ser incrementado o decrementado; o posfijo, en cuyo caso es el valor de la variable `a` antes de ser incrementado o decrementado.

| Instrucción               | a | b |
|---------------------------|---|---|
| <code>int a = 0;</code>   | 0 |   |
| <code>a++;</code>         | 1 |   |
| <code>++a;</code>         | 2 |   |
| <code>a--;</code>         | 1 |   |
| <code>--a;</code>         | 0 |   |
| <code>int b = a++;</code> | 1 | 0 |
| <code>b = ++a;</code>     | 2 | 2 |
| <code>b = a--;</code>     | 1 | 2 |
| <code>b = --a;</code>     | 0 | 0 |

**Tabla 3.5:** Traza de ejecución de incrementos y decrementos en uno.

### 3.6.5 Desbordamiento

Realizar operaciones con números puede producir que el resultado exceda la capacidad de representación del tipo. En ese caso, se habla de *desbordamiento* (*overflow* en inglés).

En la aritmética real los desbordamientos se producen hacia infinito (*overflow*) o hacia cero (*underflow*). Cuando el resultado de una operación está fuera de rango, se obtiene **Infinity** o **-Infinity**. Por ejemplo, en el tipo **float** la evaluación de la expresión `1e38f*10` resulta en el valor **Infinity**, y lo mismo ocurre con el tipo **double** al evaluar la expresión `1e308*10`. Los infinitos también se propagan en la evaluación de expresiones. El resultado de evaluar la expresión `(5.0/0.0)+166.386` devuelve **Infinity** al sumar el valor **Infinity** resultante de la división por cero.

### 3.7 Tipo carácter

El tipo carácter se utiliza para representar letras latinas minúsculas y mayúsculas, números, signos de puntuación, caracteres de control y caracteres de distintos alfabetos como el griego, cirílico, hebreo, mandarín, árabe, etc.

Un literal de tipo carácter se representa internamente como un valor entero positivo pero sin la representación en complemento a dos, ya que no se requieren valores negativos. Así pues, los valores de este tipo no son enteros.

La asociación de cada carácter a un código numérico determinado se hace siguiendo algún *estándar de codificación de caracteres* que, en el caso del Java, es el denominado Unicode. Mediante el Unicode es posible representar varios millones de caracteres diferentes.<sup>4</sup>

A su vez, cada uno de los símbolos Unicode, denominados *puntos de código*, se pueden representar físicamente de varias maneras, esto es, como diferentes secuencias de bits. Las representaciones o codificaciones más habituales utilizadas para ello son las denominadas UTF-8, UTF-16 y UTF-32. Según que representación se utilice puede ocurrir que un mismo código Unicode, correspondiente a un determinado carácter, se represente físicamente como una secuencia de bits distinta y más o menos larga.

Internamente, el Java utiliza UTF-16, por lo que se puede decir que el Java codifica sus caracteres en Unicode siguiendo la representación UTF-16.<sup>5</sup> Sin embargo, desde un punto de vista externo, configurando adecuadamente el lenguaje, es posible organizar los programas para trabajar con *flujos de caracteres* que sigan casi cualquier codificación existente.

Se recomienda visitar la *URL* <http://es.wikipedia.org/wiki/Unicode> o la de la propia organización Unicode: <http://unicode.org> para más información.

---

<sup>4</sup>La versión más reciente del estándar, la 6.0, codifica algo menos de un millón de caracteres reales, prácticamente todos los correspondientes a los lenguajes conocidos.

<sup>5</sup>En UTF-16 cada carácter ocupa habitualmente dos bytes aunque, en casos excepcionales puede llegar a ocupar hasta cuatro bytes.

Por motivos de compatibilidad histórica, ya que los ordenadores tienen su origen en el mundo anglosajón, los 256 primeros caracteres del Unicode coinciden con los del estándar *ASCII/ANSI* de 8 bits de los que en la tabla 3.6 se muestran los codificables con 7 bits para los primeros 128 caracteres.<sup>6</sup> Los 128 restantes del estándar *ASCII* esto es, los caracteres Unicode desde el 128 hasta el 255, se utilizan para codificar caracteres específicos para diversos alfabetos europeos. Naturalmente, mediante el resto de caracteres Unicode es posible representar casi cualquier símbolo existente en algún lenguaje así, por ejemplo, mediante los códigos existentes entre el 1536 y el 1791 se codifican los símbolos básicos del árabe.

La tabla 3.6 organiza visualmente los caracteres siguiendo una numeración hexadecimal, aunque para facilitar la lectura figura debajo de cada carácter el correspondiente código en decimal. Así en la línea 6, columna E se encuentra el carácter ‘n’ cuyo código en hexadecimal es 6E equivalente a 110 en decimal.

Obsérvese que las letras y dígitos tienen códigos contiguos, que las mayúsculas preceden a las minúsculas y que la distancia entre cualquier carácter en mayúscula y la minúscula que le corresponde es siempre la misma.

Además, aparecen representados los símbolos de puntuación fundamentales y algunos símbolos matemáticos como son, por ejemplo, los de los operadores aritméticos.

Por último, obsérvese que hay un grupo de códigos al inicio de la tabla, que tienen un significado especial y que, a menudo, representan acciones heredadas de su uso en la comunicación con teletipos, pero que también se utilizan en los ordenadores actuales; así aparecen, por ejemplo, EOT y ACK para representar final y reconocimiento de transmisión o BCK y DEL que representan retroceso y borrado, respectivamente.

Los literales de tipo carácter se escriben entre comillas simples, y las variables que los almacenan se declaran utilizando la palabra reservada **char**.

**Ejemplo 3.5.** En este ejemplo se inicializan algunas variables de tipo **char**.

```
char aMayuscula = 'A',
 zMinuscula = 'z',
 interrogacion = '?',
 digito0 = '0';
 esPacioEnBlanco = ' ';
```

---

<sup>6</sup>El juego de caracteres *ASCII-7* tiene su origen en los años 40, utilizándose en los teletipos.

|          | <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> | <b>A</b> | <b>B</b> | <b>C</b> | <b>D</b> | <b>E</b> | <b>F</b> |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>0</b> | NUL      | SOH      | STX      | ETX      | EOT      | ENO      | ACK      | BEL      | BS       | TAB      | LF       | VT       | FF       | CR       | SO       | SI       |
|          | 0        | 1        | 2        | 3        | 4        | 5        | 6        | 7        | 8        | 9        | 10       | 11       | 12       | 13       | 14       | 15       |
| <b>1</b> | DLE      | DC1      | DC2      | DC3      | DC4      | NAK      | SYN      | ETB      | CAN      | EM       | SUB      | ESC      | FS       | GS       | RS       | US       |
|          | 16       | 17       | 18       | 19       | 20       | 21       | 22       | 23       | 24       | 25       | 26       | 27       | 28       | 29       | 30       | 31       |
| <b>2</b> | !        | "        | #        | \$       | %        |          | '        | (        | )        | *        | +        | ,        | -        | .        | /        |          |
|          | 32       | 33       | 34       | 35       | 36       | 37       | 38       | 39       | 40       | 41       | 42       | 43       | 44       | 45       | 46       | 47       |
| <b>3</b> | <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> | :        | ;        | <        | =        | >        | ?        |
|          | 48       | 49       | 50       | 51       | 52       | 53       | 54       | 55       | 56       | 57       | 58       | 59       | 60       | 61       | 62       | 63       |
| <b>4</b> | @        | A        | B        | C        | D        | E        | F        | G        | H        | I        | J        | K        | L        | M        | N        | O        |
|          | 64       | 65       | 66       | 67       | 68       | 69       | 70       | 71       | 72       | 73       | 74       | 75       | 76       | 77       | 78       | 79       |
| <b>5</b> | P        | Q        | R        | S        | T        | U        | V        | W        | X        | Y        | Z        | [        | \        | ]        | ^        | _        |
|          | 80       | 81       | 82       | 83       | 84       | 85       | 86       | 87       | 88       | 89       | 90       | 91       | 92       | 93       | 94       | 95       |
| <b>6</b> | `        | a        | b        | c        | d        | e        | f        | g        | h        | i        | j        | k        | l        | m        | n        | o        |
|          | 96       | 97       | 98       | 99       | 100      | 101      | 102      | 103      | 104      | 105      | 106      | 107      | 108      | 109      | 110      | 111      |
| <b>7</b> | p        | q        | r        | s        | t        | u        | v        | w        | x        | y        | z        | {        |          | }        | ~        | DEL      |
|          | 112      | 113      | 114      | 115      | 116      | 117      | 118      | 119      | 120      | 121      | 122      | 123      | 124      | 125      | 126      | 127      |

Tabla 3.6: Codificación ASCII (7 bits), 128 primeros caracteres Unicode.

Los literales del tipo carácter también pueden representarse utilizando directamente el código Unicode correspondiente, usando para ello la sintaxis \ucódigo de cuatro caracteres en hexadecimal. Por ejemplo la instrucción:

```
System.out.println("¡Hola mundo! \u00A1Hello\u0020world\u0021");
```

escribe en la salida estándar:

|                            |                 |
|----------------------------|-----------------|
| ¡Hola mundo! ¡Hello world! | Salida Estándar |
|----------------------------|-----------------|

Además, como los literales y variables de este tipo se codifican utilizando números naturales, puede usarse la aritmética de enteros y la conversión forzada de tipos para operar con ellos.

**Ejemplo 3.6.** En la segunda instrucción del segmento de código siguiente, el contenido de la variable ch1 se convierte a entero para obtener su código Unicode. A este código se le suma 1 para obtener el siguiente código de la tabla que es convertido de nuevo a tipo carácter. El resultado, el carácter ‘B’, es almacenado finalmente en la variable letraB.

A continuación, el carácter asignado a `letraC` es el carácter ‘C’ obtenido sumando 1 al carácter ‘B’.<sup>7</sup>

Finalmente, la variable `letraN` que almacena inicialmente la letra ‘N’ ve incrementado su valor por la distancia entre los códigos de las letras minúsculas y mayúsculas, resultando modificada la propia variable que pasa a contener la letra ‘n’.

```
char ch1 = 'A',
char letraB = (char)((int)ch1 + 1);
System.out.println("Letra: " + letraB);

char letraC = 'B' + 1;
System.out.println(((int)letraC) + " Letra: " + letraC);

char letraN = '\u006E';
letraN += 'A' - 'a';
System.out.println("Letra: " + '\u006E' + " y " + letraN);
```

El resultado que se muestra por la salida estándar es el siguiente:

| Salida Estándar                         |  |
|-----------------------------------------|--|
| Letra: B<br>67 Letra: C<br>Letra: n y N |  |

Para representar caracteres de control que no son visibles pero tienen un efecto especial, se usan *secuencias de escape*. Estas secuencias consisten en la barra de dividir invertida ‘\’ seguida de un carácter al que le dan una funcionalidad distinta de la esperada. En la tabla 3.7 se muestra una lista con las secuencias de escape más habituales junto con el significado de las mismas.

Los cuatro primeros elementos de la lista, cuando se escriben, equivalen al carácter que se indica en la columna descripción.

Por el contrario, los tres últimos elementos se requieren para modificar la funcionalidad que esos caracteres tienen para Java. La secuencia ‘\’ se requiere para poder representar el carácter comilla simple ‘’’ eliminando su significado de delimitador de caracteres otorgándole el significado de carácter. Las dobles comillas se usan como delimitadores de cadenas de caracteres y la secuencia de escape ‘\"’ les devuelve el significado de carácter. Y por último, la secuencia ‘\\’ cambia el significado de barra de inicio de secuencia de escape a simple carácter.

<sup>7</sup>No es posible realizar la misma acción con la instrucción `char letraC = letraB+1;` ya que al sumar a `letraB` una cantidad, se corre el riesgo de desbordamiento.

| Secuencia de escape | Descripción                                 |
|---------------------|---------------------------------------------|
| \t                  | Tabulador                                   |
| \n                  | Avance de línea ( <i>new line</i> )         |
| \r                  | Retorno de carro ( <i>carriage return</i> ) |
| \b                  | Retroceso ( <i>backspace</i> )              |
| \'                  | Comillas simples                            |
| \"                  | Comillas dobles                             |
| \\                  | Barra invertida                             |

**Tabla 3.7:** Secuencias de escape.

**Ejemplo 3.7.** En este ejemplo se muestra el resultado de concatenar una cadena con un carácter. La cadena está formada por los dos caracteres " y \ que se escribe como: "\\"\\. El carácter está escrito entre comillas simples: '\'. Usando el operador de concatenación +, la expresión resultante es:

"\\\" + '\'

Como la operación de concatenación sólo admite cadenas en sus argumentos, el carácter '\'' se convierte automáticamente al tipo superior cadena de caracteres: "\\". El resultado de la concatenación es la cadena formada por los tres caracteres: " \\'.

## 3.8 Tipo lógico

El lenguaje Java implementa un álgebra booleana bivaluada (con dos valores de verdad) mediante el tipo de datos **boolean**. Los dos únicos valores de verdad de este tipo se representan con las constantes **true** para el valor verdadero y **false** para el valor falso. Las variables de este tipo se definen usando la palabra reservada **boolean**.

**Ejemplo 3.8.** En este ejemplo se definen e inicializan dos variables de tipo lógico.

```
boolean encontrado = false, estaCompleto = true;
```

Se dice que una expresión es de tipo lógico si se evalúa a los valores lógicos. Las expresiones lógicas o de tipo **boolean** se construyen a partir de operadores relacionales con argumentos de tipo básico y operadores lógicos con argumentos de tipo lógico.

### 3.8.1 Operadores relacionales

En la tabla 3.8 se muestran los operadores relacionales con el mismo significado que tienen en matemáticas.

| Operador           | Operación         |
|--------------------|-------------------|
| <code>==</code>    | Igual             |
| <code>!=</code>    | Distinto          |
| <code>&lt;</code>  | Menor que         |
| <code>&lt;=</code> | Menor o igual que |
| <code>&gt;</code>  | Mayor             |
| <code>&gt;=</code> | Mayor o igual que |

Tabla 3.8: Operadores relacionales.

**Ejemplo 3.9.** En este ejemplo se muestran algunas expresiones lógicas con estos operadores y el resultado de su evaluación se asigna a una variable lógica. En las cinco primeras definiciones de variables lógicas, `b1` toma el valor `false`, `b2` el valor `true`, `b3` el valor `false`, `b4` el valor `true` ya que el código Unicode del carácter ‘a’ es menor que el de ‘b’, y `b5` toma el valor `false`. Recuérdese que la asignación `=` es un operador que devuelve el valor asignado. Así, a `b2` y `b3` se les asigna el valor `true`.

```
int x = 5;
boolean b1 = 6 == x,
 b2 = x <= 7,
 b3 = (4 + x) > 10,
 b4 = 'a' < 'b',
 b5 = true == false;
b2 = b3 = 5.5 != 6.3;
```

### 3.8.2 Operadores lógicos

En la tabla 3.9 se muestran los operadores lógicos vistos en este capítulo. Estos operadores reciben valores lógicos como argumentos y a su vez devuelven un valor lógico. Normalmente, sus argumentos son expresiones relacionales o métodos.

Los operadores lógicos y los cortocircuitados se diferencian en que los primeros evalúan necesariamente sus dos argumentos, mientras que los segundos no continúan con la evaluación si se obtiene el resultado antes de evaluar toda la expresión. Por ejemplo, en la expresión `5 < 3 && 5 < x` no se evalúa el segundo argumento de la conjunción, pues la evaluación del primero resulta `false` y, por lo tanto, el resultado de la conjunción ya es falso sin necesitar evaluar su segundo argumento.

| Operador | Operación                  |
|----------|----------------------------|
| !        | Negación                   |
| &        | Conjunción lógica          |
|          | Disyunción lógica          |
| ~        | Disyunción exclusiva       |
| &&       | Conjunción cortocircuitada |
|          | Disyunción cortocircuitada |

**Tabla 3.9:** Operadores lógicos.

En la tabla 3.10 se muestran los valores resultantes de los operadores lógicos para cada combinación de sus argumentos ( $x$  e  $y$ ).

| x       | y       | $x \&& y$ | $x    y$ | $x ^ y$ | $!x$  |
|---------|---------|-----------|----------|---------|-------|
| $x & y$ | $x   y$ |           |          |         |       |
| true    | true    | true      | true     | false   | false |
| true    | false   | false     | true     | true    | false |
| false   | true    | false     | true     | true    | true  |
| false   | false   | false     | false    | false   | true  |

**Tabla 3.10:** Significado de los operadores lógicos.

Los operadores relacionales y lógicos pueden usarse conjuntamente para formar expresiones lógicas.

**Ejemplo 3.10.** En este ejemplo la expresión es cierta si el valor contenido por la variable de tipo entero  $x$  es par y está comprendido en los rangos  $[0, 5[$  y  $[10, 20]$ .

```
x%2 != 1 && (x >= 0 && x < 5 || x >= 10 && x <= 20)
```

¿Se corre algún riesgo si la variable es de tipo real?

### 3.9 Precedencia de operadores

En Java se aplican las reglas de precedencia de operadores usuales: los paréntesis preceden a los operadores multiplicativos, que se ejecutan antes que los aditivos. En la tabla 3.11 se muestran los grupos de precedencia para los operadores en Java. Cuanto menor es el número del grupo mayor precedencia tiene. Si en una expresión aparecen operaciones del mismo grupo, se evalúan con asociatividad por la izquierda, es decir, se evalúan de izquierda a derecha. La precedencia puede alterarse con el uso habitual de los paréntesis.

| Grupo | Clasificación               | Operadores                     |
|-------|-----------------------------|--------------------------------|
| 0     | Paréntesis                  | ()                             |
| 1     | Operadores unarios posfijos | (parametros), expr++, expr--   |
| 2     | Operadores unarios prefijos | ++expr, --expr, +expr, -expr ! |
| 3     | Creación o conversión       | new, (tipo) expr               |
| 4     | Multiplicación              | *, /, %                        |
| 5     | Suma                        | +, -                           |
| 6     | Relacionales                | <, <=, >, >=,                  |
| 7     | Igualdad                    | ==, !=                         |
| 8     | Conjunción lógica           | &                              |
| 9     | Disyunción exclusiva        | ~                              |
| 10    | Disyunción lógica           |                                |
| 11    | Conjunción cortocircuitada  | &&                             |
| 12    | Disyunción cortocircuitada  |                                |
| 13    | Operador ternario           | ? :                            |
| 14    | Asignación                  | =, +=, -=, *=, /=, %=          |

Tabla 3.11: Precedencia de los operadores.

**Ejemplo 3.11.** En este ejemplo se aprecia el efecto de la asociatividad por la izquierda, la precedencia y como puede alterarse esta última con el uso de los paréntesis. La expresión:

5.4 < 36%30 || 3\*4-6<7 && 32 >= 'a';

se evalúa a `true`, mientras que la expresión

(5.4 < 36%30 || 3\*4-6<7) && 32 >= 'a';

se evalúa a `false`.

## 3.10 Bloques de instrucciones

El lenguaje Java es un lenguaje orientado a *bloques*, lo que significa que la sintaxis del lenguaje está basada en dicho concepto. Las instrucciones de un programa, como ya se ha visto, aparecen de forma consecutiva. Es decir, se trata de una *composición secuencial* de instrucciones. Dichas instrucciones se pueden agrupar, constituyendo un *bloque* de instrucciones.

Un *bloque* es una secuencia de instrucciones comprendidas entre los símbolos de llaves, { y }. El propósito de un bloque es agrupar una secuencia de instrucciones en una sola instrucción. Así, un bloque se puede utilizar en cualquier lugar en el que una instrucción simple se pueda utilizar.

Los bloques pueden *anidarse* unos dentro de otros. De forma que se habla de *bloques externos* (porque contienen a otros) o *internos* (cuando están contenidos dentro de otros). El siguiente es un ejemplo de bloques anidados (el bloque que va desde la línea 1 a la 11 es un bloque externo que contiene al bloque interno que va desde la línea 3 a la 8):

```
1 {
2 int dia = 10, mes = 12, año = 2011;
3 {
4 // int dia = 30; si se descomenta --> error de compilación
5 double temperatura = 36.8;
6 System.out.println(dia); // se escribe 10
7 System.out.println(mes); // se escribe 12
8 }
9 System.out.println(dia); // se escribe 10
10 // temperatura no se puede referenciar aquí
11 }
```

En Java, las variables se deben definir en el bloque en el que se utilizan. El *ámbito* de una variable es la parte del bloque en el que la variable es conocida y se puede utilizar.

Una variable declarada dentro de un bloque es completamente inaccesible e invisible desde fuera de ese bloque. Dentro de un bloque se pueden utilizar tanto las variables definidas en el mismo, como en cualquier otro bloque externo que lo comprenda.

Una variable se dice que es *local* en el bloque que se define y *global* para los bloques internos a éste. Java no permite que un mismo identificador se utilice para definir diferentes variables en bloques anidados. En el ejemplo anterior, las variables **dia**, **mes** y **año** (en la línea 2) son locales para el bloque externo y globales para el bloque interno, y la variable **temperatura** (en la línea 5) es local para el bloque interno y no puede utilizarse fuera del mismo. Si se declara de nuevo la variable **dia** en el bloque interno (línea 4) provocará un error de compilación.

A continuación se describen ciertas reglas relacionadas con el concepto de bloque y el uso de variables:

- Todas las variables definidas en el mismo bloque deben tener nombres diferentes.
- Una variable definida en un bloque es conocida desde su definición hasta el final del bloque. Como caso particular, una variable definida en un bloque es conocida en todos los bloques internos a éste.
- Las variables se deben definir al comienzo del bloque más interno en el que se utilizan.

## 3.11 Problemas propuestos

1. Hacer una traza del siguiente programa en Java

```
public class Prueba {
 public static void main (String[] args) {
 double x, y;
 x = 5.0;
 y = 7/9 * (x + 1);
 System.out.println("x = " + x + " y = " + y);
 }
}
```

2. Escribir una instrucción en Java tal que, suponiendo que las variables **x**, **y**, **z** son de tipo **double**, asigne a **z** el valor que indica la fórmula:

$$z = \frac{1 + \frac{x^2}{y}}{\frac{x^3}{1 + y}}$$

3. ¿A qué valor se evalúan las siguientes expresiones?

| Nº | Expresión    | Nº | Expresión     |
|----|--------------|----|---------------|
| 1  | 123456/10    | 5  | 123456 %10    |
| 2  | 123456/100   | 6  | 123456 %100   |
| 3  | 123456/1000  | 7  | 123456 %1000  |
| 4  | 123456/10000 | 8  | 123456 %10000 |

A la vista de los resultados obtenidos, ¿qué se puede concluir?

4. Dadas las siguientes expresiones, en donde **a** y **b** son variables enteras que toman los siguientes valores **a** = 5 y **b** = 3, indicar:

- a) El resultado al que se evalúan actualmente.
- b) La expresión modificada para que el resultado sea el que se indica como correcto.

| Nº | Expresión     | Resultado correcto |
|----|---------------|--------------------|
| 1  | 3/4*(a*a-b)   | 16.5               |
| 2  | a/b*1000+304  | 1970.6666666666667 |
| 3  | (100/a+b/2)*5 | 107.5              |

5. Escribir una instrucción de asignación en Java tal que a partir de una temperatura en grados Celsius (**celsius** de tipo **double**) obtenga su equivalente en grados Fahrenheit (**fahrenheit** de tipo **double**), aplicando la fórmula  ${}^{\circ}\text{F} = (9/5) * {}^{\circ}\text{C} + 32$ .

6. Escribir una instrucción de asignación en Java tal que a partir de una temperatura en grados Fahrenheit (**fahrenheit** de tipo **double**) obtenga su equivalente en grados Celsius (**celsius** de tipo **double**), aplicando la fórmula  ${}^{\circ}\text{C} = (5/9) * ({}^{\circ}\text{F} - 32)$ .
7. Escribir instrucciones de asignación en Java para:
  - a) Calcular en una variable **s** la superficie ( $4\pi r^2$ ) de una esfera a partir del valor del radio **r** (supóngase que es un valor positivo).
  - b) Calcular en una variable **v** el volumen ( $\frac{4}{3}\pi r^3$ ) de una esfera a partir del valor del radio (supóngase que es un valor positivo).
  - c) Calcular en una variable **v** el volumen de una esfera a partir del valor de su superficie **s** (supóngase que es un valor positivo).
8. Escribir una instrucción de asignación en Java tal que a partir de una cantidad (positiva) en pesetas (**pesetas** de tipo **int**) obtenga su equivalente en euros (**euros** de tipo **double**), sabiendo que 1€ son 166.386 pesetas.
9. Hacer una traza del siguiente programa:

```
public class TestOperador {
 public static void main(String[] args) {
 int a = 12, b = 8, c = 6;
 System.out.println(a + " " + b + " " + c);
 a = c;
 System.out.println(a + " " + b + " " + c);
 c += b;
 System.out.println(a + " " + b + " " + c);
 a = b + c;
 System.out.println(a + " " + b + " " + c);
 a++;
 b++;
 System.out.println(a + " " + b + " " + c);
 c = a++ + ++b;
 System.out.println(a + " " + b + " " + c);
 } // del main
} // de TestOperador
```

10. Una empresa de transporte por carretera ha adquirido vehículos nuevos que viajan más rápido que los antiguos. Les gustaría conocer cómo afectará esto a la duración de los viajes. Supóngase que la reducción media que se consigue del tiempo total de viaje es del 15 %. Escribir las instrucciones necesarias en Java tales que a partir de ciertos valores dados de horario de salida (**horaSalida** y **minSalida** de tipo **int**) y llegada antiguo (**horaLlegada** y **minLlegada** de tipo **int**) –siendo la salida anterior a la llegada y suponiendo horas (de 0 a 23) y minutos (de 0 a 59) correctos–, para trayectos realizados en el mismo día, calcule el nuevo horario de llegada y muestre en pantalla el

nuevo tiempo de viaje y la nueva hora de llegada. Un ejemplo de ejecución considerando como hora de salida 4:55 y hora de llegada 6:30 sería:

**Salida Estándar**

Duración inicial: 95 minutos (1h y 35m)

Nueva hora de llegada: 6

Nuevos minutos de llegada: 15

Duración del viaje: 80 minutos (1h y 20m)

11. Determinar el valor, **true** o **false**, de cada una de las siguientes expresiones lógicas, asumiendo que el valor de la variables **cont** y **limite** (de tipo **int**) es 10 y 20, respectivamente.
  - a) `(cont == 0) && (limite < 20)`
  - b) `(limite >= 20) || (cont < 5)`
  - c) `((limite/(cont-10)) > 7) || (limite < 20)`
  - d) `(limite<=20) || ((limite/(cont-10)) > 7)`
  - e) `((limite/(cont-10)) > 7) && (limite < 0)`
  - f) `(limite < 0) && ((limite/(cont-10)) > 7)`
12. Si se ejecuta la siguiente secuencia de instrucciones, ¿se produce una división por cero?

```
int j = -2;
boolean b = (j > 0) && (1/(j+2) > 10);
```

## Más información

- [CGo03] J. Carretero, F. García, y otros. *Problemas resueltos de programación en lenguaje Java*. Thomson, 2003. Capítulos 2 y 6.
- [Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011. URL: <http://math.hws.edu/javanotes/>. Capítulo 2 (2.2 y 2.5).
- [For03] B.A. Forouzan. *Introducción a la Ciencia de la Computación, de la manipulación de datos a la teoría de la computación*. Thomson, 2003. Capítulos 2, 3 y 4.
- [Ora11d] Oracle. *The Java<sup>TM</sup> Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Language Basics.

## Capítulo 4

# Tipos de datos: clases y referencias

Además de poder trabajar con variables cuyos valores son de tipos primitivos, en Java es posible manipular, definiéndolos y utilizándolos, elementos cuyos valores no son tipos primitivos. Cuando un valor no pertenece a un tipo primitivo, se dice que se trata de un tipo compuesto o estructurado. Las características de los valores de este tipo se definen mediante clases. En Java, cualquier elemento perteneciente a un tipo no primitivo, definido por lo tanto mediante una clase, es un objeto.

Valores no primitivos u objetos han sido utilizados ya anteriormente; en particular, se han utilizado variables así en alguno de los ejemplos del capítulo 2, como en el programa de la figura 2.3 donde, por ejemplo, se tenía el segmento de código siguiente

```
...
//Crear un Circulo de radio 50, amarillo, con centro en (100,100)
Circulo c1 = new Circulo(50,"amarillo",100,100);
//Añadirlo a la Pizarra y dibujarlo
miPizarra.add(c1);
...
```

en el que se crea una variable (objeto) no primitiva de tipo **Circulo** que, después se añade, mediante la operación **add** a un objeto creado previamente, de tipo **Pizarra**, denominado **miPizarra**.

Como se recordará, las características que tienen las variables de tipo **Circulo**, esto es, cuáles son sus valores posibles y las operaciones que se pueden utilizar con ellas, vienen dadas por la definición de la clase **Circulo** (veáse la figura 2.1).

Obviamente, los valores que puede adoptar una variable de tipo **Circulo** son distintos de los que podría tomar en el caso de que perteneciera a un tipo primitivo

como pueden ser los tipos numéricos vistos en el capítulo previo, el tipo lógico o el carácter. De hecho, el valor de una variable de tipo **Círculo** debe entenderse como la agregación de los valores individuales de sus componentes o atributos; además estos valores individuales podrán a su vez pertenecer tanto a tipos primitivos como a otros tipos estructurados.

En el resto del capítulo, se retomará la definición de tipos no primitivos en Java que se comenzó, a título de ejemplo, en el capítulo 2; se mostrará cómo se representan en memoria los valores de dichos tipos y se estudiarán las implicaciones que ese modo de representación tiene en algunas de las operaciones ya vistas, tales como la asignación o la comparación, cuando se aplican sobre objetos. Finalmente se verá que es posible asociar información conjuntamente a todos los objetos de una clase, en lugar de hacerlo de forma individual a cada uno de ellos, utilizando para ello el modificador *static*.

## 4.1 Un nuevo ejemplo de definición de una clase

Un punto en un espacio de dos dimensiones puede definirse mediante sus dos coordenadas cartesianas que representarán respectivamente su posición con respecto al eje de las *X* (abscisa) y al eje de las *Y* (ordenada). Cada uno de estos dos valores será un número real que se podrá representar en Java mediante un valor de tipo **double**.

Para poder definir variables de tipo **Punto** en Java, será necesario definirlas mediante una clase en la que se indicará que un **Punto** es la agregación de dos componentes; uno que se denominará **x** (la abscisa) y otro **y** (la ordenada).

Según lo anterior, una primera definición de la clase **Punto** en Java es el siguiente:

```
class Punto {
 double x;
 double y;
}
```

Aunque la declaración y uso de métodos en una clase se trata posteriormente en el capítulo 5, a tenor de lo presentado en el capítulo 2, puede adelantarse que la definición más arriba de la clase **Punto**, sin operaciones constructoras, es equivalente a la misma clase **Punto** con una constructora sin instrucciones, esto es, es equivalente a la siguiente clase:

```
class Punto {
 double x;
 double y;

 public Punto() {}
}
```

Esto es así porque, en general, cuando se define una clase sin operaciones constructoras (como `Punto` en este ejemplo) el sistema le añade una por defecto y de ahí la equivalencia de ambas clases.

Almacenado cualquiera de los dos códigos anteriores en un fichero y tras haberlo compilado, puede usarse el mismo para definir y utilizar variables de tipo `Punto` en otra clase Java tal y como se hace, por ejemplo, en el programa siguiente:

```
class PruebaPunto {
 public static void main(String[] args) {
 // se definen e inicializan dos variables de tipo Punto
 Punto p1 = new Punto();
 Punto p2 = new Punto();

 // se asignan valores a los atributos de p1:
 p1.x = 1.0;
 p1.y = 1.0;

 // se asignan valores a los atributos de p2 usando p1:
 p2.x = 2.0 * p1.x;
 p2.y = -2.0 * p1.y;

 // se escriben los valores de los atributos de p2:
 System.out.println("(" + p2.x + "," + p2.y + ")");
 }
}
```

Aunque es sencillo deducir lo que hace el programa anterior (crea dos variables de tipo `Punto` y asigna valores a cada uno de los atributos de las mismas), hay algunos elementos que cabe destacar y que son generalizables a otras clases distintas a las del ejemplo que se deseen definir:

- Mediante la operación `new Punto()` se crea un objeto de dicho tipo y, en general, del tipo referenciado en la operación. A partir de la creación de un objeto se pueden utilizar los elementos del objeto, esto es, sus atributos y métodos.

Es posible crear tantos objetos como se deseé en un programa. Cada uno de ellos mantendrá la información correspondiente siguiendo la definición que se haya efectuado en la clase. Así, para la clase `Punto`, cada uno de los objetos que se construyan, como `p1` y `p2`, tendrá dos atributos, sus campos `x` e `y`, así como una operación constructora sin instrucciones.

- Los atributos de la clase `Punto` son variables de tipo `double` y pueden, por lo tanto, ser utilizadas como tales. En general, los atributos de una clase pueden ser de cualquier tipo (elementales o no) y pueden utilizarse siguiendo las reglas de uso de las variables correspondientes a dicho tipo.

- Los atributos de la clase `Punto` se han definido sin modificador de acceso<sup>1</sup> lo que se conoce en la terminología del Java como de acceso *friendly*. Los elementos con esa modalidad de acceso son accesibles desde todas las clases existentes en el mismo paquete en que se encuentre la clase en que se definen. Equivalen, a grandes rasgos, a que tengan acceso público (modificador `public`). Por eso es posible asignar y leer posteriormente el valor a los atributos de las variables definidas en la clase `PruebaPunto`.

Si los atributos se hubiesen definido privados (usando para ello el modificador `private`) no serían accesibles fuera de la clase en que se hubiesen definido (el compilador daría un error si se intentase acceder a los mismos). Como se verá más adelante, por motivos de seguridad, ésta última (privada) será la forma habitual de definir el acceso a los atributos de las clases.

- El uso de los atributos de la clase `Punto`, para asignarles valor o leerlo, se hace utilizando la notación de punto que sigue la sintaxis ya conocida:

`nombreDeVariableObjeto.nombreDeAtributo`<sup>2</sup>

## 4.2 Inicialización de los atributos

La asignación de valores iniciales a los atributos de un objeto puede efectuarse declarando explícitamente un valor inicial de los mismos, de forma que cuando el objeto se cree se asigne a sus atributos los valores deseados; así, por ejemplo, en la clase `Punto` se podría definir:

```
class Punto {
 double x = 1.0;
 double y = 1.0;

 public Punto() {}
}
```

de forma que todo objeto de tipo `Punto` tendrá inicializados sus dos atributos al valor 1.0 cuando se cree.

Además, tal y como se ha introducido en el capítulo 2 y se detallará en el capítulo 5, los constructores también se pueden utilizar para asignar un valor inicial a los atributos en el momento de la creación de los objetos.

Por último, conviene saber que en el caso en que no se den valores iniciales a los atributos, el sistema Java los inicializará por defecto de forma que,

---

<sup>1</sup>Según lo introducido en el capítulo 2, mediante los modificadores de acceso se define desde dónde es posible hacer uso de los elementos de la clase.

<sup>2</sup>En realidad, la notación de punto puede utilizarse no solamente con variables, sino con cualquier expresión que represente a un objeto; por ello, se puede decir más precisamente que su uso es: `expresiónDeTipoObjeto.nombreDeAtributo`.

en esencia, dará a los atributos numéricos el valor 0, a los de tipo carácter (`char`) el carácter de código 0 y a los lógicos (`boolean`) el valor `false`<sup>3</sup>.

### 4.3 Representación en memoria de los objetos. Variables referencia

En el capítulo anterior se ha visto que al definir una variable de un tipo elemental el sistema le asigna un espacio de memoria donde se almacena el valor que la variable tiene en cada momento durante la ejecución del programa. Las operaciones de asignación alteran, modificándolo, el contenido de las variables. Mediante su uso en expresiones, es posible conocer y operar con el valor que tenga en un momento dado cada variable.

Sin embargo, cuando se crea un objeto y se asigna a una variable de un tipo no elemental, el sistema le asocia memoria de una forma distinta al anterior. Lo que hace en este caso, es dividir la memoria en dos partes diferenciadas:

- Una parte asociada al objeto como tal, mediante la que se mantendrá la información propia del objeto (esto es el valor de sus atributos y los métodos que puede utilizar). Las operaciones de asignación a los atributos del objeto alterarán la memoria asociada a los mismos en dicha parte de la memoria.

La zona del sistema en la que se guarda la memoria de los objetos que se crean durante la ejecución de un programa, se denomina *montículo* o, en inglés, *heap*. Es una parte dinámica ya que, al crearse un objeto, se le asigna memoria en esta zona, pero cuando un objeto se destruye (por ejemplo, por que se haya dejado de utilizar) se le desasigna la memoria, pudiendo ser reutilizada posteriormente por el sistema.

- Otra parte de memoria se asocia a la variable con la que se nombra al objeto. Mediante esta parte, el sistema es capaz de determinar durante la ejecución del programa dónde se encuentra el objeto para poder actuar con él.

Cuando se crea un objeto, se dice que mediante la variable que da nombre al objeto se *referencia* a la zona de memoria donde se encuentra el objeto en sí. Por eso es habitual denominar a estas variables *variables referencia*. Se puede decir, aunque en cierta manera sea una simplificación, que el sistema almacena en la variable objeto el lugar del *montículo* donde se encuentra la información, o contenido, de dicho objeto.

Todos los objetos en Java se manipulan mediante variables referencia. Los únicos elementos en Java que no están referenciados son los valores de tipos elementales.

---

<sup>3</sup> En el caso de los atributos que sean una variable referencia, esto es, un objeto, el sistema los inicializará por defecto a `null`.

Las referencias a los objetos se asignan automáticamente por el gestor de memoria de la *JVM* y permanecen inaccesibles al usuario, lo que quiere decir que en un programa en Java no pueden existir operaciones explícitas de manejo de referencias.

Si dos variables referencia mantienen un mismo valor, entonces ambas están refiriéndose al mismo objeto. En Java sí que es posible comprobar este hecho. En particular, los únicos operadores permitidos para manipular los tipos referencia son las asignaciones vía el operador `=`, el operador de acceso `.` y las comparaciones, mediante `==` y `!=`.

Puede ocurrir que una variable referencia no identifique ningún objeto; en este caso se puede utilizar una constante especial del lenguaje, `null`, que se puede asignar a cualquier variable referencia y que indica que no existe un objeto referenciado por la variable que tiene ese valor especial. Es a este valor `null` al que se inicializan las variables de tipo objeto por defecto cuando se crean en el montículo y no se les asigna explícitamente un valor inicial.

Como se ha visto, la diferencia principal entre valores primitivos y valores referencia consiste en que las variables que representan a los primeros mantienen el valor de los mismos, mientras que las que representan a los segundos mantienen una referencia a los mismos. Todo ello implica un comportamiento distinto entre ambos tipos de variables, elementales y no elementales, que se refleja en su diferente manipulación. A continuación se muestran brevemente algunos de dichos aspectos.

#### 4.3.1 Declaración de variables. Operador `new`

Ya se conoce la forma de declarar variables para los tipos primitivos; sin embargo, en el caso de los objetos existe una diferencia importante: la declaración del objeto no genera ese mismo objeto. En el momento de la declaración de una variable de tipo referencia aún no existe el objeto referenciado.

Considérese, por ejemplo, la siguiente secuencia en Java que declara y crea un objeto de tipo `Circulo`:

```
// Se declara la variable circulo de tipo Circulo que de momento
// no referencia a ningún objeto
Circulo circulo;

// Se crea un objeto de tipo Circulo con los valores por defecto,
// asignándose a la variable circulo una referencia al objeto
circulo = new Circulo();
```

Esto es, cuando se desea utilizar un nuevo objeto de cierto tipo, es necesario crearlo explícitamente utilizando el operador `new`. Hasta el momento de su creación

el objeto no existe y cualquier intento de referenciarlo antes de dicho momento provocará un error en tiempo de ejecución. Las dos instrucciones anteriores se pueden agrupar en una única como sigue:

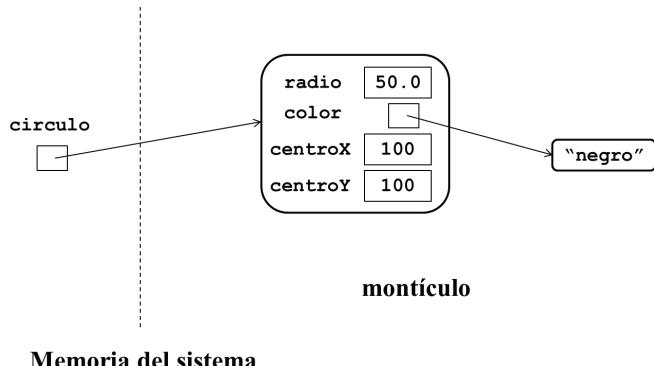
```
// Se declara la variable circulo de tipo Circulo, se crea un
// objeto de tipo Circulo con los valores por defecto y se
// asigna a la variable circulo una referencia al objeto
Circulo circulo = new Circulo();
```

En ambos casos, la ejecución del operador `new` tiene como resultado la construcción en el *montículo* del objeto correspondiente (en el ejemplo un `Circulo`) con espacio para sus atributos y métodos, así como la ejecución del constructor correspondiente en la clase que, recuérdese, puede no contener instrucciones.

Mediante la asignación a la variable (en el ejemplo `circulo`) se copia en dicha variable la referencia al objeto en el *montículo*. Desde ese momento, cualquier uso de la variable objeto permite que el sistema, gracias a la referencia que esta contiene, pueda operar con el mismo de forma transparente al programador.

En la figura 4.1 se ha representado gráficamente la situación de la memoria del sistema después de asignar a la variable `circulo` un objeto, creado con el constructor por defecto, de tipo `Circulo`. En las representaciones gráficas, se suele mostrar que una variable referencia señala a una zona de memoria determinada (que contiene el valor del objeto) mediante una flecha que apunta de la variable referencia a dicha zona.

Nótese que debido a que el atributo `color` es también un objeto (de la clase predefinida `String`) la memoria del mismo residirá, a su vez, en el montículo, encontrándose referenciada en este caso mediante el atributo `color`.



**Figura 4.1:** Situación de la memoria del sistema tras la creación de un objeto de tipo `Circulo` y su asignación a la variable `circulo`.

### 4.3.2 El operador de acceso “.”

Asociados a los objetos pueden existir operaciones o métodos que se aplicarán a los mismos. Como ya se ha indicado en el capítulo 2, el operador punto se emplea para seleccionar el método específico que se desee utilizar sobre el objeto en curso y también para acceder a los atributos de los objetos. Por ejemplo:

```
// definición y creación de un objeto de tipo Pizarra
// referenciado por la variable miPizarra
Pizarra miPizarra = new Pizarra("ESPACIO DIBUJO",300,300);
// definición y creación de un objeto de tipo Circulo
// referenciado por la variable circulo
Circulo circulo = new Circulo(50, "amarillo",100,100);
// uso del método add de la clase Pizarra
miPizarra.add(circulo);
```

Si cuando se ejecuta un método asociado a un objeto, no existe este último (por ejemplo, cuando tiene el valor `null`), se producirá un error que en Java se representa mediante lo que se denomina una excepción `NullPointerException`. El uso y tratamiento de las excepciones se discute en detalle en el capítulo 15.

### 4.3.3 La asignación

Dadas dos variables cualesquiera de tipos compatibles `v1` y `v2`, la operación

```
v1 = v2; // Asignación a v1 del valor de v2
```

reemplaza el contenido de `v1` con el de `v2`, tanto si ambas pertenecen a uno de los tipos primitivos como si se trata de variables referencia. La compatibilidad de tipos en el último caso y hasta la introducción de la herencia en el capítulo 14, sigue las dos reglas siguientes:

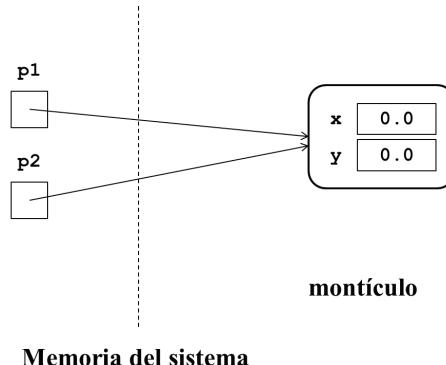
1. Dos variables son compatibles si referencian al mismo tipo de objeto.
2. Cualquier variable se puede convertir mediante *casting* explícito a tipo `Object`.

Nótese que cuando en la instrucción de asignación están involucradas variables referencia, la asignación significa tan solo un reemplazamiento de las referencias correspondientes, no del contenido referenciado por las mismas. Considérese el siguiente ejemplo en el que se utiliza, sin pérdida de generalidad la clase `Punto`:

```
Punto p1 = new Punto(); // p1 referencia a un objeto
 // de la clase Punto
Punto p2 = p1; // p1 y p2 referencian al
 // mismo objeto
```

Tras la ejecución de las instrucciones, se tiene un único objeto referenciado por las dos variables `p1` y `p2` y, por lo tanto, se tiene un mismo objeto al que se puede nombrar de dos formas distintas: `p1` y `p2`.

Esta situación está representada gráficamente en la figura 4.2. Una vez más, se muestra gráficamente mediante flechas, que las variables objeto referencian el espacio del montículo que mantiene la memoria del objeto.



**Figura 4.2:** Las dos variables de tipo Punto, `p1` y `p2`, referencian al mismo objeto.

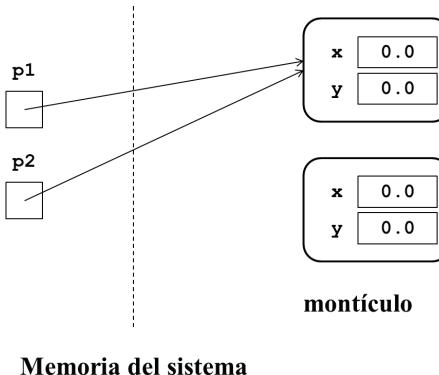
Considérese ahora el ejemplo siguiente, en el que se pierde la referencia a un objeto,

```
Punto p1 = new Punto(); // p1 referencia a un objeto
Punto p2 = new Punto(); // p2 referencia a otro objeto
p2 = p1; // p1 y p2 referencian al
 // primer objeto, nada referencia
 // al objeto segundo
```

Al igual que antes, `p1` y `p2` nombran al mismo objeto. Pero ahora nada referencia al objeto creado en segundo lugar. En una situación así, esto es, cuando ninguna variable referencia a un objeto, se dice que dicho objeto está *desreferenciado*. Gráficamente, la situación se muestra en la figura 4.3.

Considérese el ejemplo siguiente, donde se trata de dibujar dos círculos en cierta pizarra ya definida `miPizarra`:

```
// la Pizarra miPizarra ya ha sido creada
Circulo circulo1 = new Circulo();
Circulo circulo2 = circulo1;
circulo2.setColor("amarillo");
miPizarra.add(circulo1);
miPizarra.add(circulo2);
```



**Figura 4.3:** Las variables `p1` y `p2` referencian al mismo `Punto`, sin embargo el segundo `Punto` creado queda desreferenciado.

Obsérvese que tan sólo se ha creado un `Círculo`, por ello tras la asignación efectuada en la segunda línea se tienen dos variables que referencian un mismo objeto, el creado en la primera línea. Tanto `circulo1` como `circulo2` representan un único objeto. La operación `setColor()` se efectúa sobre el único objeto realmente existente, por lo que cuando ambos círculos se añadan en la `Pizarra` (objeto referenciado por la variable `miPizarra`) aparecerán dibujados con las mismas características.

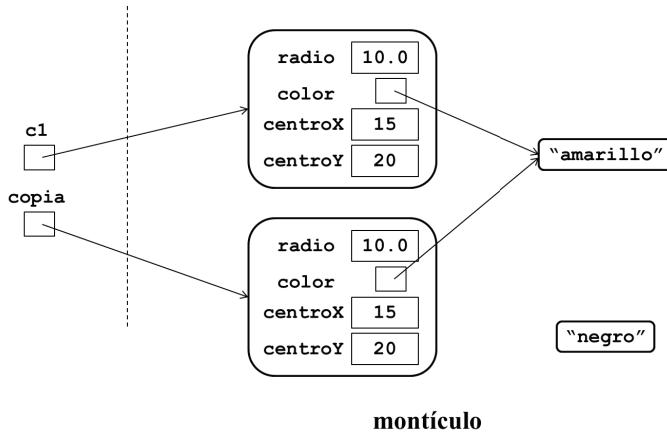
La conclusión que se puede extraer del ejemplo anterior es que es posible operar sobre un objeto utilizando cualquiera de los nombres de variables que lo representen o, lo que es lo mismo, que lo referencien. Y que, por supuesto, se tiene que ser cuidadoso cuando se dan situaciones de referenciación múltiple.

#### 4.3.4 Copia de objetos

Puede parecer que, ya que la asignación entre objetos sólo supone una copia de las referencias, entonces es imposible efectuar una copia de los objetos como tales. Sin embargo esto no es cierto, ya que si la estructura del objeto es conocida y accesible, entonces es posible realizar dicha copia mediante una copia individual, atributo a atributo, de cada uno de los elementos del tipo primitivo correspondiente. A efectos del ejemplo siguiente, supóngase ahora que los atributos de la clase `Círculo` fuesen accesibles, esto es, que se hubieran declarado de acceso `public` o `friendly` en lugar de `private`, entonces, mediante el código siguiente, se obtiene en la variable `copia` un objeto distinto con los mismos valores de sus atributos que tiene el objeto original `c1`.

```
// Crear un Circulo de radio 10, amarillo, con centro en (15,20)
Circulo c1 = new Circulo(10,"amarillo",15,20);
// se crea una copia:
Circulo copia = new Circulo();
copia.radio = c1.radio;
copia.color = c1.color;
copia.centroX = c1.centroX;
copia.centroY = c1.centroY;
```

Gráficamente, la situación que se muestra en la figura 4.4 es la que se da al finalizar la ejecución del segmento anterior. Nótese que los atributos del **Circulo copia**, aún habiéndose creado con el constructor por defecto, tienen los mismos valores que los de **c1**. La copia del atributo **color**, al tratarse de un objeto, ha supuesto solamente la copia de la referencia al mismo y, de ahí, que haya sólo una representación en memoria de la **String "amarillo"**. Por último, obsérvese que el objeto **String** original correspondiente al atributo **copia.color**, que tiene el valor **"negro"**, ha quedado desreferenciado.



### Memoria del sistema

**Figura 4.4:** Las variables **c1** y **copia** referencian dos **Circulo** distintos pero con los mismos valores.

Sin embargo, una solución como la anterior no se podrá utilizar si se sigue la regla de visibilidad que se propuso en el capítulo 2 según la cual, los atributos de las clases deben ser privados.

En esa situación la propia clase deberá incluir en su funcionalidad la copia de los objetos propios de la clase, ya que en la misma los atributos privados sí son accesibles (esta forma de actuar se introducirá en el capítulo 5).

#### 4.3.5 El operador == y el método equals

En los tipos primitivos el operador == es cierto o falso según sean o no iguales los valores de las variables que se comparan. Como cabe esperar, cuando este operador se aplica a objetos devolverá cierto o falso según sean o no iguales las referencias contenidas en cada una de las variables correspondientes. Por lo tanto, si se aplica el operador a dos objetos distintos, pero con el mismo valor, el resultado que se obtendrá será **false**.

Así, si se ejecuta la secuencia de código:

```
Punto p1 = new Punto();
p1.x = 3.0;
p1.y = -2.5;

Punto p2 = p1;

Punto p3 = new Punto();
p3.x = 3.0;
p3.y = -2.5;

System.out.println(p1==p1);
System.out.println(p1==p2);
System.out.println(p1==p3);
```

Se obtiene como resultado la secuencia:

| Salida Estándar |  |
|-----------------|--|
| true            |  |
| true            |  |
| false           |  |

Lo que es debido a que se están comparando las referencias contenidas en las variables. Siendo las contenidas en p1 y p2 iguales entre si y distintas, a su vez, de la contenida en p3.

Si lo que se desea es determinar la igualdad interna de los objetos, es necesario comparar la igualdad de su estructura mediante una comprobación de igualdad atributo a atributo.

Para efectuar dicha comprobación, como los atributos serán por lo general privados, se recurrirá a la definición de un método especializado, **equals** que existe, además, predefinido en la clase **Object**. La definición de este método se detalla en el capítulo 5.

### 4.3.6 El *garbage collector*

Cuando para un objeto dado, creado en algún momento de la ejecución de un programa no existe ninguna variable que lo referencie entonces dicho objeto está *desreferenciado*, lo que quiere decir que no es posible volver a operar con el mismo. Por ejemplo, en el siguiente ejemplo ya mostrado:

```
Punto p1 = new Punto(); // p1 referencia a un objeto
Punto p2 = new Punto(); // p2 referencia a otro objeto
p2 = p1; // p1 y p2 referencian al
 // primer objeto, nada referencia
 // al segundo objeto
```

tras la tercera instrucción toda referencia al objeto p2 se ha perdido. Dicho objeto ya no será accesible.

Para evitar la pérdida innecesaria de memoria, los lenguajes de programación introducen operaciones explícitas para informar al sistema que una zona de memoria determinada (por ejemplo, la ocupada por un objeto) no está referenciada, por lo que podría ser reutilizada. En Java, cuando un objeto está *desreferenciado* la memoria que consume se reclama automáticamente por un elemento que se denomina *recogedor de basura* (*garbage collector*). El proceso puede ser temporalmente costoso y su funcionamiento suele ser automático, aunque también puede ser ejecutado deliberadamente utilizando el método `System.gc()`.

## 4.4 Información de clase. Modificador static

En ocasiones es necesario mantener información común a todos los objetos de una clase en lugar o además de la información que pueda contener cada objeto. Como ejemplo, piénsese en que se deseara conocer para la clase `Punto` cuántos objetos de dicho tipo se han creado, o cuántos existen en un momento dado o, en el caso de la clase `Circulo`, ¿cuántos `Circulo` de color **negro** han cambiado su color a **amarillo** a lo largo de la ejecución de un programa?

La solución a los problemas anteriores pasa por la posibilidad de mantener información conjunta a toda la clase. En ese sentido se define *variable de clase* o *atributo de clase*, como una variable mediante la que es posible mantener información común a todos los elementos de la clase; ello en contraposición a las ya vistas *variables de objeto*, *de instancia* o *atributos*, mediante las que se mantiene información individual de cada objeto.

En Java, el sistema asigna memoria a las variables de clase la primera vez que se ejecuta código de dicha clase, lo que puede ocurrir la primera vez que se crea un objeto de la misma.

La memoria asociada a las variables de clase permanece en uso por el sistema hasta que la clase deje de estar *cargada* en memoria, normalmente al finalizar el programa que la utiliza.

Desde un punto de vista sintáctico, la definición en una clase de *variables de clase* o *atributos de clase* se hace precediendo sus identificadores por el modificador **static**. Por ejemplo, es posible alterar la definición de la clase **Punto** de la forma siguiente:

```
class Punto {
 double x;
 double y;

 static int contador = 0;

 public Punto() { contador++; }
}
```

Ahora se ha declarado un atributo, **contador**, de tipo **int**, que se ha inicializado a 0. Al llevar este atributo el modificador **static**, se está declarando que se trata de una variable de clase, por lo que existe un único almacenamiento para la misma, en lugar de existir para cada objeto como ocurre con el resto de atributos.

Como se ve, cada vez que se cree un nuevo **Punto**, se ejecutará el constructor que incrementará en uno el valor de dicha variable. A efectos de su visibilidad, un atributo de clase, tiene el mismo comportamiento que cualquier otro, por lo que es posible declararlo público (modificador **public**), privado (**private**) o *friendly*, tal y como es el caso.

Para poder referenciar variables de clase fuera de la clase donde se definen, se debe recordar que no están asociadas a ningún objeto, sino a una clase. En esa situación, para poder acceder a ellas, se utiliza en Java la notación:

**NombreDeClase.nombreDeAtributoDeClase**

esto es, se utiliza la notación de punto ya vista en el caso del acceso al contenido de los objetos, pero anteponiendo al nombre del atributo de clase el nombre de la clase donde este está declarado.

Así, almacenado y compilado el nuevo código de la clase **Punto**, se puede definir una clase en la que se determine o manipule el número de objetos de tipo **Punto** creados en un momento dado, como en la clase **PruebaPunto2** que se muestra a continuación.

```

class PruebaPunto2 {
 public static void main(String[] args) {
 // se escribe el valor inicial del contador de Puntos,
 // nótense que aún no se ha creado ningún objeto
 System.out.println("Num. puntos inicial: " + Punto.contador);

 // se definen e inicializan varias variables de tipo Punto
 Punto p1 = new Punto();
 Punto p2 = new Punto();
 Punto p3 = new Punto();

 // se calcula el número de Puntos creados
 int puntosCreados = Punto.contador;
 // puntosCreados vale 3

 // se resetea el número de Puntos:
 Punto.contador = 0;
 ...
 }
}

```

Un uso habitual de las variables de clase consiste en utilizarlas para almacenar un valor constante, común a todos los elementos de la clase, para el que no tendría sentido mantener una copia con el mismo valor en cada uno de los objetos de la clase.

En el siguiente ejemplo se utiliza esta técnica para definir una constante DOS\_PI en la clase `Circulo`. Como se puede ver, dicha constante, que se declara pública, es utilizada más adelante para modificar la definición del método `perimetro()`. En el ejemplo interviene otra constante (`PI`), de la clase predefinida `Math`, que es accedida y utilizada en el método `area()` según lo descrito.

El resto de la clase `Circulo`, señalado mediante puntos suspensivos, queda igual que en la figura 2.1.

```

public class Circulo {
 private double radio; private String color;
 private int centroX, centroY;
 public static final double DOS_PI = 2 * Math.PI;
 ...
 /** calcula el área del Circulo. */
 public double area() { return Math.PI * radio * radio; }
 /** calcula el perímetro del Circulo. */
 public double perimetro() { return DOS_PI * radio; }
 ...
}

```

## 4.5 Problemas propuestos

1. Considerando la definición de la clase Punto, ¿qué se escribe en la pantalla cuando se ejecuta el programa siguiente?

```
class XPunto {
 public static void main(String[] args) {
 Punto p1 = new Punto(), p2 = new Punto(), p3 = new Punto();

 p1.x = 1.0;
 p1.y = 1.0;

 p2 = p1;
 p2.x = 2.0 * p1.x;
 p2.y = -2.0 * p1.y;

 p3 = p1;
 System.out.println("(" + p2.x + "," + p2.y + ")");
 System.out.println("(" + p3.x + "," + p3.y + ")");
 }
}
```

2. Simplificar el código del programa anterior eliminando del mismo todas aquellas instrucciones que se consideren irrelevantes, pero manteniendo la misma salida.

3. ¿Qué se escribe por pantalla cuando se ejecuta el siguiente programa?

```
class XPunto2 {
 public static void main(String[] args) {
 Punto p1 = new Punto(), p2 = new Punto(), p3 = new Punto();

 p1.x = 1.0;
 p1.y = 1.0;

 p2 = p1;
 p2.x = 2.0 * p1.x;
 p2.y = -2.0 * p1.y;

 p3.x = p1.x;
 p3.y = p1.y;

 System.out.println(p1==p2);
 System.out.println(p1==p3);
 System.out.println(p2==p3);
 }
}
```

4. Consultar la ayuda (*API*) de Java para determinar las constantes existentes en la clase predefinida Math del paquete estándar `java.lang`.

5. Modificar la clase **Circulo** de la figura 2.1, para poder determinar:
  - El número de objetos que se han creado utilizando el constructor sin argumentos.
  - El número de objetos que se han creado utilizando el constructor con argumentos.

## Más información

[Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.  
URL: <http://math.hws.edu/javanotes/>. Capítulo 5 (5.1 y 5.2).

[Ora11d] Oracle. *The Java<sup>TM</sup> Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Classes and Objects - Objects.

[Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.  
Capítulos 4 y 5.

# Capítulo 5

## Métodos

El presente capítulo describe los conceptos relacionados con la funcionalidad de las clases y la subprogramación. En Java, ambos se concretan en los métodos, de los que ya se han visto ejemplos de uso en los capítulos 2 y 4.

Los métodos se utilizan para definir operaciones sobre ciertos datos con el fin de proporcionar cierto resultado. Estos métodos, a partir de ciertos datos, devuelven un resultado (valor de retorno) o realizan acciones sobre los datos sin devolver explícitamente ningún valor.

A través de los métodos se define la funcionalidad de una clase. Por ejemplo, en la definición de la *Clase Tipo de Dato Circulo* se tienen los métodos consultores `getRadio()` o `getArea()` o el método modificador `setRadio()`, que definen qué sobre un objeto de tipo *Circulo* se pueden realizar las operaciones de consultar su radio o su área o modificar el valor de su radio. También son ejemplos de métodos públicos los que definen la funcionalidad de una *Clase de Utilidades*, y el método `main` de una *Clase Programa*.

En cualquier clase se pueden definir métodos para agrupar e identificar una secuencia de acciones con el fin de poder ser utilizadas una o más veces a lo largo de la clase. Es lo que se conoce como *subprogramación* o *encapsulamiento del código*. Así, se facilita la reutilización y mantenimiento del código ya que se favorece:

- la *legibilidad*, puesto que el código queda mejor organizado en tareas y subtareas, cuyos detalles más o menos prolijos no se describen innecesariamente más de una vez.
- la *seguridad*, dado que si se llega a desarrollar un método sin errores, cualquiera de sus usos funcionará correctamente.

El objetivo de este capítulo es presentar de forma detallada la definición, el uso y la declaración de un método en Java.

## 5.1 Definición y uso de métodos

### 5.1.1 Definición de métodos: métodos de clase y de objeto

Un método es un segmento de código, debidamente encapsulado y parametrizado que se puede usar en otras partes del código, produciendo un valor resultante o teniendo algún efecto sobre los datos o en la ejecución del programa.

La *declaración de un método* consiste en describir el método, dándole un nombre, e indicando qué parámetros tiene, de qué tipo es el resultado que produce y qué código se ejecuta al usarlo. La declaración de los métodos se incluye en una clase para que se puedan usar en la propia clase o queden dispuestos para dar servicio a otras aplicaciones. Su sintaxis se describe con todo detalle en la sección 5.2.

El uso de un método con unos valores concretos de los parámetros se denomina *llamada* o *invocación* del método, y su sintaxis se describe en la sección 5.1.2.

El lenguaje otorga al programador la capacidad de decidir qué métodos puede ser útil declarar y, con ciertas restricciones, en qué clases puede ser útil incluir su declaración. Sin embargo, ambos aspectos tienen una gran influencia en la usabilidad del código desarrollado, por lo que el propio lenguaje propicia una determinada forma de organización de los métodos, la que en capítulos anteriores se ha denominado *orientada a objetos*.

**Ejemplo 5.1.** En el programa de la figura 5.1 se introducen como objetos Punto los tres vértices de un triángulo, para calcular la longitud de sus lados y finalmente su perímetro. Supóngase que en la clase del programa se hubiese declarado un método de nombre *distancia* que calculase la distancia entre los dos puntos que se le pasasen como parámetros. Entonces el siguiente código muestra cómo se podrían simplificar las líneas 20 a 30 del programa, reutilizándose tres veces el mismo método.

```
double lado12 = distancia(p1,p2);
double lado23 = distancia(p2,p3);
double lado13 = distancia(p1,p3);
```

Si otro programa necesitase calcular también distancias entre puntos, podría volver a declarar este mismo método. Sin embargo, no parece aceptable que aquellos métodos de utilidad general en la manipulación de puntos se reescriban ad hoc en cada programa que necesite usarlos. Lo más lógico parece ser concentrar en lo

possible todos estos métodos básicos en un solo lugar, que debe ser en donde resida toda la información acerca de cómo es y qué se puede hacer con un punto, es decir, la propia clase Punto.

```

1 /** Clase ProgramaTri: define un triángulo en el plano cartesiano,
2 * a partir de sus vértices y muestra su perímetro en la salida
3 * estándar.
4 * @author Libro IIP-PRG
5 * @version 2011
6 */
7 public class ProgramaTri {
8 public static void main(String[] args) {
9 Punto p1 = new Punto();
10 p1.x = 2.5; p1.y = 3;
11 Punto p2 = new Punto();
12 p2.x = 2.5; p2.y = -1.2;
13 Punto p3 = new Punto();
14 p3.x = -1.5; p1.y = 1.4;
15 System.out.println("Triángulo de vértices: ");
16 System.out.println("(" + p1.x + "," + p1.y + ")");
17 System.out.println("(" + p2.x + "," + p2.y + ")");
18 System.out.println("(" + p3.x + "," + p3.y + ")");
19
20 double dx = p1.x - p2.x;
21 double dy = p1.y - p2.y;
22 double lado12 = Math.sqrt(dx*dx + dy*dy);
23
24 dx = p2.x - p3.x;
25 dy = p2.y - p3.y;
26 double lado23 = Math.sqrt(dx*dx + dy*dy);
27
28 dx = p1.x - p3.x;
29 dy = p1.y - p3.y;
30 double lado13 = Math.sqrt(dx*dx + dy*dy);
31
32 double perimetro = lado12 + lado23 + lado13;
33 System.out.println("Perímetro: " + perimetro);
34 }
35 }
```

Figura 5.1: Clase ProgramaTri.

En concreto, una clase en la POO y en particular en Java, es tanto el contenedor de la descripción de la estructura de sus objetos como del repertorio de métodos que se consideren fundamentales en la manipulación de dicha clase de objetos. Con ello se facilita:

- El desarrollo de la clase, ya que al reunir información de la forma de los objetos y la funcionalidad deseada, la implementación de los métodos puede ser más eficiente. Incluso la estructura que se dé a los objetos de una clase puede supeditarse a facilitar la escritura de sus métodos.
- El uso de la clase, en concreto la localización y consulta de los métodos asociados, dado que el programador cuenta con que en dicha clase se reúna su funcionalidad básica.

Incidiendo en este aspecto, Java distingue entre métodos de *objeto* o *dinámicos* y métodos de *clase* o *estáticos*.

- Los *métodos de objeto* son aquellos que se definen en una clase para crear y manipular la información de un objeto de la clase. Se distingue entre:
  - *Métodos constructores*: Permiten crear los objetos de una clase (usando el operador `new`). Son consustanciales a las clases y existen por defecto, tal como se ha explicado en el capítulo 4. Se han visto ejemplos de su uso al crear objetos de la clase `Círculo` o de la clase `Punto`.
  - *Métodos de instancia*: Permiten manipular la información de un objeto de la clase previamente creado, aplicándoselos mediante la notación especial de “.”.
- Los *métodos de clase*, por el contrario, son los métodos que no se aplican sobre un objeto de la clase. Se deben declarar como `static`, de ahí que se les llame también *métodos estáticos* (y, en contraposición, a los métodos de objeto se les llame *métodos dinámicos*).

Las clases suelen venir acompañadas por una documentación que da información acerca de los métodos que proporciona cada una de ellas.

**Ejemplo 5.2.** En las figuras 5.2, 5.3, 5.4 y 5.5 se muestran respectivamente unos extractos de la documentación de las clases `String`, `Math` y `System` del estándar de Java. Los métodos estáticos se reconocen porque vienen precedidos por la palabra `static`, y el resto son métodos dinámicos.

| <i>Tipo de retorno</i> | <i>Nombre del método</i>                       | <i>Lista de parámetros</i>                                                                                                           |
|------------------------|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| int                    | <a href="#">indexOf(int ch)</a>                | Returns the index within this string of the first occurrence of the specified character.                                             |
| int                    | <a href="#">indexOf(int ch, int fromIndex)</a> | Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. |
| :                      | :                                              |                                                                                                                                      |
| static String          | <a href="#">valueOf(double d)</a>              | Returns the string representation of the double argument.                                                                            |

**Método estático**

Figura 5.2: Extracto de la documentación de String.

|                                         |                                                                                                                                                                                             |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">String()</a>                | Initializes a newly created String object so that it represents an empty character sequence.                                                                                                |
| :                                       |                                                                                                                                                                                             |
| <a href="#">String(String original)</a> | Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string. |

Figura 5.3: Extracto de la documentación de String: constructores.

|               |                                         |                                                                                              |
|---------------|-----------------------------------------|----------------------------------------------------------------------------------------------|
| static double | <a href="#">pow(double a, double b)</a> | Returns the value of the first argument raised to the power of the second argument.          |
| static double | <a href="#">random()</a>                | Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. |

Figura 5.4: Extracto de la documentación de Math. Todos los métodos de esta clase son estáticos.

|             |                                     |                                                        |
|-------------|-------------------------------------|--------------------------------------------------------|
| static long | <a href="#">currentTimeMillis()</a> | Returns the current time in milliseconds.              |
| static void | <a href="#">exit(int status)</a>    | Terminates the currently running Java Virtual Machine. |
| static void | <a href="#">gc()</a>                | Runs the garbage collector.                            |

Figura 5.5: Extracto de la documentación de System. Contiene algún método void.

### 5.1.2 Llamadas a métodos: perfil y sobrecarga.

En los ejemplos de documentación de las figuras 5.2 y 5.3 se observa que, además de dar una descripción del cálculo y efecto del uso de cada método, se da su *perfil* o *cabecera* que incluye:

- *Nombre*, identificador que distingue al método. Un constructor siempre tiene el mismo nombre que la clase.
- *Lista de parámetros*, indica cuántos parámetros tiene el método y de qué tipo es cada uno de ellos.
- *Tipo de retorno*, tipo del resultado del método. Los métodos constructores no tienen tipo de retorno. Todos los otros métodos sí que tienen tipo de retorno, incluso cuando el método no da ningún valor resultante, en cuyo caso el tipo es `void`.
- Si el método es estático, aparece al inicio del perfil la palabra reservada `static`.

Este perfil indica al usuario cómo se debe escribir una llamada al método.

Para un método de instancia debe escribirse de acuerdo a la siguiente sintaxis:

```
objeto.nombreMetodo(arg1, arg2, ..., argn)
```

en donde:

- `objeto` es cualquier expresión que se evalúe a un objeto de la clase.
- `nombreMetodo` es el nombre del método.
- `arg1, arg2, ..., argn` forman la lista de argumentos o datos de entrada del método, siendo  $n$  el número de parámetros del perfil ( $n \geq 0$ ).

Entre los argumentos de la llamada y los del perfil del método debe existir concordancia en cuanto a número, tipo y orden de aparición. Así, cada argumento deberá ser una expresión que se evalúe a un valor del mismo tipo (o compatible) que el que se indica en el perfil para el parámetro correspondiente. Los métodos constructores, que siempre se invocan con el operador `new`, también pueden recibir parámetros.

Para un método estático la llamada debe escribirse de acuerdo a la siguiente sintaxis:

```
NombreClase.nombreMetodo(arg1, arg2, ..., argn)
```

La diferencia con respecto a los métodos dinámicos es que no precisan aplicarse sobre ningún objeto. No obstante, continúa siendo válido todo lo que se ha dicho con respecto a los argumentos de la llamada.

El poner como prefijo de la llamada el nombre de la clase permite que Java conozca en qué clase se encuentra la declaración del método invocado, y no es necesario escribirlo cuando la llamada se hace dentro de la propia clase del método. Cabe notar que en los métodos de instancia esta información se deduce de la clase del objeto sobre el que se aplica el método.

La notación de “.” en el uso de los métodos dinámicos tiene además otras consecuencias relacionadas con la herencia y que no cabe discutir por el momento (véase capítulo 14).

**Ejemplo 5.3.** Considérese la siguiente declaración de variable:

```
String s = new String("Java");
```

La siguiente llamada calcularía la primera posición de **s** en la que aparece el carácter ‘v’:

```
s.indexOf('v')
```

y teniendo en cuenta que en los **Strings** los caracteres vienen numerados de 0 en adelante, la llamada se evaluaría a 2.

Cabe notar que si se hubiese iniciado **s = null**; la llamada anterior produciría un error de ejecución **NullPointerException**, dado que los métodos de instancia se deben aplicar sobre objetos efectivamente existentes.

En la misma clase **String** se encuentran métodos estáticos, como **valueOf**, que retorna un **String** con los caracteres que representan al número que se le pasa como parámetro. Este método no se aplica a ningún objeto preexistente y el único dato con el que trabaja es un valor **double**, como en la siguiente llamada:

```
String.valueOf(23.5+5.2)
```

que se evaluaría al **String "28.7"**.

En la clase **Math** todos los métodos son estáticos, pues sólo trabajan con datos numéricos. Así por ejemplo:

```
Math.pow(49.0,0.5)
```

se evalúa a 7.0, y

```
Math.random()
```

retorna un valor real aleatorio en  $[0.0,1.0[$ . Cabe notar que siempre se deben escribir los paréntesis () que encierran la lista de parámetros, aunque como en este último ejemplo dicha lista esté vacía.

En una misma clase puede haber más de un método con el mismo nombre, siempre que se distingan por su lista de parámetros: número, tipo u orden de los parámetros en la lista. En ese caso se dice que están *sobre*cargados. La sobrecregada de métodos es muy común, dado que es lógico que comparten el mismo nombre métodos que se pueden considerar unos como ligeras variantes de los otros.

**Ejemplo 5.4.** En la clase **String** el método **index0f** está sobrecregada, y se distingue cuál de ellos se está invocando por los argumentos de la llamada. Así, suponiendo que la variable **String s** fuese "sobrecregada", la llamada

```
s.indexOf('r')
```

se refiere al método que busca la primera ocurrencia de 'r' en s, y se evalúa a 3. En cambio

```
s.indexOf('r',4)
```

se refiere al método que busca la posición del carácter a partir del índice indicado, y que en este ejemplo se evalúa a 7.

Otro método sobrecregado de **String** es **substring**, que como se puede comprobar en la documentación de la clase admite uno o dos parámetros, con los resultados que ilustran las siguientes llamadas:

```
s.substring(0,5)
```

devuelve la subcadena de s comprendida entre el carácter 0 inclusive y el 5 exclusive, y por lo tanto vale "sobre". En cambio

```
s.substring(5)
```

devuelve la subcadena de s que se extiende desde el carácter 5 inclusive hasta el final, y por lo tanto vale "carga".

También el constructor de `String` está sobrecargado, pues además del usado al inicio del ejemplo 5.3, contiene otros constructores, como por ejemplo el constructor sin parámetros:

```
String s = new String();
```

que crea la secuencia "" vacía de caracteres.

Para discriminar entre dos métodos sobrecargados sólo es necesario distinguir el orden y el tipo de los parámetros, por lo que es habitual citarlos por su *signatura*:

nombre(lista de tipos)

como en `indexOf(char)` e `indexOf(char,int)`.

Hay que tener especial cuidado con la conversión automática de tipos en los parámetros de un método sobrecargado. Por ejemplo, en la clase `Math` se encuentran los métodos que calculan el mínimo de dos `int` y de dos `doubles` con signaturas `min(int,int)`, `min(double,double)`, respectivamente.

La llamada `Math.min(3,8)` está usando el primero de ellos, dado que Java siempre busca que el perfil de un método coincida exactamente con la invocación del mismo antes que utilizar la conversión automática de tipos. Si encuentra un método en el que los parámetros coincidan exactamente en número, tipo y posición con los argumentos de la llamada, usa dicho método. Sólo si no encuentra esa coincidencia, aplica la conversión implícita de tipos para hacer coincidir el perfil del método con los tipos de los parámetros reales de la invocación.

El perfil del método indica asimismo en qué contexto del código se puede usar una llamada:

- Si el tipo de retorno del método es `T`, la llamada puede aparecer en cualquier punto en el que se admite una expresión de tipo `T`.
- Los métodos `void` se convierten en una instrucción escribiendo un ; al final de la llamada.

**Ejemplo 5.5.** Son sintácticamente correctas las siguientes líneas de código:

```
String s = new String("Elemento");
int i = s.indexOf('e',s.indexOf('e')+1); // i vale 4
System.out.println("Segunda aparición de la letra e en "+s+": "+i);
double x = 25.47, y = 14.368;
s = String.valueOf(x*y); // s es "365.95296"
s = null; // ya no se puede aplicar ningún método a s
i = (String.valueOf(x*y)).indexOf('.'); // i vale 3
System.gc(); // instrucción de ejecución del método void gc()
System.out.println("Se ha ejecutado el garbage collector");
```

## 5.2 Declaración de métodos

Para que un método se pueda usar, el código que Java ejecuta al invocar el método debe estar declarado o descrito en alguna clase. De hecho, en el apartado anterior se han mostrado ejemplos de uso de métodos declarados en clases del estándar de Java.

Asimismo un usuario puede declarar en sus clases y programas los métodos que considere oportuno, con la restricción de que los métodos de objeto, constructores y de instancia, que podrán aplicarse a los objetos de una cierta clase se deben incluir siempre dentro de la propia clase.

La declaración de un método describe además de la cabecera, el *cuerpo* o bloque de instrucciones a ejecutar cada vez que se use el método. La sintaxis para un método no constructor es:

```
[modificadores] [static] TipoRetorno nombreMetodo([ListaParametros])
{
 // instrucciones del cuerpo del método
}
```

en donde los corchetes indican opcionalidad.

El modificador **static** de la cabecera indica que se está declarando un método estático o de clase.

La sintaxis de la declaración de los constructores es especial, dado que deben nombrarse obligatoriamente como la clase y no incluir ningún tipo de retorno en la cabecera:

```
[modificadores] NombreClase([ListaParametros])
{
 // instrucciones del cuerpo del método
}
```

Si no se declara ningún constructor, la clase tiene el constructor por defecto. En otro caso existirán en la clase única y exclusivamente aquellos constructores explícitamente declarados.

En las siguientes subsecciones se analizan cada uno de estos elementos con mayor detalle.

### 5.2.1 Modificadores de visibilidad o acceso

El control del acceso a los métodos de una clase, igual que en el caso de sus atributos, se logra mediante el uso de tres *modificadores*: **public**, **private** y **protected**.

Los métodos `public` se pueden usar en cualquier otra clase, y por lo tanto son los que definen la funcionalidad de la clase. Todos los métodos que aparecen documentados en las clases del *API* de Java [Ora11c] son públicos (por ejemplo los que se muestran en las figuras 5.2, 5.3, 5.4 y 5.5).

Los métodos `private` son aquellos que sólo se pueden usar en el código que se escriba dentro de la propia clase en la que se declara. Son métodos auxiliares que sirven únicamente como apoyo al desarrollo de la propia clase. Las clases del *API* de Java también contienen métodos privados, pero no se muestran en la documentación.

La visibilidad `protected` está relacionada con la herencia y se verá en el capítulo 14. Si no aparece ninguno de los tres modificadores comentados, la visibilidad es *friendly* e indica que el método es público únicamente dentro del paquete en el que está incluido (inaccesible para el resto).

El constructor por defecto siempre es `public`. Por razones obvias y salvo muy contadas excepciones (ver sección 5.6), también se declaran `public` el resto de constructores de una clase.

### 5.2.2 Tipo de retorno. Instrucción `return`

El tipo de retorno puede ser tanto un tipo primitivo como una referencia a objeto. Por ejemplo, en el método `indexOf(char)` de `String` retorna un `int`, mientras que el método `valueOf(double)` de la misma clase retorna un objeto `String`. Cabe insistir en que la palabra reservada `void` indica que el método no devuelve ningún valor, como es el caso del método `gc` de la clase `System`.

Existe una instrucción especial que se usa en el cuerpo de los métodos no `void` para darle valor a cada llamada, y cuya sintaxis es:

```
return expresion;
```

en donde `expresion` es cualquier expresión del tipo de retorno del método (o compatible con el mismo). Cuando se ejecuta esta instrucción, se evalúa la expresión y el valor resultante es el que se devuelve como resultado de la ejecución del método, que se da por terminada (aunque hubiera escritas a continuación otras instrucciones).

### 5.2.3 Lista de parámetros

La sintaxis de la lista de parámetros de un método (que se sitúa entre paréntesis tras el identificador del método) es la siguiente:

```
tipo1 nomParam1, tipo2 nomParam2, ..., tipon nomParamn
```

Es habitual confundir la sintaxis de la lista de parámetros con la de la declaración de variables; en la lista de parámetros hay que especificar separadamente el tipo de cada parámetro, no siendo posible listas de este estilo: (int a, b, c).

A los parámetros que aparecen en la cabecera se les denomina *parámetros formales* del método, mientras que los valores que se pasan como argumentos en la llamada al método se denominan los *parámetros reales* de la llamada.

#### 5.2.4 Cuerpo del método. Acceso a variables. Referencia this

El cuerpo de un método puede contener cualquier secuencia de instrucciones válida, incluyendo declaraciones de variables, llamadas a otros métodos o incluso a él mismo (capítulo 11). Por ejemplo:

```
public static void bisiesto(int año) {
 boolean caso1 = año%4==0 && año%4!=0;
 boolean caso2 = año%400==0;
 System.out.println("El año " + año + " es bisiesto: "
 + (caso1 || caso2));
}
```

En la declaración del método los parámetros no tienen un valor concreto asignado, pues éste se les da cuando se realiza la llamada al método. Así, al hacer una llamada como `bisiesto(2012)` es cuando en concreto el parámetro `año` toma el valor 2012, y el valor que toma la expresión `caso1 || caso2` es `true`.

Si el método tiene un tipo de retorno distinto de `void` debe contener al menos una instrucción `return`. Por ejemplo:

```
public static char minusAMayus(char c) {
 return (char)(c + 'A' - 'a');
}
```

Este método, siempre que `c` reciba como valor una letra minúscula, retornará la correspondiente mayúscula. Por ejemplo, la llamada `minusAMayus('f')` toma el valor '`F`'.

El siguiente método:

```
public static String fecha(int dia, int mes, int año) {
 String d = "0" + dia; d = d.substring(d.length()-2);
 String m = "0" + mes; m = m.substring(m.length()-2);
 String a = "000" + año; a = a.substring(a.length()-4);
 return d + "/" + m + "/" + a;
}
```

retornaría "28/01/0814" si se invoca como `fecha(28,1,814)`.

El ámbito de las variables en el cuerpo de un método es como sigue:

- Las variables declaradas en el cuerpo son locales al método.
- Los parámetros formales se consideran variables locales que se inician con el valor de los parámetros reales de la llamada.
- Cualquier método puede acceder a los atributos de los objetos que manipule, sea cual sea su clase, siempre que lo permitan los criterios de visibilidad `private/public` de dichos atributos.

Por ejemplo, los objetos `String` contienen un atributo `private int count` cuyo valor es la longitud de la cadena de caracteres. Entonces, si el método anterior se hubiese escrito:

```
public static String fecha(int dia, int mes, int año) {
 String d = "0" + dia; d = d.substring(d.count-2);
 String m = "0" + mes; m = m.substring(m.count-2);
 String a = "000" + año; a = a.substring(a.count-4);
 return d + "/" + m + "/" + a;
}
```

se produciría un error de compilación debido a que `count` tiene un acceso privado en `String`.

En cambio, si los atributos `x` e `y` de la clase `Punto` están declarados públicos, en la clase de la figura 5.1 se podría incluir la declaración del siguiente método y se podría usar para calcular la longitud de los tres lados del triángulo:

```
public static double distancia(Punto p1, Punto p2) {
 double x = p1.x - p2.x;
 double y = p1.y - p2.y;
 return Math.sqrt(x*x + y*y);
}
```

El cálculo de la distancia entre puntos se podría declarar dentro de la propia clase `Punto`, en cuyo caso sería posible escribirlo como un método de instancia con el perfil:

```
public double distancia(Punto p)
```

en donde uno de los puntos es `p` y el otro es el punto sobre el que se aplica el método.

En general, los métodos de instancia trabajan con un parámetro adicional que no aparece explícito en la cabecera: el *objeto en curso*. En el caso de un método de instancia, el objeto en curso es el objeto concreto de la clase al que se le aplica el método en una llamada. En el caso de un constructor el objeto en curso es el propio objeto creado por el método.

Para poder manipular el objeto en curso en el cuerpo del método como se hace con el resto de parámetros, existe una variable local **final** denominada **this**, que no hay que declarar y que se inicia al objeto en curso.

**Ejemplo 5.6.** Supóngase los siguientes métodos declarados en la clase Punto:

```
/** Crea un Punto de coordenadas 0.0, 0.0. */
public Punto() {
 // los atributos de this se inician
 // a los valores por defecto
}

/** Crea un Punto de abscisa px y ordenada py. */
public Punto(double px, double py) {
 this.x = px;
 this.y = py;
}

/** Retorna la abscisa del Punto. */
public double getX() {
 return this.x;
}

/** Retorna la distancia entre el Punto y p. */
public double distancia(Punto p) {
 double x = p.x - this.x;
 double y = p.y - this.y;
 return Math.sqrt(x*x + y*y);
}
```

entonces se podría escribir el siguiente código:

```
Punto p = new Punto(4.0,3.0); // p tiene abscisa 4.0, ordenada 3.0
double x = -2.6*p.getX(); // x vale -10.4
double dist = p.distancia(new Punto()); // dist vale 5.0
```

En todos estos métodos **this** es un objeto de clase Punto. En los dos últimos métodos **this** se inicia con el punto sobre el que se aplique el método en cada llamada, de igual modo que el parámetro **p** de **distancia** se inicia en cada llamada con el punto que se pase como parámetro real.

En Java es correcto declarar en un método un parámetro o una variable local con el mismo nombre que un atributo de la clase. Para resolver la posible ambigüedad se sigue el *principio de máxima proximidad*: siempre se asocia un nombre a una variable local antes que a un atributo. Se dice también que la variable local **oscurece** u oculta al atributo.

Por agilidad en la escritura, Java permite obviar el uso `this` cuando se cita a algún atributo del objeto en curso, siempre que no haya ambigüedad. Por ejemplo, la siguiente declaración es correcta:

```
public void mover(double px, double py) {
 x += px;
 y += py;
}
```

en donde Java entiende que `x`, `y` sólo pueden ser los correspondientes atributos del punto a mover, dado que no hay ninguna otra variable del método con alguno de estos nombres.

Sin embargo, si hubiera confusión como en el siguiente ejemplo, Java aplicaría el principio de máxima proximidad:

```
public void mover(double x, double y) {
 x += x; // Error lógico
 y += y; // Error lógico
}
```

Por dicho principio, las instrucciones del método se están aplicando a los parámetros `x` e `y` en lugar de a los atributos `x` e `y`, es decir, ocurre un error lógico. Para evitar esta ambigüedad y conseguir el resultado deseado, el uso de `this` no se puede obviar en este caso.

**Ejemplo 5.7.** En el método `distancia` del ejemplo 5.6 la palabra `this` no se puede obviar, pues además de producir un error lógico por confusión entre el nombre de los atributos y el nombre dado a las variables locales, se produciría un error de compilación. En efecto, si el método se escribiese

```
public double distancia(Punto p) {
 double x = p.x - x; // Error de compilación
 double y = p.y - y; // Error de compilación
 return Math.sqrt(x*x + y*y);
}
```

en la parte derecha de la asignación `double x = p.x - x;` se intenta acceder a la variable local `x` que se está declarando en la parte izquierda y no está inicializada.

Aunque en el cuerpo de un método se puede modificar el valor de los atributos del objeto `this`, no es posible cambiar de objeto en curso porque la variable `this` es `final`.

La palabra reservada `this` también se puede usar dentro de un constructor para llamar a otro constructor de la misma clase en la forma `this(...)`, en cuyo caso debe ser la primera instrucción del cuerpo. Por ejemplo, el siguiente constructor de la clase `Punto` crea un punto cuyas coordenadas copia de `p`, para lo que llama al constructor de la clase que recibe como parámetros las coordenadas del punto:

```
public Punto(Punto p) {
 this(p.x,p.y);
}
```

### 5.3 Clases Programa: el método main

Un caso especial de método estático es el método `main` que se ha ido utilizando en la implementación de pequeñas aplicaciones en los capítulos previos. Como es sabido, los métodos no se ejecutan nunca por sí mismos, sino que necesitan ser llamados desde otro punto del código de la aplicación. De esta forma, sería imposible poder ejecutar una aplicación, pues ningún método se ejecutaría por sí mismo (requiere la llamada desde otro método). El método `main` es la excepción, ya que es el método que, por omisión, invoca la *JVM* cuando ejecuta la orden `java NombreClase`.

Las clases que contienen un `main` son las que pueden iniciar la ejecución del código. A estas clases, como se ha visto en el capítulo 2, se las denomina *Clases Programa*. En la figura 5.6 se muestra la clase `PruebaMetodos`, una Clase Programa que declara un par de métodos que permiten comparar y escribir en la salida unas horas que se leen de teclado. El método `main` realiza la entrada de datos<sup>1</sup> y la escritura de resultados.

El método `main` presenta la siguiente cabecera predefinida:

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

A la vista de su cabecera, se puede decir de `main` lo siguiente:

- Es un método visible desde fuera de su clase (`public`) y es un método de clase (`static`).
- No devuelve ningún resultado (`void`).

---

<sup>1</sup> Mediante la clase `Scanner`, discutida con detalle en el capítulo 7

```
/**
 * Clase PruebaMetodos.
 * @author Libro IIP-PRG
 * @version 2011
 */
import java.util.Scanner;
public class PruebaMetodos {
 public static void main(String[] args) {
 Scanner tec = new Scanner(System.in);

 System.out.println("Introduce una hora: hora minutos");
 int h1 = tec.nextInt(), m1 = tec.nextInt();

 System.out.println("Introduce otra hora: hora minutos");
 int h2 = tec.nextInt(), m2 = tec.nextInt();

 String s1 = displayHora(h1,m1);
 String s2 = displayHora(h2,m2);

 System.out.print(s1 + " es anterior a " + s2 + "? ");
 System.out.println(anterior(h1,m1,h2,m2));
 }

 /** Devuelve true si la hora dada por h1 m1 (h y min,
 * respectivamente) es anterior a la dada por h2 m2. */
 public static boolean anterior(int h1, int m1, int h2, int m2) {
 return h1<h2 || (h1==h2 && m1<m2);
 }

 /** Devuelve la hora en el formato "hh:mm". */
 public static String displayHora(int hora, int minutos) {
 String h = "0" + hora; h = h.substring(h.length()-2);
 String m = "0" + minutos; m = m.substring(m.length()-2);
 return h + ":" + m;
 }
}
```

Figura 5.6: Clase PruebaMetodos.

- Recibe como parámetro un *array* (capítulo 10) de **String**, que contiene cada uno de los argumentos escritos en el intérprete de órdenes del terminal o instanciados a través del entorno de ejecución correspondiente (*BlueJ* u otro).

## 5.4 Ejecución de una llamada

### 5.4.1 Registro de activación. Pila de llamadas

Java sólo ejecuta un método en cada momento, del que se dice que es el *método activo*. Si un primer método invoca a otro, el primero queda en suspenso, y su ejecución sólo se reanuda tras resolver la ejecución de la llamada al segundo método.

Cuando Java procesa una instrucción en la que aparece una llamada a un método, se pasa a ejecutar las instrucciones del cuerpo. A cada llamada en ejecución Java le asocia una zona de memoria exclusiva para sus propios datos y cálculos, ajena al montículo, que se conoce como su *registro de activación*.

El registro de activación contiene:

- Una variable por cada variable local y por cada parámetro que aparece en la declaración del método, incluyendo una variable final **this** si el método es de objeto. Estas variables serán del tipo correspondiente según dicha declaración.
- Si el método tiene tipo de retorno, una variable del mismo tipo que el método y en la que se almacena el valor a devolver como resultado de la llamada.
- Una variable que almacena el punto al que tiene que regresar el control de la ejecución cuando termine de ejecutarse la llamada, o lo que es lo mismo, el punto en que se suspendió la llamada anterior.

Las dos últimas variables son manipuladas automáticamente por la *JVM* para organizar los cálculos y la gestión de las llamadas, y para facilitar la discusión que sigue, se les dará un nombre de conveniencia, el *valor de retorno (VR)* y la *dirección de retorno (DR)* respectivamente.

Cuando se está ejecutando un método *A* y se llega a una instrucción en *A* que es una llamada al método *B* se dan los siguientes pasos:

1. Se evalúan en *A* las expresiones que aparecen como parámetros reales en la llamada.

2. Se reserva espacio en memoria para un registro de activación del método invocado *B*.
3. Los parámetros del registro se inician a los valores de los argumentos que se han calculado en *A*.  
Si el método es de instancia, la variable `this` del nuevo registro se inicia con el objeto en curso. Si es un constructor, se inicia con el propio objeto creado.
4. Se realiza la ejecución del código del cuerpo de *B*. Si el método tiene un tipo de retorno, acaba en cuanto se ejecuta la instrucción `return expr;` y el valor de retorno es entonces el resultado de evaluar `expr`.
5. Acabada la llamada a *B*, su registro de activación se libera, no sin antes recuperar de *DR* el punto por dónde se retomar la ejecución de *A*.
6. Se reanuda la ejecución de *A*. Si *B* es un método con retorno, la llamada recién terminada se evalúa en *A* al valor de retorno; si *B* es un constructor (invocado con el preceptivo `new`) la llamada se evalúa al propio objeto creado.

El código del método activo sólo pueden manipular las variables de su registro de activación, conocido como *registro activo*. Este mecanismo preserva el estado del método suspendido, fundamentalmente el estado de sus variables locales, hasta que el control regrese al punto del método en el que se detuvo su ejecución.

**Ejemplo 5.8.** Supóngase definida la clase `Punto` como en el ejemplo 5.6, y un programa `Prueba` cuyo método `main` ejecuta el siguiente código:

```
double x = 0.0;
Punto q1 = new Punto(3.0,4.0);
Punto q2 = new Punto();
x = q1.distancia(q2);
System.out.println(x);
```

La figura 5.7 muestra cómo evoluciona la pila a medida que avanza la ejecución.

Dado que el registro de un método en suspenso no puede destruirse hasta que dicho método no termine, en memoria pueden coexistir varios registros de activación: uno para el método activo, y uno por cada método que esté suspendido. Para facilitar su gestión, los registros aparecen ordenados en memoria por el momento en que se crean. Se suelen representar uno encima de otro, o apilados. Efectivamente, la zona de memoria en que se disponen los registros se denomina *pila de llamadas*<sup>2</sup> y se gestiona como tal: cuando se hace una nueva llamada, se apila un nuevo registro, y cuando termina se desapila su registro. El registro activo es entonces el que se encuentra en la cima de la pila, es decir, el más reciente.

---

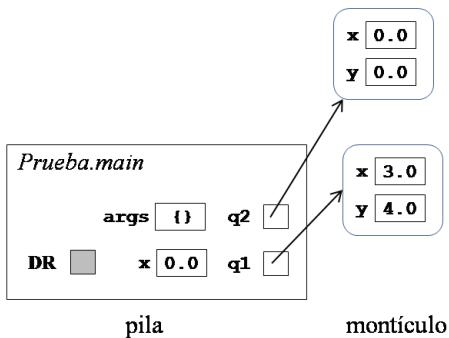
<sup>2</sup> *stack* en inglés.

```

public double distancia(Punto p) {
 double x = p.x - this.x;
 double y = p.y - this.y;
 return Math.sqrt(x*x + y*y);
}

public static void main(String[] args) {
 double x = 0.0;
 Punto q1 = new Punto(3.0,4.0);
 Punto q2 = new Punto();
 → x = q2.distancia(q1);
 System.out.println(x);
}

```



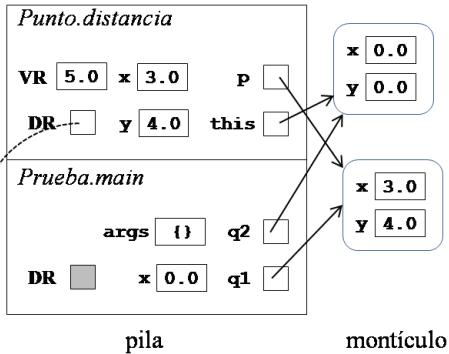
(a) Estado de la pila justo antes de ejecutar la llamada a `distancia`

```

public double distancia(Punto p) {
 double x = p.x - this.x;
 double y = p.y - this.y;
 → return Math.sqrt(x*x + y*y);
}

public static void main(String[] args) {
 double x = 0.0;
 Punto q1 = new Punto(3.0,4.0);
 Punto q2 = new Punto();
 x = q2.distancia(q1); ←
 System.out.println(x);
}

```



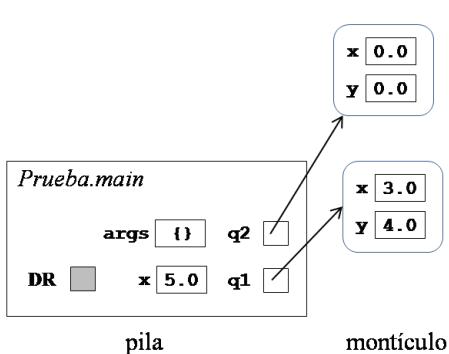
(b) Estado de la pila justo antes de devolver el resultado de `distancia`

```

public double distancia(Punto p) {
 double x = p.x - this.x;
 double y = p.y - this.y;
 return Math.sqrt(x*x + y*y);
}

public static void main(String[] args) {
 double x = 0.0;
 Punto q1 = new Punto(3.0,4.0);
 Punto q2 = new Punto();
 x = q2.distancia(q1);
 → System.out.println(x);
}

```



(c) Estado de la pila al regresar de la llamada a `distancia`

Figura 5.7: Evolución de la ejecución de Prueba y la pila de llamadas.

Como caso aparte, las variables de clase aparecen solo una vez en memoria. Cada clase que interviene en un programa dispone de una zona de memoria reservada para sus atributos estáticos. Dicha zona es accesible a cualquier método, que puede leer o modificar una variable siempre que los criterios de visibilidad se lo permitan.

### 5.4.2 Paso de parámetros por valor

Como se ha indicado en el apartado anterior, los parámetros formales se tratan como variables locales del registro de activación de la llamada, que se inician con el valor de los parámetros reales. Debido a ello, se dice que Java realiza un *paso de parámetros por valor*. En consecuencia, cualquier cambio realizado por un método sobre sus parámetros es local a la llamada.

No obstante, cabe resaltar que los objetos residen en el montículo y no se copian en el registro de activación, por lo que aunque el registro de activación acaba despareciendo, los cambios realizados en el montículo por el método perduran tras la llamada.

**Ejemplo 5.9.** Supóngase el siguiente método que pretende intercambiar el valor de dos variables reales:

```
public static void intercambio(double a, double b) {
 double aux = a;
 a = b;
 b = aux;
}
```

y una llamada al método desde la clase en la que está implementado:

```
double x = 5.0, y = 7.0;
intercambio(x,y); // después de la llamada: x = 5 e y = 7
```

Con esta invocación, los argumentos **a** y **b** se inicializan a los valores 5.0 y 7.0, respectivamente. Tras la ejecución de las instrucciones del cuerpo del método, **a** y **b** intercambian sus valores, pero las variables **x** e **y** no sufren ningún cambio.

Por un razonamiento análogo, la ejecución del siguiente método tampoco tiene ningún efecto sobre sus argumentos:

```
public static void intercambio(Punto a, Punto b) {
 Punto aux = a;
 a = b;
 b = aux;
}
```

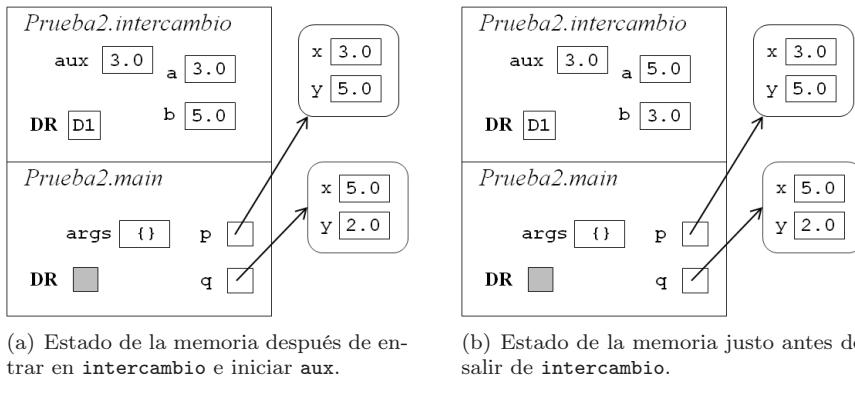
En cambio, el siguiente método de Punto sí que produce efectos perdurables sobre las abscisas de los puntos:

```
public void intercambioX(Punto b) {
 double aux = this.x;
 this.x = b.x;
 b.x = aux;
}
```

Supóngase una clase Prueba2 que contuviese la declaración de los anteriores métodos `intercambio(double,double)`, `intercambioX(Punto)` y un método `main` en el que se ejecutase el siguiente código:

```
Punto p = new Punto(3.0,5.0), q = new Punto(5.0,2.0);
intercambio(p.x,q.x); // punto de retorno D1
p.intercambioX(q); // punto de retorno D2
```

En las figuras 5.8 y 5.9 se aprecia la diferencia de comportamiento de ambos métodos.

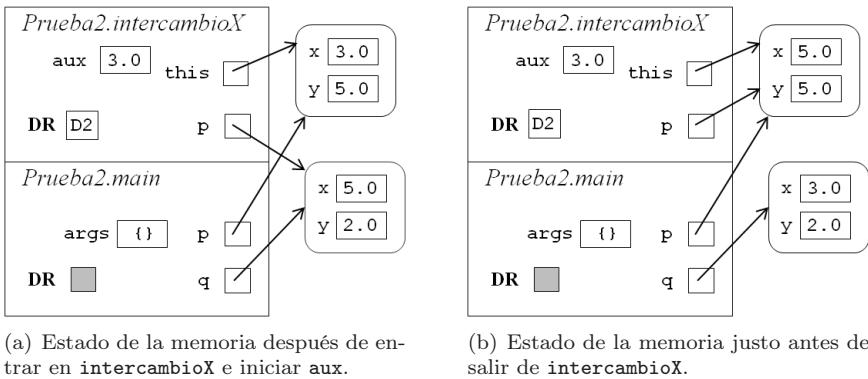


**Figura 5.8:** Ejecución de la llamada `intercambio(p.x,p.y)`.

## 5.5 Clases Tipo de Dato

### 5.5.1 Funcionalidad básica de una clase

Los métodos públicos definen la funcionalidad de una clase. Así pues, definir una *Clase Tipo de Dato* consiste tanto en declarar los atributos que estructuran los objetos de la clase, como en declarar los métodos que permiten manipularlos.



**Figura 5.9:** Ejecución de la llamada `p.intercambioX(q)`.

Por ejemplo, al inicio del capítulo se discutió la utilidad de que un método de cálculo de la distancia entre dos puntos se declare en la clase `Punto`, en lugar de declararse repetidamente en las diversas clases que manejen puntos. De esta manera quedaría disponible para usarse en cualquier otra aplicación, como por ejemplo en un programa que calculase el perímetro de un triángulo a partir de sus vértices, o en un programa que calculase el radio de un círculo a partir del punto central y un punto en la circunferencia.

En general, se puede resumir el diseño de una *Clase Tipo de Dato* en los siguientes apartados:

1. Declaración de los atributos que componen los objetos de la clase. A veces es útil incluir otros atributos auxiliares, estáticos o dinámicos, que no son estrictamente necesarios pero que pueden ayudar a desarrollar los métodos de la clase. En general, los atributos se declaran `private`.
2. Declaración de los métodos de la clase. Aquellos que se declaran `public` son los que aparecen en la documentación y podrán usarse en cualquier otra clase.

Además de los constructores, un repertorio suficiente de métodos permitirán consultar y modificar el estado de los atributos de los objetos, llamándoseles por ello *consultores* y *modificadores*. Es una norma de estilo ampliamente seguida que en las clases existan un consultor y un modificador por cada atributo. Estos métodos se suelen nombrar con identificadores que empiezan por `get` y `set` respectivamente, lo que enfatiza su función y los hace fácilmente reconocibles.

También puede resultar útil incluir métodos `static` que serán en general aquellos que no precisen aplicarse sobre un objeto previamente creado de la

clase, además de métodos auxiliares `private` cuyo cometido será ayudar al desarrollo de otros métodos.

**Ejemplo 5.10.** En este ejemplo se va a desarrollar una clase `Punto` cuyos objetos serán puntos en el plano, con coordenadas `x` e `y`, y que además tendrá la capacidad de contar los puntos que se creen a lo largo de una ejecución. Sus atributos son:

```
private double x; // abscisa del Punto
private double y; // ordenada del Punto
private static int contadorPuntos = 0;
private final static int DIMENSION = 2;
public final static Punto ORIGEN = new Punto();
```

La dimensión es un atributo común a todos los objetos y no necesita replicarse en cada instancia, por lo que se ha declarado estático. Se ha definido además una constante que da nombre al origen de coordenadas.

Los métodos constructores deberán actualizar la variable global o atributo de clase `contadorPuntos`:

```
public Punto() { contadorPuntos++; }

public Punto(double x, double y) {
 this.x = x; this.y = y;
 contadorPuntos++;
}

public Punto(Punto p) {
 this(p.x,p.y);
 contadorPuntos++;
}
```

El código para los métodos “gets” y “sets” de la clase `Punto` es:

```
public double getX() { return this.x; }
public double getY() { return this.y; }
public void setX(double x) { this.x = x; }
public void setY(double y) { this.y = y; }
public int getDim() { return DIMENSION; }
```

De esta forma se puede consultar estos atributos del objeto en otra clase, como por ejemplo la de la figura 5.11. Si por ejemplo en esta clase, para el punto `p` se

hubiese citado a `p.x` se habría dado un error de compilación por intentar acceder a información privada de `Punto`.

El último método no precisa actuar sobre ningún objeto de la clase, por lo que se incluye también su contrapartida estática:

```
public static int dimension() { return DIMENSION; }
```

de este modo se puede consultar la dimensión como `Punto.dimension()`, antes incluso de crear ningún punto. Se incluyen otros métodos estáticos, como el que consulta el número de puntos creados, y el que permite obtener un punto a partir de las coordenadas polares.

La clase se completa con otros métodos como el cálculo de la distancia entre puntos, y el que mueve un punto cambiando sus coordenadas. En esta clase se va a incluir además un método auxiliar `private`:

```
private aleatorio() {
 return (int)(Math.random()*(this.distancia(ORIGEN)+1));
}
```

que calcula un valor aleatorio en el rango `[0.0,d]`, siendo `d` la distancia del punto al origen. Este método se usa en la declaración del siguiente método:

```
public void moverAleat() {
 this.x += this.aleatorio();
 this.y += this.aleatorio();
}
```

Cabe recordar que Java permite obviar la palabra `this`, de modo que las asignaciones anteriores se podrían escribir como:

```
x += aleatorio();
y += aleatorio();
```

sobreentendiéndose que el método `aleatorio()` se aplica al objeto en curso.

En la figura 5.10 se muestra el código completo de la clase `Punto` y en la figura 5.11 el ejemplo ya mencionado de uso de los métodos de dicha clase.

### 5.5.2 Sobrescritura de los métodos implementados en `Object`

La fuerte orientación a objetos de Java y su uso intensivo de características de la misma (como la herencia, véase capítulo 14) hace que todos los objetos creados de

```


/**
 * Clase Punto: define un punto en el plano cartesiano,
 * determinado por sus coordenadas x e y.
 * Contabiliza el número de puntos creados.
 * @author Libro IIP-PRG
 * @version 2011
 */
public class Punto {
 private double x;
 private double y;
 public final static Punto ORIGEN = new Punto();
 public final static int DIMENSION = 2;
 private static int contadorPuntos = 0;

 /** Crea un Punto con coordenadas (0.0,0.0). */
 public Punto() { contadorPuntos++; }
 /** Crea un Punto con coordenadas (x,y). */
 public Punto(double x, double y) {
 this.x = x; this.y = y;
 contadorPuntos++;
 }
 /** Crea un Punto copia de p. */
 public Punto(Punto p) {
 this(p.x,p.y);
 contadorPuntos++;
 }

 /** Consulta la coordenada x del Punto. */
 public double getX() { return this.x; }
 /** Consulta la coordenada y del Punto. */
 public double getY() { return this.y; }
 /** Consulta el número de coordenadas del Punto. */
 public int getDim() { return DIMENSION; }

 /** Actualiza la coordenada x del Punto. */
 public void setX(double x) { this.x = x; }
 /** Actualiza la coordenada y del Punto. */
 public void setY(double y) { this.y = y; }

 /** Incrementa la abscisa del Punto en x y
 * la ordenada del Punto en y. */
 public void mover(double x, double y) {
 this.x += x; this.y += y;
 }

 /** Devuelve un valor entero aleatorio en el
 * rango [0,2*(distancia del Punto al origen)]. */
 private int aleatorio() {
 return (int)(Math.random()*(2*this.distancia(ORIGEN)+1));
 }
}


```

Figura 5.10: Clase Punto.

```

/** Mueve el Punto a unas coordenadas aleatorias. */
public void moverAleat() {
 this.x += this.aleatorio();
 this.y += this.aleatorio();
}

/** Calcula la distancia del Punto a p. */
public double distancia(Punto p) {
 double x = p.x - this.x;
 double y = p.y - this.y;
 return Math.sqrt(x*x + y*y);
}

/** Retorna un Punto cuya distancia al origen es r,
 * y con ángulo a respecto al eje X. */
public static Punto polaresAPunto(double a, double r) {
 return new Punto(r*Math.cos(a),r*Math.sin(a));
}

/** Consulta la dimensión de los Puntos de la clase. */
public static int dimension() { return DIMENSION; }

/** Consulta el número de Puntos creados a lo largo
 * de la ejecución. */
public static int numPuntos() { return contadorPuntos; }
}

```

**Figura 5.10:** Clase Punto (cont.).

una clase incorporen una serie de métodos por omisión, que vienen heredados de la clase `Object`. Por tanto, en cualquier objeto puede hacerse uso de esos métodos, que incluyen operaciones como convertir el objeto a una cadena para su impresión (método `toString`) y comprobar si dos objetos son iguales (método `equals`).

Sin embargo, la mayor parte de las veces la funcionalidad de estos métodos por omisión no es la más apropiada para los objetos que se definen en una aplicación. Por tanto, lo habitual es *sobrescribirlos*, es decir, implementar un código distinto, para adecuarlos a cada clase. Para ello se debe incluir en la clase la declaración del método con la misma cabecera y con el cuerpo que implemente la funcionalidad deseada. Así, el perfil de `equals` es:

```
public boolean equals(Object o)
```

cuyo significado si no se sobrescribe es el de comprobar si el objeto en curso y `o` son exactamente el mismo:

```
public boolean equals(Object o) {
 return this == o;
}
```

```
1 /**
2 * Clase TestPunto: prueba de operaciones de la clase Punto.
3 * @author Libro IIP-PRG
4 * @version 2011
5 */
6 import java.util.Scanner;
7 import java.util.Locale;
8 public class TestPunto {
9 /** Método principal que prueba la clase Punto. */
10 public static void main(String[] args) {
11 System.out.println("Prueba de la clase Punto: " +
12 "puntos de dimensión " + Punto.dimension());
13
14 Scanner tec = new Scanner(System.in).useLocale(Locale.US);
15 System.out.println("Introduce las coordenadas de un punto: ");
16 System.out.print("Abscisa? "); double abs = tec.nextDouble();
17 System.out.print("Ordenada? "); double ord = tec.nextDouble();
18 Punto p = new Punto(abs,ord);
19 System.out.println("Se ha creado un punto de " + p.getDim() +
20 " dimensiones: (" + p.getX() + "," + p.getY() + ")");
21
22 System.out.println("Distancia entre el punto y el origen de " +
23 " coordenadas: " + p.distancia(Punto.ORIGEN));
24
25 System.out.println("Introduce las coordenadas de otro punto:");
26 System.out.print("Abscisa? "); abs = tec.nextDouble();
27 System.out.print("Ordenada? "); ord = tec.nextDouble();
28 Punto q = new Punto(abs,ord);
29
30 System.out.println("Se han creado "+Punto.numPuntos()+" puntos.");
31 System.out.print("Distancia entre los puntos introducidos: ");
32 System.out.println(p.distancia(q));
33 }
34 }
```

Figura 5.11: Ejemplo de uso de la clase Punto.

Para `toString` se tiene el perfil:

```
public String toString()
```

y por defecto devuelve una cadena de caracteres que incluye la clase del objeto en curso y un cierto código numérico. Hay que indicar que si una expresión de objeto cuyo valor es diferente de `null` aparece en un contexto de `String`, Java automáticamente le aplica el método `toString()`. Supóngase por ejemplo el siguiente código:

```
Punto p = null;
System.out.println("p es : " + p);
p = new Punto(3.0,4.0);
System.out.println("p es : " + p);
```

Produce en la salida estándar un resultado como:

```
p es null
p es Punto@426295eb
```

Para que estos métodos sean realmente útiles se sobrescribirán en cada clase en la forma que más convenga, de manera que `toString` sirva para describir textualmente los objetos de la clase. En cuanto a `equals`, el estándar Java recomienda que, salvo indicación en contrario, valga `true` si y sólo si el objeto en curso y o son de la misma clase (se puede comprobar con el operador `instanceof`), y sus atributos correspondientes coinciden.

Java aplicará entonces a los objetos de la clase los métodos sobreescritos, en lugar de los correspondientes métodos de `Object` que se aplicarían por defecto.

**Ejemplo 5.11.** La declaración de sobreescritura de estos métodos se puede incluir en la clase `Punto` como:

```
public boolean equals(Object o) {
 return o instanceof Punto &&
 this.x == ((Punto) o).x &&
 this.y == ((Punto) o).y;
}
```

en donde el casting le indica a Java que reconozca a `o` como un `Punto` y permita acceder a sus atributos.

Si `toString()` se sobrescribe como:

```
public String toString() {
 return "(" + this.x + "," + this.y + ")";
}
```

entonces, las líneas de código 19 y 20, de la figura 5.11 se podrían escribir:

```
System.out.println("Se ha creado un punto de " + p.getDim() +
 " dimensiones: " + p.toString());
```

o incluso

```
System.out.println("Se ha creado un punto de " + p.getDim() +
 " dimensiones: " + p);
```

## 5.6 Clases de utilidades

En ocasiones, se desea agrupar un conjunto de métodos porque ofrecen funcionalidades de alguna manera relacionadas entre sí, sin que necesariamente estén definiendo un nuevo tipo de dato. Por ejemplo, pueden ser métodos que calculen ciertas propiedades estadísticas, métodos relacionados con diversos formatos de salida en pantalla, etc. En general, es conveniente tener agrupados dichos métodos para poder incorporarlos de manera completa y sencilla a cualquier aplicación que quisiera hacer uso de sus funcionalidades. Este concepto también es extensible a las clases, permitiendo su agrupación bajo algún criterio, dando lugar a los *packages* (como se ha comentado en el capítulo 2 y como se ampliará en el capítulo 14). Por ejemplo, se podría crear un grupo de clases para tratar con diversos dispositivos de entrada de datos.

Muchos lenguajes de programación permiten este tipo de agrupación, llamándolas de diversas formas (bibliotecas, módulos, etc.). En Java, se denomina *Clases de utilidades* (en inglés, *Utility Classes*), a aquellas clases que agrupan misceláneas de métodos de utilidad general, y a menudo no incluyen atributos que den estructura a nuevo tipo de dato. Estas clases suelen incorporar un conjunto de atributos y métodos de clase (`static`) que dan la funcionalidad requerida. Si no incluyen atributos de instancia lo normal es que no se creen objetos de la misma, por lo que el constructor por defecto se suele hacer `private`.

Las clases de utilidades permiten agrupar un conjunto de métodos con una serie de características comunes en una clase. Cuando una aplicación haga uso intensivo de las funcionalidades que proporciona ese conjunto, el mecanismo de paquetes aporta una forma sencilla para incorporarlo a ésta organizadamente.

El acceso a métodos de una clase de utilidades está relacionado con el paquete en el que se sitúa la clase y la naturaleza de los métodos. Si la clase usuaria no pertenece al mismo paquete que la clase de utilidades, la clase usuaria debe importarla. Las clases en las que no se especifica el paquete de pertenencia se reúnen todas en un paquete por omisión (*anonymous*), con lo que son accesibles entre sí, excepto lo declarado como `private`.

Un ejemplo paradigmático de clase de utilidades dentro de las clases predefinidas de Java es la clase `Math` (capítulo 6), que ofrece una serie de métodos de clase que actúan sobre datos numéricos. Aunque define atributos estáticos públicos, `Math.PI` y `Math.E`, no define atributos dinámicos, habiéndose dejado privado el constructor por defecto.

Otro ejemplo es la clase `System`, que define como atributos estáticos públicos la entrada estándar `System.in`, la salida estándar `System.out` y la salida de error `System.err`. No tiene constructor público, pero proporciona métodos de utilidad general de interacción con el sistema, como la consulta del reloj, ejecución del *garbage collector*, finalización de la *JVM*, etc...

Estas dos clases pertenecen al paquete `java.lang` que es el único que no precisa ser importado, y que incluye otras clases, como `String`.

También el usuario puede desarrollar las clases de utilidades que se requieran en el desarrollo de sus aplicaciones.

**Ejemplo 5.12.** La siguiente clase de utilidades `UtilPunto` (figura 5.12) implementa una serie de operaciones útiles para una aplicación que necesite crear puntos a partir de datos introducidos por teclado, y realizar operaciones de desplazado, girado, escalado, etc. de puntos.

Los siguientes son ejemplos de llamadas a algunos métodos de la clase `UtilPunto` desde una clase usuaria:

```
Punto p = UtilPunto.leePunto();
Punto pBak = p;
double x = p.distancia(new Punto(0,0));
p = UtilPunto.desplazado(p,x+1.0,x-1.0);
p = UtilPunto.rotado(Math.PI/2.0);
```

## 5.7 Documentación de métodos: javadoc

Como ya se ha comentado en el capítulo 2 es importante documentar las clases adecuadamente para que, usando la herramienta `javadoc` [Ora11b], se pueda generar la documentación asociada. Así, cualquier programador que necesite usar una determinada clase sólo tendrá que consultar su documentación para saber cómo

```
/** Clase UtilPunto: define métodos estáticos para realizar
 * operaciones útiles con puntos del plano cartesiano.
 * @author Libro IIP-PRG
 * @version 2011
 */
import java.util.Scanner; import java.util.Locale;
public class UtilPunto {

 private UtilPunto() { } // Se oculta el constructor por defecto

 /** Devuelve un Punto cuyas coordenadas cartesianas
 * se leen de teclado. */
 public static Punto leePunto() {
 Scanner tec = new Scanner(System.in).useLocale(Locale.US);
 System.out.print("Abscisa? "); double abs = tec.nextDouble();
 System.out.print("Ordenada? "); double ord = tec.nextDouble();
 return new Punto(abs,ord);
 }

 /** Devuelve un Punto cuyas coordenadas polares
 * se leen de teclado. */
 public static Punto leePuntoPolar() {
 Scanner tec = new Scanner(System.in).useLocale(Locale.US);
 System.out.print("Radio? "); double rad = tec.nextDouble();
 System.out.print("Angulo? "); double ang = tec.nextDouble();
 return Punto.polaresAPunto(rad,ang);
 }

 /** Devuelve el punto medio situado entre otros dos dados. */
 public static Punto puntoMedio(Punto a, Punto b) {
 return new Punto((a.getX()+b.getX())/2,(a.getY()+b.getY())/2);
 }

 /** Devuelve un Punto resultado de la suma de otros dos dados. */
 public static Punto suma(Punto a, Punto b) {
 return new Punto(a.getX()+b.getX(),a.getY()+b.getY());
 }

 /** Devuelve un Punto resultado de la resta de otros dos dados. */
 public static Punto resta(Punto a, Punto b) {
 return new Punto(a.getX()-b.getX(),a.getY()-b.getY());
 }
}
```

Figura 5.12: Clase UtilPunto.

```

/** Devuelve un Punto con la abscisa de a incrementada en incX,
 * y la ordenada de a incrementada en incY. */
public static Punto desplazado(Punto a, double incX, double incY) {
 Punto p = new Punto(a);
 p.mover(incX,incY);
 return p;
}

/** Devuelve un Punto resultado de escalar a en un factor e. */
public static Punto escalado(Punto a, double e) {
 return new Punto(e*a.getX(),e*a.getY());
}

/** Devuelve un Punto resultado de girar a un ángulo alfa. */
public static Punto rotado(Punto a, double alfa) {
 double x = a.getX(), y = a.getY();
 double r = Math.sqrt(x*x +y*y), theta = Math.atan2(y,x);
 return Punto.polaresAPunto(r,theta+alfa);
}

```

**Figura 5.12:** Clase UtilPunto (cont.).

usarla. Los comentarios de documentación deben preceder a la definición de una clase, de un método u otro ítem público para que `javadoc` los genere.

En particular, la documentación de los métodos de una clase sirve para dos fines diferentes:

- Previamente a la implementación, especifica todas las características que se esperan del método, y dicha implementación debe ajustarse a lo especificado.
- Posterior e independientemente de la implementación, indica cómo usar los métodos, es decir, cuál es su perfil, qué condiciones especiales deben cumplir los parámetros, si las hubiere, y cuál es el resultado que se obtiene en cada caso de los datos.

Se debe evitar, en la medida de lo posible, incluir cualquier referencia a cómo se han implementado. Es en el cuerpo de los métodos donde deben aparecer los comentarios de implementación.

Un comentario de documentación de un método [Ora11a] incluye la descripción del método y la descripción de los parámetros y del resultado del método. Se debe incluir una descripción de cada parámetro y del resultado, precedidas por las etiquetas `@param` y `@return`, respectivamente. Las descripciones pueden extenderse por varias líneas y terminan en la siguiente etiqueta o al final del comentario. La etiqueta `@return` sólo aparece para aquellos métodos que devuelven un resultado.

A continuación, se muestran algunos de los métodos de la clase Punto precedidos por los comentarios de documentación correspondientes:

```
/*
 * Constructor de un nuevo Punto con coordenadas (abs,ord).
 * @param abs - double que es la abscisa del nuevo Punto.
 * @param ord - double que es la ordenada del nuevo Punto.
 */
public Punto(double abs, double ord) {
 x = abs; y = ord;
 numeroPuntos++;
}

/**
 * Consulta la coordenada x del Punto actual.
 * @return double que representa la x del Punto actual.
 */
public double getX() { return x; }

/**
 * Calcula la distancia entre el Punto actual y otro Punto dado.
 * @param p - Punto para calcular la distancia con el actual.
 * @return double que representa la distancia entre el Punto
 * actual y el Punto dado.
 */
public double distancia(Punto p) {
 double x = p.x - this.x;
 double y = p.y - this.y;
 return Math.sqrt(x*x + y*y);
}
```

En la figura 5.13 se muestra la documentación generada por javadoc para esta clase a partir de los comentarios introducidos.

## Class Punto

```
java.lang.Object
└ Punto
```

---

```
public class Punto extends java.lang.Object
```

Clase Punto: define un punto en el plano cartesiano, determinado por sus coordenadas x e y.

**Version:**

2011

**Author:**

Libro IIP-PRG

### Constructor Summary

[Punto\(double abs, double ord\)](#)

Constructor de un nuevo Punto con coordenadas (abs,ord).

### Method Summary

|        |                                    |
|--------|------------------------------------|
| double | <a href="#">distancia(Punto p)</a> |
|--------|------------------------------------|

Calcula la distancia del Punto actual a otro Punto dado.

|        |                        |
|--------|------------------------|
| double | <a href="#">getX()</a> |
|--------|------------------------|

Consulta la coordenada x del Punto.

|      |                                     |
|------|-------------------------------------|
| void | <a href="#">setX(double nuevaX)</a> |
|------|-------------------------------------|

Actualiza la coordenada x del Punto actual a una nueva.

Figura 5.13: Documentación de la clase Punto generada por javadoc.

## 5.8 Problemas propuestos

- Las Clases Programa `programaTri` que se muestran en la figuras 5.1 y 5.14 definen un triángulo en el plano cartesiano a partir de sus vértices, y muestran su perímetro en la salida estándar. Responder a las siguientes preguntas para comparar la calidad del código de ambas clases: ¿Cuál de ellas es más fácil de leer? ¿Cuál será más fácil de modificar? ¿Y de mantener?

```
/**
 * Clase ProgramaTri: define un triángulo en el plano cartesiano,
 * a partir de sus vértices y muestra su perímetro en la salida
 * estándar.
 * @author Libro IIP-PRG
 * @version 2011
 */
public class ProgramaTri {
 public static void main(String[] args) {
 Punto p1 = new Punto();
 p1.x = 2.5; p1.y = 3;
 Punto p2 = new Punto();
 p2.x = 2.5; p2.y = -1.2;
 Punto p3 = new Punto();
 p3.x = -1.5; p1.y = 1.4;
 System.out.println("Triángulo de vértices: ");
 System.out.println("(" + p1.x + "," + p1.y + ")");
 System.out.println("(" + p2.x + "," + p2.y + ")");
 System.out.println("(" + p3.x + "," + p3.y + ")");

 double lado12 = distancia(p1,p2);
 double lado23 = distancia(p2,p3);
 double lado13 = distancia(p1,p3),
 double perimetro = lado12 + lado23 + lado13;
 System.out.print("Perímetro: " + perimetro);
 }

 /** Calcula la distancia entre dos puntos. */
 static double distancia(Punto p, Punto q) {
 double dx = p.x - q.x;
 double dy = p.y - q.y;
 return Math.sqrt(dx*dx + dy*dy);
 }
}
```

Figura 5.14: Clase ProgramaTri. Versión 2.

2. Para estudiar el comportamiento en ejecución de los siguientes programas se sugiere utilizar la representación de registro de activación y pila de llamadas vista en el capítulo.

a) ¿Qué escribe en la salida estándar la ejecución del programa?

```
public class Ejemplo {
 public static void cambio(int j, int k) {
 int aux = j;
 j = k;
 k = aux;
 System.out.println("Dentro: " + j + " " + k);
 } // de cambio

 public static void inc(int j, int k) {
 j++;
 k++;
 System.out.println("Dentro: " + j + " " + k);
 } // de inc

 public static void main(String[] args) {
 int j = 1335;
 int k = 3672;
 System.out.println("Antes: " + j + " " + k);
 cambio(j,k);
 inc(k,j);
 System.out.println("Después: " + j + " " + k);
 } // de main
} // de Ejemplo
```

b) ¿Qué se escribe en la salida estándar cuando se ejecuta el programa?

Se supone la existencia de la clase Punto del apartado 5.5.

```
public class Ejemplo2 {
 public static void main(String[] args) {
 Punto p1 = new Punto(1.0,-1.0);
 Punto p2 = new Punto();
 System.out.print("Antes de la llamada ");
 System.out.println("los puntos son: " + p1 +" y "+ p2);
 cambioPunto(p1,p2);
 System.out.print("Después de la llamada ");
 System.out.println("los puntos son: " + p1 +" y "+ p2);
 } // de main

 private static void cambioPunto(Punto a, Punto b) {
 Punto aux = a;
 a = b;
 b = aux;
 }
} // de Ejemplo2
```

- c) Escribir de nuevo el método `cambioPunto(Punto, Punto)` para que en el programa anterior se efectúe realmente el intercambio de los valores de los atributos de los dos objetos `Punto` que se reciben como parámetros.
3. ¿Por qué los métodos consultores y modificadores de los valores de los atributos (métodos gets y sets) se deben definir como métodos públicos?
4. Se tiene definida la siguiente clase:

```
/**
 * Clase Hora.
 * @author Libro IIP-PRG
 * @version 2011
 */
public class Hora {
 private int hora;
 private int minutos;

 /** Crea una Hora con 0 horas y 0 minutos. */
 public Hora() { }

 /** Crea una Hora con valores hora y minutos. */
 public Hora(int hora, int minutos) {
 this.hora = hora; this.minutos = minutos;
 }

 /** Consulta la hora de la Hora actual. */
 public int getHora() { return this.hora; }

 /** Consulta los minutos de la Hora actual. */
 public int getMinutos() { return this.minutos; }

 /** Devuelve true si la Hora es anterior a h. */
 public boolean anterior(Hora h) {
 return this.hora<h.hora ||
 (this.hora==h.hora && this.minutos<h.minutos);
 }

 /** Devuelve la Hora en el formato "hh:mm". */
 public String toString() {
 String h = "0" + this.hora;
 h = h.substring(h.length()-2);
 String m = "0" + this.minutos;
 m = m.substring(m.length()-2);
 return h + ":" + m;
 }
}
```

Reescribir el código de la figura 5.6 a partir de la línea 16 utilizando únicamente métodos de la clase `Hora`.

5. Completar la clase `Hora` del problema anterior con los métodos modificadores y con la sobrescritura del método `equals(Object)`.
6. En el capítulo 2 se introdujo la clase `Circulo`. En ella se utiliza la pareja de atributos enteros `centroX` y `centroY` para representar las coordenadas de su centro. Habiendo visto la clase `Punto`, parece lógico redefinir ahora la clase `Circulo` utilizando un elemento de dicho tipo para representar el centro del `Circulo`. A estos efectos, se pide:
  - a) Sustituir en la definición de la clase `Circulo` los atributos con los que se representa su centro por un único atributo `centro` de tipo `Punto`.
  - b) Redefinir todos los métodos de la clase `Circulo` que sea necesario siguiendo la nueva definición de `centro`. Se debe tener en cuenta que los atributos `x` e `y` de un `Punto`, son valores de tipo `double` y no `int`.
7. Redefinir la clase `Cuadrado` sustituyendo, al igual que en el problema anterior, la representación del centro por una variable de tipo `Punto`.
8. Añadir a la clase `Cuadrado`, tras haberla modificado en el problema anterior, un nuevo constructor `Cuadrado(Punto, Punto)` que permita definir un `Cuadrado` a partir de dos esquinas, situadas en diagonal, representadas como objetos de tipo `Punto`.
9. Escribir una clase `Rectangulo`, idéntica a la clase `Cuadrado` anterior, pero representando ahora su `base` y `altura`, en lugar del `lado`. Deberá incluir, al igual que en el problema anterior, un constructor que recibirá como argumentos dos esquinas en diagonal representadas como objetos de tipo `Punto`.
10. Sobrescribir convenientemente el método `boolean equals(Object)` en las clases `Circulo`, `Cuadrado` y `Rectangulo`.
11. Escribir una clase `Triangulo` que permita representar un triángulo a partir de sus tres vértices (definidos como objetos `Punto`) y que incluya métodos constructores, consultores y modificadores, un método que calcule el área de un triángulo, un método `toString()` que obtenga una representación de un objeto `Triangulo` como un `String` y un método `equals` para comparar dos objetos `Triangulo`.
12. Escribir una Clase Programa cuyo `main` permita probar la funcionalidad de la clase anterior.

13. Añadir los siguientes métodos a la clase **Hora** del problema 4:
  - a) Método constructor **public Hora()** en lugar del constructor por defecto, que inicie la **Hora** creada a la hora actual (Greenwich, o UTC). Se sugiere utilizar el método **static long currentTimeMillis()** de la clase predefinida **System**, que devuelve el número de milisegundos transcurridos desde el 1 de Enero de 1970 (UTC).
  - b) Método **public int minTotales()**, que devuelva el número de minutos que han transcurrido del día hasta la hora en curso. Por ejemplo, para las 07:45 debe devolver 455, y para las 17:18 debe devolver 1038.
  - c) Método **public int comparar(Hora h)**, que devuelva un valor negativo si la hora en curso es anterior a **h**, 0 si son iguales, y un valor positivo si **h** es anterior. En valor absoluto, el valor devuelto debe ser el número de minutos de diferencia que hay entre ambas horas.
14. Escribir una Clase Programa que emplee la clase **Hora** con los métodos del ejercicio previo y que en su **main** escriba la hora actual, pida dos horas al usuario y las escriba en la salida mostrando además la diferencia en minutos entre ellas.

## Más información

- [AGH01] K. Arnold, J. Gosling, and D. Holmes. *El lenguaje de programación Java*. Addison-Wesley, 2001. Capítulo 2.
- [Ora11a] Oracle. *How to Write Doc Comments for the Javadoc Tool*, 2011. URL: <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>.
- [Ora11b] Oracle. *Javadoc Tool*, 2011. URL: <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>.
- [Ora11d] Oracle. *The Java<sup>TM</sup> Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Classes and Objects.
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010. Capítulos 4 y 5.
- [Sch07] H. Schildt. *Fundamentos de Java*. McGraw-Hill, 2007. Capítulo 4 (4.4, 4.5, 4.6 y 4.7).

# Capítulo 6

## Algunas clases predefinidas: **String, Math.** Clases envolventes

Una de las ventajas de los lenguajes de programación actuales es que ofrecen bastantes elementos predefinidos de forma que no es necesario reinventarlos si estos se necesitan. El lenguaje Java contiene un número elevado de clases ya definidas que abarcan, por ejemplo, aspectos relativos a la computación numérica, a la representación gráfica, la entrada/salida, la ejecución paralela, conexiones remotas y un largo etcétera.

En este capítulo se contemplan algunas de las clases existentes en Java. En concreto, se describen las clases **String** y **Math** que permiten, respectivamente, operar con cadenas de caracteres y realizar cálculos matemáticos avanzados; así como las denominadas clases *envolventes* (o *envoltorios*) que ofrecen funcionalidades adicionales para operar con valores elementales tras su transformación en objetos. Todas estas clases forman parte del paquete `java.lang` y, puesto que este paquete se importa por defecto, los elementos públicos de sus clases son accesibles directamente desde cualquier otra.

### 6.1 La clase String

La clase **String** permite crear y manipular cadenas de caracteres en Java. Aunque esta clase ha sido ya utilizada en parte en capítulos anteriores, es interesante revisarla ahora tomando en consideración lo visto en los capítulos 4 y 5 (referencias y métodos).

### 6.1.1 Aspectos básicos

Los objetos de la clase *String* permiten representar secuencias de caracteres Unicode cualesquiera. Los literales de la clase se escriben entre comillas dobles como, por ejemplo, "Esto es un literal de la clase *String*".

En Java se distingue entre caracteres individuales y cadenas de caracteres. Como los primeros pertenecen al tipo elemental *char* mientras que los segundos son elementos de la clase *String*, presentan entre ellos las diferencias que se dan entre valores elementales y objetos que se examinaron en el capítulo 4. Algunas de estas diferencias se examinan con más detalle, más adelante, en este mismo punto.

Otra diferencia significativa en Java entre los caracteres y las cadenas consiste en cómo se expresan sus literales. Mientras que los literales de la clase *String* se escriben, como ya se ha dicho, entre comillas dobles, los del tipo *char* se escriben entre comillas simples. Así, 'z' es un carácter mientras que "z" es una cadena.

#### *Declaración y creación*

En el momento de la declaración, como ocurre con cualquier variable de tipo referencia, el objeto referenciado aún no se ha creado.

```
String st1; // no existe objeto referenciado
```

Existen dos formas de crear un objeto *String*. La primera, es una forma directa, exclusiva de las *String*, en la que basta con utilizar un literal del tipo.

La segunda, al igual que en cualquier otra clase, consiste en utilizar alguno de sus constructores<sup>1</sup>. Lo siguiente es una muestra de declaración y creación de varias *String* utilizando primero el método directo y, a continuación, uno de sus constructores:

```
String st1 = "Esto es un ejemplo de String";
String st2 = new String("Esto es un ejemplo de String");
```

Tras ejecutar las instrucciones anteriores, las variables *st1* y *st2* referencian a dos objetos *String* distintos.

### 6.1.2 Concatenación

Una característica importante de las *String* en Java es que son *inmutables*, lo que significa que, una vez creadas e inicializadas, no es posible alterar su contenido

---

<sup>1</sup>Se puede consultar al respecto el API del Java en [Ora11c].

(añadiendo, eliminando o cambiando caracteres individuales de las mismas), por lo que se hace necesario operar sobre ellas generando, cuando sea necesario, nuevas `String`.

La operación básica que se utiliza para obtener nuevas `String` a partir de otras ya existentes es la *concatenación* que consiste en generar una nueva `String` a partir del contenido de otras dos, ya existentes, situando los caracteres de una detrás de los de la otra.

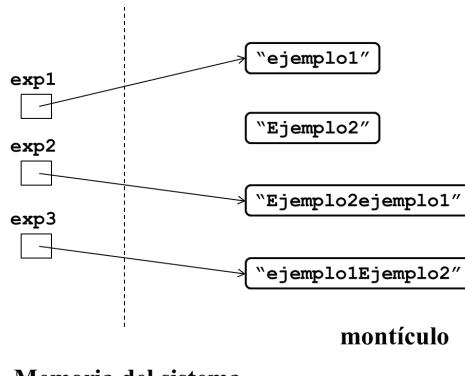
Aunque, como puede observarse en el *API* del Java ([Ora11c]), existe un método especializado para realizarla (`concat(String)`), lo más habitual es utilizar en su lugar los operadores `+` y `+=` que para las `String` representan, respectivamente, la concatenación y la asignación con concatenación.

En cualquier caso, el resultado de una concatenación es un nuevo objeto `String` construido a partir de los objetos `String` a concatenar.

**Ejemplo 6.1.** Considérese el siguiente código:

```
String exp1 = "ejemplo1";
String exp2 = "Ejemplo2";
String exp3 = exp1+exp2; // exp3 referencia a "ejemplo1Ejemplo2"
 exp2 += exp1; // exp2 referencia a "Ejemplo2ejemplo1"
```

Tras la ejecución de estas instrucciones, la situación de la memoria es la que se muestra en la figura 6.1. Se tienen tres objetos de tipo `String` "`ejemplo1`", "`Ejemplo2ejemplo1`" y "`ejemplo1Ejemplo2`" referenciados, respectivamente, por las variables `exp1`, `exp2` y `exp3`, y un objeto "`Ejemplo2`" (referenciado inicialmente por `exp2`) desreferenciado.



**Figura 6.1:** Las variables `exp1`, `exp2` y `exp3` referencian tres `String` distintos. El `String` "`Ejemplo2`" queda desreferenciado.

La clase **String** es el único tipo no elemental que admite operadores, los de concatenación, que están sobrecargados ya que pueden utilizarse, con un significado distinto, sobre valores numéricos.

Si en una expresión se utiliza el operador + y algún operando es un **String**, toda la operación se convierte en una concatenación de cadenas. Recuérdese, además, que el operador + es un operador asociativo por la izquierda evaluándose, por lo tanto, la expresión en la que aparece de izquierda a derecha<sup>2</sup>.

**Ejemplo 6.2.** En el código que sigue, al evaluar la expresión `1 + 2 + "a"` de la primera asignación, la suma de 1 y 2 se concatena con "a", dando como resultado el objeto "3a". En la segunda asignación, primero se evalúa la expresión entre paréntesis, cuyo resultado es "2a", que se concatena con el 1 y da como resultado "12a". Por último, en la expresión de la tercera asignación se concatena el objeto referenciado por `s2` ("12a") con el 2. La variable `s2` referencia ahora a "12a2" (quedando el objeto "12a" desreferenciado).

```
String s1 = 1 + 2 + "a"; // s1 referencia a "3a"
String s2 = 1 + (2 + "a"); // s2 referencia a "12a"
s2 += 2; // s2 pasa a referenciar a "12a2"
```

### 6.1.3 Formación de literales

Como ya se ha visto, los literales de la clase **String** se forman mediante secuencias de caracteres Unicode situadas entre comillas dobles. Además, en la creación de literales de tipo **String** es posible utilizar tanto las secuencias hexadecimales con las que se puede expresar (mediante su código) un carácter Unicode cualquiera<sup>3</sup>, como las secuencias de escape para representar tabuladores, retornos de carro y otros elementos de composición<sup>4</sup>, de forma que sea posible incluir los mismos en la creación de las cadenas.

Del mismo modo, es necesario utilizar las secuencias de escape correspondientes cuando se necesite incluir en una **String** alguno de los caracteres que tienen un significado especial como puede ser, por ejemplo, las comillas. Por ejemplo:

```
System.out.println("Hola.\n\" Bienvenidos al mundo de los \tString.\\"");
```

donde el carácter de escape delante de las letras 'n' y 't' cambia su significado habitual, dándoles el de retorno de carro y tabulación, respectivamente. Asimismo,

---

<sup>2</sup>Según las reglas de precedencia de los operadores, descritas en el capítulo 3.

<sup>3</sup>Véase el tipo carácter en el capítulo 3.

<sup>4</sup>Véase la tabla 3.7, en el mismo capítulo.

la barra delante de las dobles comillas les da significado de meros caracteres. El resultado de la ejecución de la instrucción anterior es:

| Salida Estándar                                |
|------------------------------------------------|
| Hola.<br>"Bienvenidos al mundo de los String." |

En la escritura de programas, una peculiaridad de los literales de tipo **String** es que deben escribirse en una misma línea en el código ya que, de lo contrario, el compilador señalará un error de elemento no concluido adecuadamente. Por lo tanto, si se desea insertar un retorno de carro en una cadena, escrita en un programa, debe utilizarse el carácter '\n', mientras que si se desea escribir una cadena que ocupe más de una línea en el programa, se tendrán que escribir las partes que la formen entre comillas dobles para, después, concatenarlas, así, por ejemplo:

```
String largo = "Este es un ejemplo de la declaración de " +
 "un String que no cabe en una línea de código";
```

#### 6.1.4 Comparación

Para la comparación de variables **String**, los operadores == y != tienen el significado ya visto en cuanto a la comparación de valores de tipos no elementales; esto es, comparan referencias, no los objetos **String** que representan. Además, los operadores relacionales (>, >=, <, <=) que se utilizan para comparar valores de tipos elementales no están definidos para el tipo **String**.

Para comprobar la igualdad de dos objetos en Java se utiliza el método **equals()**, que es el que se utilizará también para determinar la igualdad de dos **String**. De forma más particular, ya que los elementos de la clase **String** admiten una ordenación total entre ellos, se puede utilizar para comparar dos cadenas el método **compareTo()**. En la tabla 6.1 se muestra un resumen de estos métodos.

|                                               |                                                                                                                                                                                                                                 |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>public boolean equals (Object x)</code> | Devuelve true si y sólo si el argumento es un objeto <b>String</b> que representa la misma secuencia de caracteres que el <b>String</b> actual.                                                                                 |
| <code>public int compareTo (String x)</code>  | Compara dos <b>String</b> lexicográficamente. Devuelve un entero indicando si el <b>String</b> actual es mayor (el resultado es > 0), igual (el resultado es = 0) o menor (el resultado es < 0) que el <b>String</b> argumento. |

Tabla 6.1: Métodos **equals** y **compareTo** de la clase **String**.

La comparación de cadenas, como tal, se basa en el *orden lexicográfico*. Es decir, las cadenas se comparan entre sí del mismo modo que en un diccionario, pero teniendo en cuenta que dos caracteres cualesquiera son distintos entre sí cuando tienen diferente código Unicode.

Para que dos cadenas sean iguales, han de tener el mismo número de caracteres y cada carácter de cualquiera de ellas debe ser idéntico al carácter en la misma posición de la otra.

El siguiente fragmento de código muestra, con algunos ejemplos, el uso de los métodos `equals()` y `compareTo()`:

```
String s1 = "Hola", s2 = "Hello", s3;
boolean iguales;

iguales = s1==s2; // iguales = false
iguales = s1.equals(s2); // iguales = false
s3 = s1;
iguales = s3==s1; // iguales = true
iguales = s3.equals(s1); // iguales = true
int comp1 = s3.compareTo(s1); // comp1 = 0
int comp2 = s3.compareTo(s2); // comp2 > 0
int comp3 = s2.compareTo(s3); // comp3 < 0
```

### 6.1.5 Algunos métodos

La clase `String` define un gran número de métodos adicionales para operar sobre objetos pertenecientes a la misma, aportando con ello las funcionalidades más comunes relativas a elementos de dicho tipo. En la tabla 6.2 se muestran algunos de ellos; una referencia exhaustiva a estos métodos puede encontrarse en la ayuda *on line* del lenguaje [Ora11c].

Algunos métodos de la clase `String` como, por ejemplo, métodos para encontrar caracteres o subcadenas dentro de una cadena (`charAt(int)`, `indexOf(String, int)`, `substring(int, int)`, etc.), dependen de la posición que ocupan los caracteres en la cadena. Los caracteres se numeran empezando por el 0 (no por el 1). En la figura 6.2 se muestra la numeración de caracteres del `String` "Hola, Adios! 58". Obsérvese que los caracteres están numerados a partir de la posición cero y que todos, tanto los espacios, los signos de puntuación, como las letras y los dígitos cuentan en la numeración, ya que también son caracteres.

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| H | o | 1 | a | , |   | A | d | i | o | s  | !  |    | 5  | 8  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Figura 6.2: Numeración de los caracteres de un `String`.

|                                                                                                                                                                                                                                  |                                                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>public int length ()</code>                                                                                                                                                                                                | Devuelve la longitud del <i>String</i> actual (número de caracteres).                                                                                                                       |
| <code>public String concat (String cad)</code>                                                                                                                                                                                   | Devuelve un nuevo <i>String</i> resultado de concatenar al final del <i>String</i> actual el <i>String</i> <code>cad</code> .                                                               |
| <code>public String substring (int ini,int fin)</code><br><code>public String substring (int ini)</code>                                                                                                                         | Devuelve la subcadena comprendida entre las posiciones <code>ini</code> y <code>fin-1</code> (el final) del <i>String</i> actual.                                                           |
| <code>public void trim ()</code>                                                                                                                                                                                                 | Elimina los espacios en blanco que puedan existir al principio o al final del <i>String</i> actual.                                                                                         |
| <code>public char charAt (int i)</code>                                                                                                                                                                                          | Devuelve el carácter situado en la <code>i</code> -ésima posición del <i>String</i> actual.                                                                                                 |
| <code>public String toUpperCase ()</code>                                                                                                                                                                                        | Devuelve un nuevo <i>String</i> resultado de convertir a mayúsculas o minúsculas todos los caracteres del <i>String</i> actual.                                                             |
| <code>public int indexOf (String cad)</code><br><code>public int lastIndexOf (String cad)</code>                                                                                                                                 | Devuelve la posición de la primera (última) aparición de <code>cad</code> en el <i>String</i> actual o -1 si no existe.                                                                     |
| <code>public int indexOf (String cad,int i)</code><br><code>public int lastIndexOf (String cad,int i)</code>                                                                                                                     | Devuelve la posición de la primera (última) aparición de <code>cad</code> en el <i>String</i> actual a partir de la posición <code>i</code> o -1 si no existe.                              |
| <code>public String startsWith (String cad)</code><br><code>public String endsWith (String cad)</code>                                                                                                                           | Devuelve <code>true</code> si el <i>String</i> actual comienza (termina) por <code>cad</code> .                                                                                             |
| <code>public String replace (CharSequence t,<br/>                          CharSequence r)</code><br><br>Un <i>CharSequence</i> es una secuencia de valores de tipo <i>char</i> y, en su lugar, se puede usar un <i>String</i> . | Devuelve un nuevo <i>String</i> resultado de sustituir cada substring del <i>String</i> actual que coincide con la secuencia de caracteres <code>t</code> por la secuencia <code>r</code> . |
| <code>public boolean contains (CharSequence t)</code>                                                                                                                                                                            | Devuelve <code>true</code> si el <i>String</i> actual incluye a la secuencia <code>t</code> .                                                                                               |

Tabla 6.2: Algunos métodos de la clase *String*.

A continuación se muestra, mediante ejemplos, el uso de algunos métodos de la clase *String*:

```
String st1 = "Ejemplo 1";
String mayus = st1.toUpperCase(); // mayus es "EJEMPLO 1"
String minus = st1.toLowerCase(); // minus es "ejemplo 1"
int longitud = st1.length(); // longitud es 9
char caracter = st1.charAt(1); // caracter es 'j'
 // (el de la posición 1)
String sub = st1.substring(3,5); // sub es "mpl"
String st = st1.concat(" y 2"); // st es "Ejemplo 1 y 2"
boolean b = st1.startsWith("Eje"); // b es true
int inicio = st1.indexOf("mpl"); // inicio es 3, posición
 // de "mpl" en "Ejemplo 1"
int desde = st1.indexOf("mpl",2); // desde es 3
String st2 = " Ejemplo 2 ";
int ultima = st2.lastIndexOf(" "); // ultima es 12
String noblanc = st2.trim(); // noblanc es "Ejemplo 2"
```

## 6.2 La clase Math

En el capítulo 3 se presentó un grupo de operadores aritméticos básicos con los que realizar operaciones sencillas con datos de tipo numérico. En Java es posible, además, realizar operaciones matemáticas más complejas (tales como funciones trigonométricas, logarítmicas, potencias, etc.) usando la clase predefinida *Math* que se encuentra incluida en el paquete *java.lang* que, como ya se ha dicho, es importado por defecto por cualquier programa.

### 6.2.1 Constantes y métodos

En concreto, *Math* es una Clase de Utilidades que implementa un conjunto de constantes y métodos estáticos para poder realizar operaciones matemáticas avanzadas sobre expresiones de tipo numérico. Sus atributos y métodos públicos, al ser estáticos, se deben utilizar anteponiéndoles el identificador de la clase seguido de un punto. Por ejemplo, *Math.E* y *Math.PI* son dos constantes que definen, respectivamente, los valores en coma flotante de los números *e* y  $\pi$  (tabla 6.3).

La clase *Math* define un conjunto importante de métodos que, en atención a su funcionalidad pueden resumirse como operaciones matemáticas de carácter básico, de generación de valores aleatorios, exponenciales y logarítmicas, y, finalmente, trigonométricas.

|                                      |                                                                                                                                    |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <code>public static double E</code>  | El valor de tipo <code>double</code> más cercano al número $e$ , base del logaritmo natural.                                       |
| <code>public static double PI</code> | El valor de tipo <code>double</code> más cercano al número $\pi$ , relación entre la longitud de una circunferencia y su diámetro. |

**Tabla 6.3:** Constantes públicas de la clase **Math**.

En las tablas 6.4 a 6.7, se muestran de forma resumida algunos de estos métodos. El conjunto completo de operaciones se puede consultar en el *API* de Java [Ora11c] donde, junto con las características de cada método, se establecen con detalle las condiciones de uso del mismo, denominadas *precondiciones*, así como los errores o excepciones que se pueden provocar si se hace caso omiso de ellas.

|                                                                                                                                                                                                                                    |                                                                                                         |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| <code>public static double abs (double a)</code><br><code>public static float abs (float a)</code><br><code>public static long abs (long a)</code><br><code>public static int abs (int a)</code>                                   | Devuelve el valor absoluto de $a$ .                                                                     |
| <code>public static double max (double a, double b)</code><br><code>public static float max (float a, float b)</code><br><code>public static long max (long a, long b)</code><br><code>public static int max (int a, int b)</code> | Devuelve el mayor de sus dos parámetros.                                                                |
| <code>public static double min (double a, double b)</code><br><code>public static float min (float a, float b)</code><br><code>public static long min (long a, long b)</code><br><code>public static int min (int a, int b)</code> | Devuelve el menor de sus dos parámetros.                                                                |
| <code>public static double ceil (double a)</code>                                                                                                                                                                                  | Devuelve el número real correspondiente a $\lceil a \rceil$ (entero más pequeño mayor o igual a $x$ ).  |
| <code>public static double floor (double a)</code>                                                                                                                                                                                 | Devuelve el número real correspondiente a $\lfloor a \rfloor$ (entero más grande menor o igual a $x$ ). |
| <code>public static long round (double a)</code><br><code>public static int round (float a)</code>                                                                                                                                 | Devuelve el número entero más cercano a $a$ .                                                           |

**Tabla 6.4:** Algunos de los métodos matemáticos de carácter básico en **Math**.

|                                             |                                                       |
|---------------------------------------------|-------------------------------------------------------|
| <code>public static double random ()</code> | Devuelve un número aleatorio en el rango $[0.0, 1.0[$ |
|---------------------------------------------|-------------------------------------------------------|

**Tabla 6.5:** Método para obtener un valor aleatorio en **Math**.

|                                                            |                                                       |
|------------------------------------------------------------|-------------------------------------------------------|
| <code>public static double log (double a)</code>           | Devuelve el logaritmo natural (en base $e$ ) de $a$ . |
| <code>public static double exp (double a)</code>           | Devuelve la potencia $e^a$ .                          |
| <code>public static double pow (double a, double b)</code> | Devuelve la potencia $a^b$ .                          |
| <code>public static double sqrt (double a)</code>          | Devuelve la raíz cuadrada positiva de $a$ .           |

**Tabla 6.6:** Algunos de los métodos exponenciales y logarítmicos en *Math*.

|                                                                                                                                                                                                                                                                                                                         |                                                                                                               |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <code>public static double sin (double a)</code><br><code>public static double cos (double a)</code><br><code>public static double tan (double a)</code><br><code>public static double asin (double a)</code><br><code>public static double acos (double a)</code><br><code>public static double atan (double a)</code> | Devuelve el seno (coseno, tangente, arcoseno, arcocoseno, arcotangente) del ángulo $a$ expresado en radianes. |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|

**Tabla 6.7:** Algunos de los métodos trigonométricos en *Math*.

En el siguiente fragmento de código se muestra el uso de algunos de los métodos de la clase, agrupados según su funcionalidad:

```
double x, y, abs, max, pot, raíz, log, sen, ceil, flr, alf, tan;
x = 2.0;
y = 5.0;

// Exponentiales y logarítmicos:
pot = Math.pow(x, y); // pot es 32.0
raíz = Math.sqrt(x); // raíz es 1.4142135623730951
log = Math.log(y); // log es 1.6094379124341003

// Trigonométricos:
sen = Math.sin(Math.PI/2); // sen es 1.0
alf = Math.arcsin(sen); // alf es 1.5707963267948966
tan = Math.tan(Math.PI/2); // tan es 1.633123935319537E16

// Matemáticos básicos:
abs = Math.abs(-x); // abs es 2.0
max = Math.max(x,y); // max es 5.0
ceil = Math.ceil(3.76); // ceil es 4.0
flr = Math.floor(3.76); // flr es 3.0
long round1 = Math.round(3.76); // round1 es 4
long round2 = Math.round(3.45); // round2 es 3
```

## 6.2.2 Algunos ejemplos

### *Redondeo de valores reales*

En el ejemplo de la figura 6.3 se muestra un método estático que, dado cierto número real (`numero`) y un entero (`numDec`), no negativo, devuelve el valor original redondeado al número de decimales expresado en `numDec`.

```
/**
 * Redondea un valor a cierto número de decimales.
 * @param numero - valor a ser redondeado.
 * @param numDec - número de decimales. Debe ser >= 0.
 * @return numero redondeado a numDec.
 */
public static double redondeoNumDec(double numero, int numDec) {
 double numRedondeado = numero*Math.pow(10,numDec);
 numRedondeado = Math.round(numRedondeado);
 numRedondeado /= Math.pow(10,numDec);
 return numRedondeado;
}
```

**Figura 6.3:** Redondeo de un valor a cierto número de decimales.

A continuación se muestran dos instrucciones en las que se realizan llamadas al método `redondeoNumDec(double,int)` utilizando argumentos distintos. En la segunda puede apreciarse que el valor devuelto es el mismo que el original cuando la cantidad de decimales a la que se quiere redondear supera a los del propio número.

```
double x = redondeoNumDec(123.987654321,4); // x vale 123.9877
double y = redondeoNumDec(123.963,5); // y vale 123.963
```

### *Generación de números aleatorios*

En la figura 6.4 se muestra un método que, dadas dos cotas enteras que recibe como argumentos, devuelve un número entero aleatorio de entre los comprendidos en el intervalo, de forma equiprobable.

Para hacerlo, utiliza el método `Math.random()` que, como ya se ha dicho, devuelve cada vez que es ejecutado un valor aleatorio<sup>5</sup> en el rango  $[0.0, 1.0[$ .

---

<sup>5</sup>En realidad se trata de un valor *pseudoaleatorio*, ya que la sucesión de números que se generan ejecutando repetidamente `Math.random()` tiene un comportamiento estadístico similar al de una sucesión de números al azar, pero es en realidad predecible o determinista.

```

1 /**
2 * Devuelve un valor aleatorio en el intervalo [a,b].
3 * @param a - Un extremo del intervalo.
4 * @param b - El otro extremo del intervalo.
5 * @return Valor aleatorio en el intervalo [a,b].
6 */
7 public static int aleatorioEnIntervalo(int a, int b) {
8 // Ordenar los extremos
9 int aux1 = a; int aux2 = b;
10 a = Math.min(aux1,aux2);
11 b = Math.max(aux1,aux2);
12
13 double x0 = Math.random();
14 // x0 contiene un número real en el rango [0,1[
15
16 int cantidadDeNumerosEntreAyB = b-a+1;
17 double x1 = x0 * cantidadDeNumerosEntreAyB;
18 // x1 es un real en el rango [0,cantidadDeNumerosEntreAyB[
19
20 double x2 = a + x1;
21 // x2 es un real en el rango
22 // [a,a+cantidadDeNumerosEntreAyB[==[a,a+(b-a+1)[==[a,b+1[
23
24 int valor = (int)x2;
25 // valor es un entero en el rango [a,b+1[==[a,b]
26 return valor;
27 }

```

**Figura 6.4:** Obtención de un valor aleatorio en el intervalo  $[a, b]$ .

Aunque el código es de lectura sencilla, se puede resumir en que, una vez se obtiene un valor aleatorio en coma flotante (línea 13), se multiplica por el número de valores existentes en el intervalo (línea 17) sumándosele, a continuación, el valor del extremo inferior de dicho intervalo (línea 20). El valor obtenido en coma flotante, en el rango  $[a, b + 1[$  es transformado mediante un *casting* a un valor entero en el rango  $[a, b]$  que es, finalmente, devuelto.

En general, dadas  $a$  y  $b$  dos variables (o expresiones) de tipo `int` cualesquiera, tales que  $a \leq b$ , se puede obtener un entero aleatorio  $x$  en el rango de enteros  $[a, b]$  mediante una expresión como la siguiente:

$$(int)(Math.random()*(b-a+1) + a)$$

## 6.3 Clases Envoltorios

En Java no existen referencias a valores pertenecientes a tipos primitivos; por lo que, por ejemplo, la única forma de tener una referencia a un valor de tipo `int` o `double` consiste en *envolverlo* en un objeto.

Esta última solución la adopta el propio lenguaje, predefiniendo para los tipos básicos las que se conocen como *clases envoltorios* o *wrapper classes* (en inglés, *wrapper classes*).

Para cada uno de los tipos básicos estudiados en el capítulo 3 existe una clase envolvente que permite tratar los tipos básicos como objetos. Los nombres de estas clases son: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` y `Boolean`.

Estas clases pertenecen al paquete `java.lang` por lo que, como ya se ha dicho, pueden utilizarse, sin tener que importarlas explícitamente, en cualquier programa.

Cada objeto de alguna de estas clases contiene un único atributo del tipo básico pero, a diferencia del tipo original, está dotado de algunas funcionalidades adicionales (proporcionadas en forma de constantes y métodos) que, debido a su utilidad, es necesario conocer. Además, todas las clases envoltorios son inmutables, lo que quiere decir que una vez creado un objeto de la clase, no puede ser modificado<sup>6</sup>.

En ocasiones, es conveniente la conversión de un dato de tipo básico a un objeto de tipo envolvente para disponer de alguna de esas funcionalidades adicionales a las que se hacía referencia; por ejemplo, las constantes `MAX_VALUE` y `MIN_VALUE` están definidas en todas las clases envoltorios de tipo numérico. Gracias a ellas es fácil conocer cuáles son en Java los valores máximo y mínimo que se pueden representar en el tipo.

Los tipos reales también disponen de las constantes `POSITIVE_INFINITY` y `NEGATIVE_INFINITY` para representar los valores infinitos, así como de la constante `Double.NaN` (*Not a Number*) que es el resultado de una expresión como, por ejemplo, `Double.POSITIVE_INFINITY/Double.POSITIVE_INFINITY`.

Una funcionalidad importante, mostrada en la tabla 6.8, es la proporcionada por algunas operaciones estáticas que permiten conversiones de valores representados como `String` a valores numéricos de tipos elementales. Gracias a ellos será sencillo, como se verá más adelante, leer un valor como una cadena de caracteres, por ejemplo en una caja de texto (`TextField`), para obtener a continuación el valor numérico correspondiente.

Los tipos envolventes y sus correspondientes tipos básicos son compatibles entre sí, lo que permite simplificar de forma notable la escritura de código.

---

<sup>6</sup>De hecho, no existen métodos modificadores en ninguna de ellas.

| Método                                  | Tipo del valor devuelto |
|-----------------------------------------|-------------------------|
| <code>Byte.parseByte(String)</code>     | <code>byte</code>       |
| <code>Short.parseShort(String)</code>   | <code>short</code>      |
| <code>Integer.parseInt(String)</code>   | <code>int</code>        |
| <code>Long.parseLong(String)</code>     | <code>long</code>       |
| <code>Float.parseFloat(String)</code>   | <code>float</code>      |
| <code>Double.parseDouble(String)</code> | <code>double</code>     |

**Tabla 6.8:** Métodos de las clases envolventes numéricas que transforman de `String` a tipo básico.

Algunos ejemplos de esta compatibilidad son los siguientes:

```
// conversión implícita de double a Double
Double d1 = 1.5;
// el método parseFloat convierte su argumento a float
// después, conversión implícita de float a Float
Float f1 = Float.parseFloat("12345677890.012345");
// conversión implícita de Float a float
float f2 = f1;
// el método parseInt convierte la String "12345" a int
// después, conversión implícita de int a long
long l = Integer.parseInt("12345");
```

Nótese que, sin embargo, la instrucción `Double d2 = 1.5f;` no es correcta ya que no se puede convertir el `float 1.5f` a `Double` directamente sin pasar por el tipo `double`. Lo correcto sería: `double d2 = 1.5f;` `Double d3 = d2;`

El tipo `char` tiene su correspondiente clase envolvente `Character` con una funcionalidad aumentada, como es el uso de códigos especiales Unicode (como los denominados *subrogados* y *suplementarios*) proporcionando métodos, tanto estáticos como dinámicos, para manejarlos.

Los tipos `char` y `Character` también son compatibles y es el propio compilador el que realiza las conversiones automáticamente. Al igual que el resto de las clases envolventes, la clase `Character` también es inmutable.

Las siguientes son instrucciones de asignación válidas que demuestran la compatibilidad entre el tipo básico `char` y su clase envolvente `Character`:

```
Character ch1 = 'a';
char ch2 = new Character('a');
```

En la tabla 6.9 se muestran algunos de los métodos estáticos (basadas en la definición del estándar Unicode) definidos en la clase `Character`.

|                                                          |                                                   |
|----------------------------------------------------------|---------------------------------------------------|
| <code>public static boolean isDigit (char ch)</code>     | Determina si el carácter es un dígito.            |
| <code>public static boolean isLetter (char ch)</code>    | Determina si el carácter es una letra.            |
| <code>public static boolean isSpaceChar (char ch)</code> | Determina si el carácter es el espacio en blanco. |
| <code>public static boolean isUpperCase (char ch)</code> | Determina si el carácter está en mayúsculas.      |
| <code>public static char toLowerCase (char ch)</code>    | Convierte un carácter a minúsculas.               |
| <code>public static char toUpperCase (char ch)</code>    | Convierte un carácter a mayúsculas.               |
| <code>public static Character valueOf (char ch)</code>   | Convierte un carácter a Character.                |

**Tabla 6.9:** Algunos métodos de la clase Character.

Para finalizar señalar que, al igual que el resto de tipos elementales, también el tipo `boolean` tiene asociada una clase envolvente, `Boolean`, de funcionalidad extendida y con características generales similares a las del resto de clases envolventes.

## 6.4 Problemas propuestos

1. Hacer una traza del siguiente programa indicando en cada paso el contenido de cada una de las variables utilizadas. Si es necesario, consúltese el *API* de Java para la clase *String*.

```
public class TestString {
 public static void main(String[] args) {
 String s1 = "Hola", s2 = "", s3;
 System.out.println("Valor de s1: " + s1);
 s3 = s1;
 System.out.println("Valor de s3: " + s3);
 s2 = s1 + ", Adios!";
 s2 = s2 + " " + 58;
 System.out.println("Valor de s2: " + s2);
 System.out.println("Longitud de s2: " + s2.length());
 System.out.println("Carácter 10: " + s2.charAt(10));
 System.out.println(s1.contains("Hola"));
 System.out.println(s2.equals(s1));
 System.out.println("alfredo".compareTo("luisa"));
 }
}
```

2. Hacer una traza del siguiente segmento de código indicando en cada instrucción el contenido de las variables involucradas.

```
String st = " Una String con números: 632 ";
st = st.trim();
String st2 = st.concat(st);
int ini = st2.indexOf(st);
st2 = st.replace("632","2128");
st2 = st2.toLowerCase().replace("str","Str");
ini = st2.indexOf("21");
int valor = st2.indexOf(st2.substring(ini,ini+4));
```

3. Completar la siguiente tabla indicando si cada expresión es correcta y escribiendo su valor en caso de que lo sea o el motivo por el cuál no es correcta.

| Expresión                  | ¿Es Correcta? | Valor o Motivo |
|----------------------------|---------------|----------------|
| Character.isDigit('A')     |               |                |
| Character.isLetter(a)      |               |                |
| Character.toLowerCase('a') |               |                |
| Character.isDigit(1)       |               |                |
| Character.toLowerCase('B') |               |                |
| Character.isDigit('1')     |               |                |
| Character.toUpperCase('A') |               |                |
| toUpperCase('b')           |               |                |

4. Diseñar una clase de utilidades que permita calcular las funciones trigonométricas habituales (las definidas en `Math`) de forma que los ángulos estén representados en grados en lugar de radianes. Para realizar la transformación necesaria, se puede tener en cuenta que  $\pi$  radianes equivalen a 180 grados.
5. Dados dos valores  $x$  e  $y$ , reales, con  $x$  positivo, una forma de calcular la potencia  $x^y$  es utilizar la equivalencia:  $x^y = e^{y \log x}$  (donde  $e$ , es la base de los logaritmos neperianos y  $\log x$  es el logaritmo en dicha base de  $x$ ), escribir un método en Java para efectuar dicho cálculo (suponiendo que no existe el método predefinido `Math.pow(double, double)`).  
Comparar los resultados numéricos obtenidos con dicho método con los que se obtienen usando el método predefinido `Math.pow(double, double)`.
6. Se desea calcular la raíz cúbica de cierto número en coma flotante  $x$ . ¿Por qué no es correcta la expresión `Math.pow(x, 1/3)` para realizar dicho cálculo? ¿Qué se debe cambiar en la misma para que el cálculo se efectúe correctamente?
7. Dadas dos variables de tipo `int` `i` y `j`, se desea escribir, con una breve descripción previa, su suma en la salida estándar, para lo que se propone la instrucción siguiente:

```
System.out.println("La suma vale: " + i + j);
```

Sin embargo, cuando se hacen algunas pruebas, se observa que el resultado escrito no es correcto. ¿Qué es lo que ocurre?, ¿Cómo debe reescribirse la instrucción anterior para que el resultado que se muestre sea el correcto?

8. Diseñar una clase `PrintNum` cuyo objetivo sea facilitar la transformación de un valor numérico, como por ejemplo uno de tipo `double` o `int`, a una `String`, permitiendo que el usuario pueda establecer características propias de la representación. Como, por ejemplo, la amplitud del campo deseada, o el número de dígitos decimales que se representarán. Los métodos definidos deberán ser estáticos.

Un ejemplo de uso de la misma puede ser su utilización para la presentación de un valor `x` de tipo `double` con dos decimales en un campo de amplitud 10 posiciones:

```
// x es un número double
String numStr = PrintNum.double(x, 10, 2);
// y entonces, ...
System.out.println("El número es: " + numStr);
```

9. Se desea imitar el lanzamiento de una pareja de dados con caras numeradas del 1 al 6. Escribir una secuencia de instrucciones en Java que imiten el comportamiento deseado, escribiendo en la salida estándar, cada vez que se

ejecute, un valor al azar comprendido entre 1 y 12, correspondiente a la suma de los valores obtenidos en cada dado.

Debe tenerse en cuenta que cuando se lanzan dos dados, es más probable obtener unos valores que otros así, por ejemplo, sólo hay una combinación mediante la que se puede obtener 2 (en ambos dados se ha obtenido 1), mientras que hay 6 combinaciones distintas con las que se puede obtener 7 (son: (1,6),(6,1),(2,5),(5,2),(3,4),(4,3)).

10. Realizar un método en Java que escriba en la pantalla, cuando se ejecute, los intervalos de valores válidos para los distintos tipos numéricos enteros. Para realizarlo se deberá utilizar las constantes predefinidas en las clases envoltorio correspondientes.
11. ¿Cómo se escribe en Java una constante *double* que represente el infinito positivo? ¿Y el infinito negativo? ¿Se puede operar con ellas? ¿Qué ocurre si se dividen entre sí?
12. Los métodos *Integer.parseInt(String)* e *Integer.valueOf(String)* tienen un cometido similar, ya que mediante ambos es posible transformar un valor entero escrito como una *String* a un valor entero. Las características de ambos se pueden estudiar en el *API* del Java. ¿Hay alguna diferencia entre ellos? ¿Afecta la misma a la ejecución del segmento de código siguiente?

```
String valor = "123456654";
int i1 = Integer.parseInt(valor);
int i2 = Integer.valueOf(valor);
```

13. ¿Qué ocurre si se ejecuta alguno de los dos segmentos de código que aparecen a continuación?

```
String valor = " 123456654 ";
int i1 = Integer.parseInt(valor);

String valor2 = " 1234a6654 ";
int i2 = Integer.parseInt(valor2)
```

14. Una situación similar a la ocurrida el problema anterior se produce si se utiliza el método *Double.parseDouble(String)*. ¿Qué ocurre ahora si se ejecuta alguno de los dos segmentos de código que aparecen a continuación?

```
String valor = " 123456654 ";
double x1 = Double.parseDouble(valor);

String valor2 = " 1234a6654 ";
double x2 = Double.parseDouble(valor2);
```

¿Hay alguna diferencia entre el tratamiento de conversión para los valores *double* con respecto a los *int*?

## Más información

- [Ora11c] Oracle. *Java<sup>TM</sup> Platform, Standard Edition 6, API Specification*, 2011.  
URL: <http://download.oracle.com/javase/6/docs/api/>.
- [Ora11d] Oracle. *The Java<sup>TM</sup> Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Numbers and Strings.
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.  
Capítulo 1 (1.3) y Capítulo 5 (5.1 - *The Math Class* y *Wrapper Classes*).

## Capítulo 7

# Entrada y salida elemental

En Java la entrada y salida de datos y resultados desde un programa se realiza utilizando flujos (*streams*), que son secuencias de información (secuencias de bytes) que se reciben desde una fuente (i.e., teclado, fichero, red, etc.) y se dirigen a un destino (i.e., pantalla, fichero, impresora, etc.). Si el flujo de datos se dirige al programa, es decir, representa una entrada de datos, entonces se habla de *flujo de entrada* (al programa). Por el contrario, si el flujo de datos parte del programa, es decir, se trata de una salida de datos, entonces se habla de *flujo de salida* (del programa). En la figura 7.1 se representan gráficamente dichos flujos.

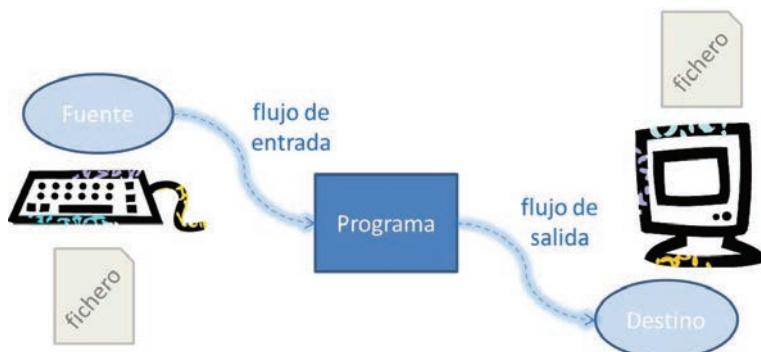


Figura 7.1: Representación de los flujos de entrada y salida.

Asociados a todo programa Java en ejecución existen dos flujos especiales, uno de entrada, o *entrada estándar*, y otro de salida, o *salida estándar*. Se trata de los objetos `System.in` y `System.out`<sup>1</sup> que se encuentran predefinidos y con los que

<sup>1</sup>Existe un tercer flujo predefinido `System.err`, similar a `System.out`, usado para mostrar mensajes de error.

siempre es posible interactuar en los programas. Lo habitual es que estos *flujos estándar* se encuentren asociados por defecto al teclado (el de entrada) y a la pantalla (el de salida).

En este capítulo se examina el uso elemental de los flujos estándar para poder introducir datos en los programas en ejecución así como para efectuar la salida de resultados desde los mismos. Debido a la asociación de los flujos con los dispositivos ya comentada, la entrada se realizará habitualmente desde el teclado mientras que la salida será hacia la pantalla o terminal del ordenador.

Cabe notar que aunque en capítulos anteriores ya se ha utilizado la salida estándar, ésta se presenta en este capítulo con más detalle. En particular, se muestra cómo darle formato. También se introduce la clase **Scanner** que simplifica la lectura de datos desde la entrada estándar.

Finalmente, señalar que en Java se pueden definir otros flujos para que un programa pueda leer desde o escribir en otros elementos, tales como ficheros, dispositivos multimedia u ordenadores remotos. Una vez definidos, se usan de manera similar a como se utilizan los flujos estándar. En el capítulo 16 se profundiza en este tema.

## 7.1 Salida por pantalla

En esta sección se revisa la instrucción `System.out.println` y se presentan las instrucciones que permiten dar formato a la salida estándar.

### 7.1.1 `System.out.println` y `System.out.print`

Se puede mostrar una línea por pantalla usando `System.out.println`. `System.out` es un objeto que forma parte del lenguaje Java y `println` es un método invocado sobre dicho objeto. Los datos a mostrar son los argumentos que van entre paréntesis y pueden ser cadenas de caracteres; sin embargo, gracias a una transformación automática que efectúa el lenguaje Java de cualquier tipo a cadena de caracteres, es posible utilizar como argumento de la instrucción una expresión de cualquier tipo. Por lo tanto, es posible escribir cadenas de caracteres, literales de las mismas entre comillas dobles, así como variables, números, expresiones o cualquier objeto que se pueda definir en Java. Para mostrar más de un dato, se utiliza el operador de concatenación de cadenas de caracteres `+`.

La sintaxis de la instrucción es:

```
System.out.println(Elem_1 + Elem_2 + ... + Elem_n);
```

siendo *Elem\_i* cada uno de los elementos a mostrar. Aunque también puede no tener argumentos:

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

En este caso, muestra una línea en blanco (es decir, escribe en la salida estándar un salto de línea).

Por ejemplo, las siguientes instrucciones:

```
double r = 5.5;
String c = "rojo";
System.out.println("Círculo de radio " + r + ", color " + c);
System.out.println();
System.out.println(" y centro (" + 6 + "," + 3 + ").");
```

muestran por pantalla:

| Salida Estándar                                     |
|-----------------------------------------------------|
| Círculo de radio 5.5, color rojo<br>y centro (6,3). |

La única diferencia entre `System.out.println` y `System.out.print` es que con `println` se escribe un cambio de línea, con lo que la siguiente salida a mostrar se muestra en una línea nueva, mientras que con `print`, la siguiente salida se muestra en la misma línea. Por ejemplo, las instrucciones:

```
double r = 5.5;
String c = "rojo";
System.out.print("Círculo de radio " + r + ", color " + c);
System.out.println(" y centro (" + 6 + "," + 3 + ").");
```

muestran por pantalla:

| Salida Estándar                                  |
|--------------------------------------------------|
| Círculo de radio 5.5, color rojo y centro (6,3). |

Cabe señalar que la instrucción `System.out.println(expresión)`; es equivalente a `System.out.print(expresión + "\n")`;

### 7.1.2 Salida formateada con printf

Desde la versión 5.0, Java incluye un método llamado `printf` que puede usarse para presentar la salida en un formato específico. La instrucción `System.out.printf` permite escribir, dándoles formato, cualquier número de argumentos y utiliza uno o dos argumentos adicionales iniciales para determinar cómo se efectuará la escritura de los argumentos restantes.

Normalmente, el primer argumento de la instrucción es un *String con formato* para el resto, una *lista de argumentos a formatear*. Todos los argumentos subsiguientes son valores que se mostrarán por pantalla en el formato especificado por el *String con formato*. El *String con formato* puede contener texto, así como *especificadores de formato*, y dicho texto se mostrará junto con los valores de la lista de argumentos a formatear.

```
System.out.printf(String con formato, Lista de argumentos a formatear);
```

Por ejemplo, se puede mostrar por pantalla el valor de `Math.PI` con tres dígitos decimales utilizando:

```
System.out.printf("El valor de Math.PI es %.3f \n", Math.PI);
```

que mostrará:

|                              |                 |  |
|------------------------------|-----------------|--|
|                              | Salida Estándar |  |
| El valor de Math.PI es 3,142 |                 |  |

Mientras que utilizando:

```
System.out.println("El valor de Math.PI es " + Math.PI);
```

se obtendrá:

|                                          |                 |  |
|------------------------------------------|-----------------|--|
|                                          | Salida Estándar |  |
| El valor de Math.PI es 3.141592653589793 |                 |  |

Los especificadores de formato indican cómo se van a formatear los valores de la lista de argumentos a formatear. Un especificador de formato comienza con el carácter % y finaliza con un carácter que indica el tipo de conversión a realizar, llamado *indicador de conversión*. Además, puede contener otros elementos que controlan el diseño del valor a convertir.

%[índice \_ argumento\$] [flags] [anchura] [.precisión] *indicador de conversión*

Los elementos entre corchetes son opcionales (pueden aparecer o no) y dependen del indicador de conversión.

- *índice\_argumento* es un entero que indica la posición del argumento en la lista de argumentos a formatear.
- *flags* permiten, entre otras cosas, la justificación a la izquierda o a la derecha.
- *anchura* indica el número mínimo de caracteres que aparecerán en la salida (útil para alinear columnas de datos).
- *precisión* indica, en la conversión de valores reales, el número de cifras decimales que deben aparecer.

El indicador de conversión es un carácter que especifica el tipo del valor que se va a mostrar y su formato.

- Para valores de tipo `byte`, `short`, `int` y `long`:
  - `d`: formato decimal.
  - `o`: formato octal.
  - `x`, `X`: formato hexadecimal.
- Para valores de tipo `float` y `double`:
  - `e`, `E`: notación científica informatizada (p.e., `3.142e+00`).
  - `f`, `F`: notación decimal (p.e., `3,142`).
  - `g`, `G`: notación científica general.
- Para valores de tipo `String`: `s`, `S`.

A continuación, se detalla el uso de la instrucción `System.out.printf` mediante una serie de ejemplos. Para cada uno, con el fin de distinguir con facilidad los espacios en blanco entre caracteres, se representa una línea de la pantalla como una línea de una cuadrícula.

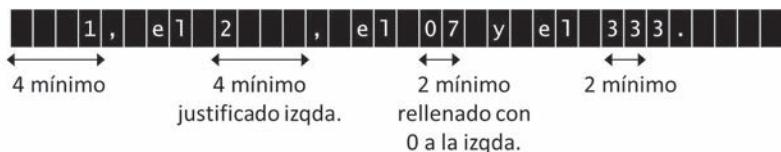
**Ejemplo 7.1.** Con la instrucción que sigue se pretende mostrar por pantalla los enteros 1, 2, 7 y 333 en distintos formatos:

```
System.out.printf("%4d, el %-4d, el %02d y el %2d.\n",1,2,7,333);
```

Los especificadores de formato `%4d` y `%-4d` indican que para mostrar un valor entero se usarán un mínimo de 4 espacios en la salida, justificando a la derecha o a la izquierda (con el flag `-`), respectivamente. El `%2d` hace que un valor entero se

muestre en 2 espacios como mínimo. El flag 0 del especificador %02d indica que se completan con ceros los espacios no utilizados del total de reservados. El \n al final del String con formato implica que el printf se comporta como un println.

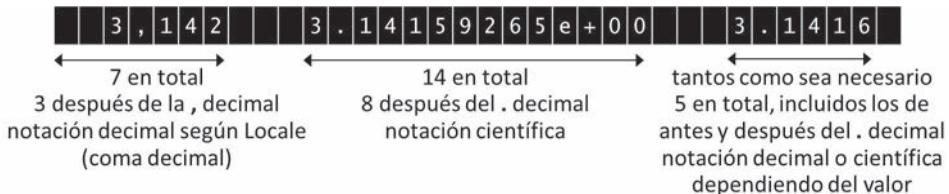
El resultado es:



**Ejemplo 7.2.** La línea de código:

```
System.out.printf("%7.3f %1$14.8e %1$1.5g", Math.PI);
```

muestra por pantalla el valor real Math.PI en distinta notación:



El especificador de formato %7.3f indica que un valor real se mostrará en notación decimal (f) en un total de 7 espacios, de los cuales 3 se reservan para la parte decimal. Nótese que el valor de Math.PI se ha redondeado a 3 decimales y como separador se utiliza la coma decimal. El especificador %1\$14.8e indica que el primer argumento (1\$) de la lista de argumentos a formatear es un valor real que se mostrará en notación científica (e) en un total de 14 espacios, de los cuales 8 son para la parte decimal. Por último, el especificador %1\$1.5g indica que para mostrar Math.PI se utilizarán 5 espacios en total, incluidos los de antes y después del punto decimal. El 1 (del 1.5) indica el número (mínimo) de caracteres a mostrar, es decir, tantos como sean necesarios. Poner un 1 o no poner nada (en el ejemplo, %1\$.5g) tiene el mismo efecto. El indicador de conversión g dependiendo del valor real a formatear, lo hace en notación decimal o en notación científica.

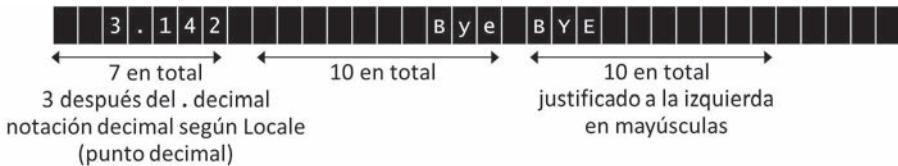
**Ejemplo 7.3.** Con la siguiente instrucción se quiere mostrar por pantalla el valor de `Math.PI` y la cadena de caracteres `Bye` en distintos formatos:

```
System.out.printf(Locale.US,"%7.3f %10s %2$-10S",Math.PI,"Bye");
```

En este caso, el primer argumento de la instrucción, anterior incluso al de especificación de formato, es `Locale.US` una constante predefinida que cambia la ubicación predeterminada (en nuestro caso, la de España) a la de Estados Unidos. Las constantes de ubicación forman parte de la clase `Locale`. Para poder usarlas, hay que incluir la instrucción `import java.util.Locale;` como primera línea del programa.

El especificador `%7.3f` mostrará el valor real `Math.PI` en notación decimal (`f`) en un total de 7 espacios, con 3 decimales separados de la parte real por el punto decimal (según `Locale`). El especificador de formato `%10s` mostrará el `String` `Bye`, justificado a la derecha, en un total de 10 espacios. Con el flag `-` se cambia la justificación. Así, el especificador `%2$-10S` justifica a la izquierda el segundo argumento (`2$`) de la lista de argumentos a formatear y lo convierte a mayúsculas (`S`).

El resultado es:



## 7.2 Entrada desde teclado

El objeto `System.in` permite leer datos introducidos por el usuario desde teclado pero su uso no es tan sencillo como el del objeto `System.out`. Desde la versión 5.0, Java incluye una clase para realizar de forma sencilla la entrada de datos al programa. En esta sección, se presenta cómo gestionar la entrada desde teclado usando dicha clase, llamada `Scanner`. Aunque en el capítulo 16, se verá el uso de la clase `Scanner` para la entrada de datos desde fichero, aquí se introduce de forma básica dicho uso.

### 7.2.1 La clase Scanner

La clase `Scanner` forma parte del paquete `java.util` y permite leer valores introducidos por el teclado de una forma cómoda para el programador. Esta clase

ofrece un grupo de métodos que bloquean el flujo de control del programa hasta que el usuario introduce una entrada por teclado. Además, permiten gestionar la entrada de teclado como si fuera un *buffer* de datos en el que se almacena lo que va tecleando el usuario para ser posteriormente leído.

```

1 /**
2 * Clase TestScanner: ejemplo de uso de la clase Scanner.
3 * @author Libro IIP-PRG
4 * @version 2011
5 */
6 import java.util.Scanner;
7 public class TestScanner {
8 public static void main(String[] args) {
9 String nombre;
10 int a_1, a_2;
11 Scanner teclado = new Scanner(System.in);
12 System.out.println("Introduce tu nombre");
13 nombre = teclado.nextLine();
14 System.out.println("Introduce tu año de nacimiento y el actual");
15 a_1 = teclado.nextInt();
16 a_2 = teclado.nextInt();
17 System.out.print("Te llamas " + nombre);
18 System.out.println(" y tienes " + (a_2 - a_1) + " años");
19 }
20 }
```

**Figura 7.2:** Ejemplo de uso de la clase **Scanner**.

La figura 7.2 muestra un ejemplo de utilización de la clase **Scanner** para la lectura de valores por teclado. En primer lugar, se importa la clase **Scanner** del paquete correspondiente (línea 6). A continuación se crea un nuevo objeto de tipo **Scanner** pasándole como argumento el flujo de entrada estándar, **System.in** (línea 11). Dado que, por defecto, la entrada estándar del programa va ligada al teclado, esto permitirá leer los valores solicitados al usuario que éste introduzca a través de teclado. Luego, se pide al usuario que introduzca su nombre (línea 12). Es muy importante mostrar un mensaje al usuario antes de realizar cualquier operación de lectura de datos para que el usuario conozca qué tipo de información debe suministrar al programa.

Mediante el método **nextLine()** se consigue leer de teclado una línea completa, es decir, todo lo que el usuario haya tecleado hasta que pulsó la tecla *Enter*. Después, se solicita al usuario que introduzca tanto su año de nacimiento como el año actual (línea 14). Ambos valores deben ser introducidos separados por, al menos, un espacio (un tabulador o un salto de línea) y son leídos mediante dos

invocaciones consecutivas al método `nextInt()`. Finalmente, el programa muestra por pantalla tanto el nombre del usuario como su edad (líneas 17 y 18).

| Entrada/Salida Estándar                    |              |
|--------------------------------------------|--------------|
| Introduce tu nombre                        | Luisa García |
| Introduce tu año de nacimiento y el actual | 1982 2011    |
| Te llamas Luisa García y tienes 29 años    |              |

La tabla 7.1 resume algunos de los principales constructores y métodos de la clase `Scanner`. Aunque se verá con detalle en el capítulo 15, se observa que las operaciones de lectura sobre `Scanner` pueden lanzar la excepción `InputMismatchException` en caso de tratar de leer un dato que no corresponda al tipo solicitado. En el caso de los métodos `next()` y `nextLine()` para leer, respectivamente, bien una palabra bien la secuencia de caracteres hasta el final de la línea, dicha excepción no se lanza puesto que cualquier entrada por teclado que realice el usuario siempre puede considerarse de tipo `String`. Se aconseja visitar el *API* de Java [Ora11c] para conocer con detalle algunos métodos adicionales.

El funcionamiento de los métodos de la clase `Scanner` puede llevar a confusión al programador e introducir un error que resulta ser bastante común y que provoca una lectura inapropiada de los datos. Para comprender el problema, es importante saber que los métodos de lectura de tipos primitivos a través de `Scanner` descartan el salto de línea `\n` que se introduce cuando el usuario pulsa la tecla *Enter*. Además, el método `nextLine()` se encarga de leer el resto de línea desde el punto de lectura de entrada de teclado donde quedó el `Scanner` tras la última operación hasta el salto de línea (excluido).

La figura 7.3 incluye un programa que plantea este problema (nótese la línea comentada). Tras solicitar al usuario un entero (línea 11), éste habrá tecleado un valor entero y habrá pulsado la tecla *Enter*. Esto provoca que en el buffer de lectura de teclado esté el valor entero y el salto de línea. La operación de la línea 12 permite leer el entero, dejando el salto de línea en el buffer de lectura. Por ello, al invocar la operación de lectura de línea (línea 15), ésta procede a leer el resto de línea pendiente (una cadena vacía) y al encontrar el salto de línea detiene la lectura y lo descarta. Por ello, la operación de la línea 15 no se detiene a la espera de que el usuario introduzca un valor.

|                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>public Scanner (InputStream source)</code>                                                                                                                                                                                                                                                                                     | Crea un nuevo <code>Scanner</code> a partir de un flujo de entrada de datos como es el caso de <code>System.in</code> (para poder leer desde teclado).                                                                                                                                                                                                                                                                           |
| <code>public String next ()</code><br><code>public String next (String pattern)</code>                                                                                                                                                                                                                                               | Obtiene el siguiente elemento leído del teclado como un <code>String</code> (si coincide con el <i>patrón</i> especificado). Lanza <code>NoSuchElementException</code> si no quedan más elementos por leer.                                                                                                                                                                                                                      |
| <code>public String nextLine ()</code>                                                                                                                                                                                                                                                                                               | Se lee el resto de línea completa, descartando el salto de línea. Devuelve el resultado como un <code>String</code> . Lanza <code>NoSuchElementException</code> si no quedan más elementos por leer.                                                                                                                                                                                                                             |
| <code>public int nextInt ()</code><br><code>public long nextLong ()</code><br><code>public short nextShort ()</code><br><code>public byte nextByte ()</code><br><code>public float nextFloat ()</code><br><code>public double nextDouble ()</code><br><code>public boolean nextBoolean ()</code>                                     | Devuelve el siguiente elemento como un <code>int</code> siempre que se trate de un <code>int</code> . Ídem para <code>long</code> , <code>short</code> , <code>byte</code> , <code>float</code> , <code>double</code> y <code>boolean</code> . Lanza <code>InputMismatchException</code> en caso de no poder obtener un valor del tipo apropiado. Lanza <code>NoSuchElementException</code> si no quedan más elementos por leer. |
| <code>public boolean hasNext ()</code>                                                                                                                                                                                                                                                                                               | Devuelve <code>true</code> si queda algún elemento por leer.                                                                                                                                                                                                                                                                                                                                                                     |
| <code>public boolean hasNextLine ()</code>                                                                                                                                                                                                                                                                                           | Devuelve <code>true</code> si queda alguna línea por leer.                                                                                                                                                                                                                                                                                                                                                                       |
| <code>public boolean hasNextInt ()</code><br><code>public boolean hasNextLong ()</code><br><code>public boolean hasNextShort ()</code><br><code>public boolean hasNextByte ()</code><br><code>public boolean hasNextFloat ()</code><br><code>public boolean hasNextDouble ()</code><br><code>public boolean hasNextBoolean ()</code> | Devuelve <code>true</code> si el siguiente elemento a obtener se puede interpretar como un <code>int</code> . Ídem para <code>long</code> , <code>short</code> , <code>byte</code> , <code>float</code> , <code>double</code> y <code>boolean</code> .                                                                                                                                                                           |
| <code>public Scanner useLocale (Locale l)</code>                                                                                                                                                                                                                                                                                     | Establece la configuración local del <code>Scanner</code> a la configuración especificada por el <code>Locale l</code> .                                                                                                                                                                                                                                                                                                         |

Tabla 7.1: Principales constructores y métodos de la clase `Scanner`.

```

1 /**
2 * Clase TestScannerLinea: ejemplo de posible problema con
3 * la clase Scanner.
4 * @author Libro IIP-PRG
5 * @version 2011
6 */
7 import java.util.*;
8 public class TestScannerLinea {
9 public static void main(String[] args) {
10 Scanner teclado = new Scanner(System.in);
11 System.out.print("Introduce el entero: ");
12 int n = teclado.nextInt();
13 //teclado.nextLine();
14 System.out.print("Introduce una línea: ");
15 String s1 = teclado.nextLine();
16 System.out.print("Introduce la otra línea: ");
17 String s2 = teclado.nextLine();
18 System.out.println("\nEnteró: " + n);
19 System.out.println("Línea 1: " + s1);
20 System.out.println("Línea 2: " + s2);
21 }
22 }
```

**Figura 7.3:** Ejemplo de posible problema con la clase Scanner.

Un ejemplo de ejecución del programa se muestra a continuación.

| Entrada/Salida Estándar                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------|
| Introduce el entero: 5<br>Introduce una línea: Introduce la otra línea: hola<br><br>Entero: 5<br>Línea 1:<br>Línea 2: hola |

En resumen, este problema ocurre si una operación de lectura de un tipo primitivo va seguida de una operación de lectura de línea completa. La solución pasa por descartar el salto de línea tras la operación de lectura del tipo primitivo. En el código, sería necesario descomentar la línea 13.

La figura 7.4 muestra un programa en el que se pide al usuario que introduzca un valor real desde teclado. Si no se indica lo contrario, Scanner usa el Locale por defecto al realizar la lectura desde teclado. Con la instrucción `Locale.getDefault()` (línea 14) podemos saber cuál es el Locale instalado por defecto. En nuestro caso

es el de España. Por ello, la llamada `teclado.nextDouble()` (línea 17) espera un valor real con la coma como separador de decimales. Con la instrucción `teclado.useLocale(Locale.US);` (línea 20) cambiamos al Locale de Estados Unidos y, ahora, el método `nextDouble()` (línea 22) espera un valor real con el punto como separador de decimales. Nótese que la instrucción `System.out.println` (líneas 18 y 23) muestra el valor real con el punto decimal.

```

1 /**
2 * Clase TestScannerLocale: ejemplo de uso de la clase Locale
3 * con la clase Scanner.
4 * @author Libro IIP-PRG
5 * @version 2011
6 */
7 import java.util.Scanner;
8 import java.util.Locale;
9 public class TestScannerLocale {
10 public static void main(String[] args) {
11 Scanner teclado = new Scanner(System.in);
12
13 System.out.print("El teclado está configurado por defecto en ");
14 System.out.println(Locale.getDefault());
15
16 System.out.print("Escribe un número real (con coma decimal): ");
17 double nReal1 = teclado.nextDouble();
18 System.out.println("El valor real leído es " + nReal1);
19
20 teclado.useLocale(Locale.US);
21 System.out.print("Escribe un número real (con punto decimal): ");
22 double nReal2 = teclado.nextDouble();
23 System.out.println("El valor real leído es " + nReal2);
24 }
25 }
```

**Figura 7.4:** Ejemplo de uso de la clase `Locale` con la clase `Scanner`.

A continuación se muestra una ejecución del programa.

| Entrada/Salida Estándar                                                                                                                                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| El teclado está configurado por defecto en es_ES<br>Escribe un número real (con coma decimal): 6,75<br>El valor real leído es 6.75<br>Escribe un número real (con punto decimal): 8.5<br>El valor real leído es 8.5 |

La lectura desde teclado de un valor de tipo `char` se puede hacer tal como se indica en la línea 11 del programa de la figura 7.5. Al método `next` se le pasa como argumento "`\S`", lo que se conoce como un *patrón*, un `String` representando una *expresión regular*. El patrón "`\S`" indica que `next` está esperando un `String` formado por un único carácter distinto de un separador (es decir, no puede ser ni un espacio en blanco, ni un salto de línea, ni un tabulador). Con el método `charAt(0)`, el carácter que ocupa la posición 0 del `String` se convierte en el valor de tipo `char` correspondiente.

```

1 /**
2 * Clase TestScannerChar: ejemplo de lectura de un valor de
3 * tipo char con la clase Scanner.
4 * @author Libro IIP-PRG
5 * @version 2011
6 */
7 import java.util.*;
8 public class TestScannerChar {
9 public static void main(String[] args) {
10 Scanner teclado = new Scanner(System.in);
11 System.out.print("Introduce un carácter: ");
12 char c = teclado.next("\S").charAt(0);
13 System.out.println("\nEl carácter leído es: " + c);
14 }
15 }
```

**Figura 7.5:** Ejemplo de lectura de un dato de tipo `char` con la clase `Scanner`.

El siguiente es un ejemplo de ejecución del programa.

|                          |
|--------------------------|
| Entrada/Salida Estándar  |
| Introduce un carácter: k |
| El carácter leído es: k  |

### *Uso de las clases Scanner y File para la entrada de datos desde fichero*

La entrada de datos desde un fichero de texto se realiza de la misma manera que la entrada desde el teclado, creando un objeto de la clase `Scanner` y reemplazando el argumento `System.in` por el flujo asociado al fichero de texto, construyendo un objeto `File` a partir del nombre del fichero a leer.

```
Scanner fich = new Scanner(new File(nombreFichero));
```

Además, al formar parte la clase `File` del paquete `java.io`, es necesario importar dicha clase:

```
import java.io.File;
```

Una vez creado el objeto `Scanner`, se puede invocar a cualquiera de los métodos de lectura de la clase `Scanner` para leer los datos del fichero.

```
/**
 * Clase TestScannerFichero: ejemplo de uso de la clase Scanner
 * para lectura de datos desde fichero.
 * @author Libro IIP-PRG
 * @version 2011
 */
import java.util.Scanner;
import java.io.File;
public class TestScannerFichero {
 public static void main(String[] args) throws Exception {
 Scanner teclado = new Scanner(new File("datos.txt"));
 String nombre = teclado.nextLine();
 int a_1 = teclado.nextInt();
 int a_2 = teclado.nextInt();
 System.out.print("Te llamas " + nombre);
 System.out.println(" y tienes " + (a_2 - a_1) + " años");
 }
}
```

**Figura 7.6:** Ejemplo de uso de la clase `Scanner` para lectura desde fichero.

La figura 7.6 muestra un ejemplo de utilización de la clase `Scanner` para la lectura de datos desde fichero. En concreto, se realiza la lectura de un fichero de nombre `datos.txt` que contiene, al menos, tres datos: un `String` en la primera línea y dos `int` en la siguiente línea, representando el nombre de una persona, su año de nacimiento y el año actual, respectivamente. La invocación del constructor de la clase `Scanner` con un objeto `File` como argumento puede lanzar la excepción `FileNotFoundException` si el fichero especificado no existe. Por ello, en este ejemplo, para manejar dicha excepción se añade en la cabecera del método `main` la cláusula `throws Exception` (véase el capítulo 15).

En el capítulo 16 se detalla la entrada de datos desde fichero y se pueden encontrar ejemplos de uso comentados.

### 7.3 Problemas propuestos

1. Escribir un programa en Java que se encargue de leer una serie de datos de la entrada estándar, cuya descripción se muestra a continuación, y los muestre por la salida estándar en el formato que se detalla.

- Los datos de entrada consistirán en los siguientes:
  - Fecha de nacimiento: tres numeros enteros.
  - NIF.
  - Nombre.
  - Dirección: calle y numero.
  - Código postal y población.
  - Teléfono fijo.
  - Teléfono móvil.
  - Salario bruto: un número real con dos decimales.
  - Retención: un número entero.

El programa los pedirá al usuario de esta forma:

| Entrada/Salida Estándar              |                               |
|--------------------------------------|-------------------------------|
| Fecha de nacimiento: dia, mes y año? | 9 10 1975                     |
| NIF?                                 | 99999999R                     |
| Nombre?                              | Rodolfo Santiesteban Amorales |
| Dirección: calle y numero?           | Avda. Mayor, 115              |
| Código postal y población?           | 67098 Villalba de la Hoz      |
| Teléfono fijo?                       | 111222333                     |
| Teléfono móvil?                      | 777555333                     |
| Salario bruto?                       | 1987.35                       |
| Retención?                           | 16                            |

- Los datos de salida se mostrarán en el formato que sigue:

| Salida Estándar      |                                                |
|----------------------|------------------------------------------------|
| Nombre.....          | : Rodolfo Santiesteban Amorales                |
| Dirección.....       | : Avda. Mayor, 115<br>67098 Villalba de la Hoz |
| Teléfono fijo.....   | : 111222333                                    |
| Teléfono móvil.....  | : 777555333                                    |
| NIF.....             | : 99999999R                                    |
| Fecha de nacimiento: | 09/10/1975                                     |
| Salario bruto.....   | : 1987.35                                      |
| Retención.....       | : 16%                                          |
| Salario neto.....    | : 1669.37                                      |

Es posible que los datos no siempre estén separados unos de otros por el mismo número de blancos. También puede haber blancos después del último dato en cada línea. Es por ello que se aconseja el uso de la función `trim()` de la clase `String`.

Para mostrar los datos de salida es suficiente la función `println()`, excepto para mostrar la fecha con dos dígitos cuando el día o el mes son de una cifra. Para este caso se debe de utilizar `printf()`.

El salario neto se calculará a partir del salario bruto y la retención como  $\text{salario bruto} - (\text{salario bruto} * \text{retención})/100$  y se mostrará con dos cifras decimales usando el punto como separador decimal.

- Escribir un programa que lea de la entrada estándar una cantidad en pesetas y muestre por la salida estándar su equivalente en euros, sabiendo que  $1\text{€} = 166.386$  ptas. La entrada consiste en un valor en pesetas y la salida es la conversión en euros con dos decimales, usando la coma como separador decimal.

| Entrada/Salida Estándar    |
|----------------------------|
| Cantidad en pesetas? 1000  |
| Equivalente en euros: 6,01 |

- La conversión de grados *Fahrenheit* a *Celsius* se realiza mediante la siguiente fórmula:  $\text{Celsius} = 5/9 * (\text{Fahrenheit} - 32)$ . Su inversa permite convertir de *Celsius* a *Fahrenheit*.

Escribir un programa que lea de la entrada estándar una temperatura, expresada en grados *Fahrenheit*, y la pase a grados *Celsius*. Y otra expresada en grados *Celsius* y la pase a *Fahrenheit*. Se usará la coma como separador decimal tanto para la entrada como para la salida. Además, la salida se mostrará con dos cifras decimales.

| Entrada/Salida Estándar                                                                                            |
|--------------------------------------------------------------------------------------------------------------------|
| Grados Fahrenheit? 32,5                                                                                            |
| Grados Celsius? 0,5                                                                                                |
| 32,5 grados Fahrenheit equivalen a 0,28 grados Celsius.<br>0,5 grados Celsius equivalen a 32,90 grados Fahrenheit. |

- Resolver los problemas relacionados con el diseño de programas propuestos en capítulos anteriores, de forma que los datos se lean desde la entrada estándar.

## Más información

[Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.

URL: <http://math.hws.edu/javanotes/>. Capítulo 2 (2.4.4) y Capítulo 11 (11.1.5).

[Ora11c] Oracle. *Java<sup>TM</sup> Platform, Standard Edition 6, API Specification*, 2011.

URL: <http://download.oracle.com/javase/6/docs/api/>.

[Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.

Capítulo 2.

# Capítulo 8

## Estructuras de control: selección

En la mayoría de los lenguajes de programación, existen distintas estructuras, llamadas *estructuras de control*, que controlan el *flujo de ejecución de un programa* (esto es, el orden que siguen las instrucciones durante la ejecución del mismo). En la programación estructurada, en particular, sólo se consideran tres estructuras de control: *secuencia*, *selección* y *repeticIÓN* (o *iteración*). En una estructura secuencial, las instrucciones se ejecutan una tras otra, en el mismo orden en el que aparecen en el programa. Una estructura de selección se utiliza para elegir entre dos o más alternativas de ejecución, dependiendo de una condición dada. En una estructura de repetición, la ejecución de una secuencia de instrucciones se repite una y otra vez mientras se cumpla una determinada condición.

Java dispone de las tres estructuras de control mencionadas: secuencia (*bloque*), instrucciones de selección (*if* y *switch*) e instrucciones de repetición (*while*, *do...while* y *for*). Cada una de estas estructuras se considera como una única instrucción, sin embargo, cada una es una instrucción estructurada que puede contener una o más instrucciones dentro de sí misma. En este capítulo se presentan en detalle las instrucciones de selección o condicionales. Las instrucciones de repetición se presentan en el capítulo 9.

### 8.1 Instrucciones condicionales

Una característica de los programas que se han visto hasta el momento es que consisten tan solo en una secuencia de instrucciones que se ejecuta inalterablemente. Sin embargo, en la resolución de un problema es a menudo necesario tomar decisiones en función de las características del mismo, esto es, se necesita poder alterar la secuencia de cálculos que se efectúan en función de los datos de entra-

da, o de resultados obtenidos con antelación. Las instrucciones condicionales, que se estudiarán seguidamente, permiten construir programas que tomen decisiones acerca de los cálculos a efectuar en función de las características del problema.

Supóngase, por ejemplo, que se desea determinar si cierta variable **temperatura** tiene un valor menor o mayor que 10 (escribiendo un mensaje según lo sea o no). Esto podría expresarse mediante las instrucciones:

```
if (temperatura < 10)
 System.out.println("Hace frío.");
else System.out.println("Se está bien o hace calor.");
...
```

Para ejecutar esta instrucción se determina, en primer lugar, la verdad o falsedad de la condición. Si la variable **temperatura** tiene un valor menor que 10 (la condición es cierta), entonces se escribirá el primer mensaje; si no es así, esto es si la variable tiene un valor mayor o igual que 10 (la condición es falsa) se escribirá el segundo mensaje. Una vez escrito el mensaje correspondiente, la ejecución del programa continuará en la instrucción siguiente (que en el ejemplo aparece indicada mediante puntos suspensivos).

Tanto después de la condición, como después de la palabra **else**, es posible escribir cualquier instrucción, por ejemplo otra instrucción condicional. Así, en el ejemplo siguiente se clasifica más aun el valor de la variable **temperatura**:

```
if (temperatura < 10)
 System.out.println("Hace frío.");
else if (temperatura < 26)
 System.out.println("Se está bien.");
else
 System.out.println("Hace calor.");
```

A continuación se examina la forma general que tienen las distintas instrucciones condicionales. Se utilizará la letra **B** para denotar una condición y la **S** para denotar una instrucción o un bloque cualquiera de instrucciones. Si existe más de una, de alguna de ellas, se hará uso de índices.

En la figura 8.1 se representa el flujo de ejecución de cada instrucción condicional mediante un diagrama de flujo.

### 8.1.1 Instrucción **if...else...**

La instrucción **if...else...** se basa en el valor de una expresión de tipo **boolean** (es decir, una expresión que se evalúa a **true** o a **false**) para elegir entre dos alternativas.

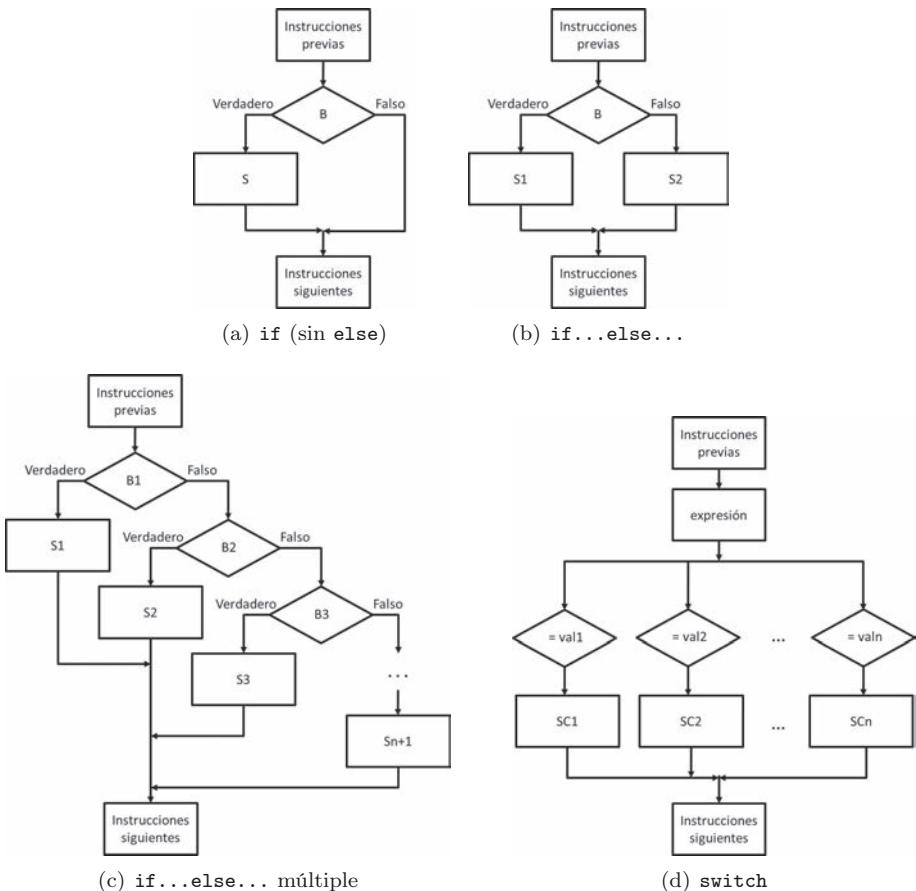


Figura 8.1: Diagramas de flujo de las instrucciones condicionales.

La forma más simple de una instrucción condicional es la siguiente:

```
if (B) S
```

donde **S** es una instrucción o un bloque cualquiera de instrucciones. Su ejecución se efectúa evaluando la condición **B**, de forma que si es cierta se ejecuta **S**. Al acabar la ejecución de la instrucción (o bloque), ésta continúa en la instrucción siguiente a la condicional. El diagrama de flujo de esta instrucción está reflejado en la figura 8.1(a).

La otra forma de instrucción condicional es:

```
if (B) S1 else S2
```

donde **S1** y **S2** son instrucciones simples (o bloques) cualesquiera. Su ejecución se efectúa evaluando la condición **B**, de forma que si ésta es cierta se ejecuta **S1**, por el contrario, si no es cierta, se ejecuta **S2**. Al acabar la instrucción condicional la ejecución continúa en la instrucción siguiente. Su diagrama de flujo es el de la figura 8.1(b).

Nótese que cuando en una instrucción **if...else...**, una o ambas alternativas contienen varias instrucciones, dichas instrucciones forman un bloque y deben escribirse entre llaves, como en el siguiente ejemplo que, dados dos valores enteros en **a** y **b**, asigna el máximo a **max** y en el mínimo a **min**, respectivamente:

```
int a, b, max, min;
...
if (a>b) {
 max = a;
 min = b;
}
else {
 max = b;
 min = a;
}
```

### *Instrucción condicional anidada*

En cualquiera de las modalidades de instrucción condicional **if...else...**, las instrucciones **S1** y **S2** pueden ser a su vez instrucciones condicionales. Es lo que se conoce como instrucción condicional *anidada*. Véase el siguiente ejemplo.

**Ejemplo 8.1.** Considérese el siguiente problema: Dado **p** un objeto **Punto** que representa a un punto  $(x, y)$ , se desea saber dónde está situado en el plano cartesiano.

La solución del problema depende de los valores  $x$  e  $y$ . En función de dichos valores, el punto puede estar situado en: el origen de coordenadas ( $x = 0$  e  $y = 0$ ), el eje de ordenadas ( $x = 0$  e  $y \neq 0$ ), el eje de abscisas ( $x \neq 0$  e  $y = 0$ ) o cualquiera de los cuatro cuadrantes ( $x \neq 0$  e  $y \neq 0$ ).

Al realizar un análisis por casos del problema, se pueden agrupar los casos en los que  $x = 0$  y los casos en los que  $x \neq 0$ , y para cada uno de ellos distinguir si  $y = 0$  o  $y \neq 0$ . Es decir,

- si  $x = 0$ , entonces,
  - si  $y = 0$ , el punto está en el origen de coordenadas  $(0, 0)$ ,
  - si  $y \neq 0$ , el punto está en el eje de ordenadas  $Y$ .
  
- si  $x \neq 0$ , entonces,
  - si  $y = 0$ , el punto está en el eje de abscisas  $X$ ,
  - si  $y \neq 0$ , el punto está en cualquiera de los 4 cuadrantes.

Es inmediato traducir este análisis por casos a una instrucción condicional en Java, en particular, a una instrucción condicional anidada como la que sigue:

```
double x = p.getX(), y = p.getY();
...
System.out.print("El punto está en ");
if (x==0)
 if (y==0)
 System.out.println("el origen de coordenadas (0,0)");
 else System.out.println("el eje de ordenadas Y");
else
 if (y==0)
 System.out.println("el eje de abscisas X");
 else System.out.println("cualquiera de los 4 cuadrantes");
```

En una instrucción condicional anidada surge un problema, sin embargo, si la instrucción S1 es una instrucción `if` que no tiene `else`. Recuérdese que la manera en que se sangra el código, no significa nada en absoluto para el compilador. En el siguiente ejemplo,

```
if (x>0)
 if (y>0)
 System.out.println("primer cuadrante");
 else System.out.println("otro");
```

se podría pensar que la parte del `else` se corresponde con el `if (x>0)`. En realidad, el compilador asocia el `else` al `if` más cercano que no tiene `else`, en este caso al `if (y>0)`. Es decir, el compilador entiende la instrucción como si estuviera escrita así:

```
if (x>0)
 if (y>0)
 System.out.println("primer cuadrante");
 else System.out.println("otro");
```

Para que el compilador realice la otra interpretación hay que incluir el **if** anidado en un bloque:

```
if (x>0){
 if (y>0)
 System.out.println("primer cuadrante");
 }
 else System.out.println("otro");
```

Cada instrucción **if** tiene un significado distinto: cuando **x<=0**, la primera instrucción no muestra nada por pantalla, mientras que la segunda instrucción muestra **otro**.

La colocación de un signo de punto y coma después de la condición de una instrucción **if** constituye un error en la lógica de un programa o un error de sintaxis. Por ejemplo, al ejecutarse el siguiente código, sea cual sea el valor de **x** siempre se muestra por pantalla **Valor positivo**, ya que el ; después de la condición se entiende como la única instrucción a ejecutar cuando la condición se evalúa a **true**.

```
if (x>0) ;
System.out.println("Valor positivo");
```

Mientras que el código siguiente provoca un error de compilación debido al error de sintaxis inducido por el ; después de la condición. El compilador entiende que la instrucción **if** acaba en dicho ; y la siguiente instrucción es la de escritura en pantalla **System.out.println("Valor positivo");**, mostrando el error '**else**' without '**if**' ya que encuentra un **else** que no tiene **if**.

```
if (x>0) ;
System.out.println("Valor positivo");
else System.out.println("Valor negativo o cero");
```

### *Instrucción condicional múltiple*

Otra modalidad de instrucción **if...else...** es la llamada instrucción **if...else... múltiple**, en la que la instrucción de cada **else** es a su vez una instrucción **if...else....** Se escribe como sigue:

```
if (B1) S1
else if (B2) S2
else if (B3) S3
...
else if (Bn) Sn
else Sn+1
```

Puede verse como una sola instrucción con  $n+1$  posibilidades de ejecución. El compilador evalúa las expresiones de tipo `boolean` una tras otra hasta que llega a una (`Bi`) que es `true`. Se ejecuta el bloque de instrucciones asociado (`Si`) y se salta el resto. Si ninguna de las condiciones da como resultado `true`, entonces se ejecuta el bloque (`Sn+1`) del `else` final. Sólo se ejecutará uno de entre los  $n+1$  bloques de instrucciones. El `else` final se puede omitir. En ese caso, si todas las condiciones son `false`, no se ejecutará ninguno de los bloques. Su diagrama de flujo puede verse en la figura 8.1(c).

A continuación, se muestra un ejemplo de instrucción condicional múltiple.

**Ejemplo 8.2.** Considérese el siguiente problema: Dada la nota media  $n$  de un alumno, obtener su equivalente en letra.

Es obvio que el resultado del problema depende del rango de valores en el que se encuentre la nota. Así, la nota equivalente en letra será:

- *Suspensos*, si  $0 \leq n < 5$ ,
- *Aprobado*, si  $5 \leq n < 7$ ,
- *Notable*, si  $7 \leq n < 9$ ,
- *Sobresaliente*, si  $9 \leq n < 10$ ,
- *Matrícula de Honor*, si  $n = 10$ , o
- un error, en cualquier otro caso.

Dada una nota media  $n$ , sólo puede darse una de las 6 posibilidades anteriores. Así pues, una instrucción condicional múltiple será la instrucción más adecuada para resolver este problema.

En la figura 8.2, el método `deNotaALetra` convierte una nota media (representada por el parámetro `nota` de tipo `double`) en su nota en letra equivalente (representada por la variable `notaLetra` de tipo `String`), usando una instrucción `if...else...` múltiple.

Por ejemplo, si la nota media es 8,75, se muestra por pantalla:

|                               |
|-------------------------------|
| Entrada/Salida Estándar       |
| Introduce la nota media: 8.75 |
| Notable (8.75)                |

```
/**
 * Clase TestIfElseMultiple: ejemplo de uso de la instrucción
 * if...else... múltiple.
 * @author Libro IIP-PRG
 * @version 2011
 */
import java.util.*;
public class TestIfElseMultiple {
 /** Dado un valor real que representa una nota media,
 * obtiene su equivalente en letra.
 * @param nota - double correspondiente a la nota media.
 * @return String - nota equivalente en letra.
 */
 public static String deNotaaLetra(double nota) {
 String notaLetra;
 if (nota<0.0 || nota>10.0)
 notaLetra="Error";
 else if (nota<5.0)
 notaLetra="Suspensos";
 else if (nota<7.0)
 notaLetra="Aprobado";
 else if (nota<9.0)
 notaLetra="Notable";
 else if (nota<10.0)
 notaLetra="Sobresaliente";
 else notaLetra="Matrícula de Honor";

 return notaLetra;
 }
 /** Método principal.
 * @param args - String[]
 */
 public static void main(String[] args) {
 Scanner tec = new Scanner(System.in).useLocale(Locale.US);
 System.out.print("Introduce la nota media: ");
 double notaMedia = tec.nextDouble();
 String notaEnLetra = deNotaaLetra(notaMedia);
 System.out.println(notaEnLetra + " (" + notaMedia + ")");
 }
}
```

Figura 8.2: Ejemplo de uso de la instrucción if...else... múltiple.

### 8.1.2 Instrucción switch

En la resolución de muchos problemas surge la necesidad de efectuar cálculos diferentes para distintos valores de una única variable o expresión simple. Si, por ejemplo, alguna variable de entrada tiene 10 posibles valores distintos, y para cada uno de los mismos es necesario un tratamiento especial, se podría utilizar una instrucción `if...else...` anidada; sin embargo ésta suele ser difícil de leer y mantener, por ello Java introduce una instrucción condicional adicional con el objetivo de facilitar el tratamiento de casos como el anterior. Es la instrucción `switch`, que tiene la siguiente forma general:

```
switch (expresion) {
 case val1: [SC1] [break;]
 case val2: [SC2] [break;]

 case valn: [SCn] [break;]
 [default: [SCn+1]]
}
```

donde, en primer lugar, hay que notar que aquellas componentes que aparecen entre corchetes indican opcionalidad (pueden o no aparecer); así, por ejemplo, la parte etiquetada como `default` puede aparecer o no. Además,

- `expresion` es una expresión de un tipo simple (excepto `float` y `double`) o un `String` (desde la versión 7.0 de Java).
- Los valores `val1`, `val2`, ... `valn`, son todos valores compatibles con el de `expresion`.
- `SC1`, `SC2`, ... `SCn+1`, son instrucciones (o secuencias de instrucciones) cualesquiera. Nótese que tanto dichas instrucciones (o secuencias) como la instrucción `break` son opcionales.

La ejecución de una instrucción `switch` es como sigue: se evalúa en primer lugar `expresion`, comparándose el valor resultante con el de cada uno de los valores asociados a las etiquetas `case`. Si coincide con alguno entonces se ejecuta todo el código que sigue a la etiqueta `case` correspondiente (incluso el asociado a las etiquetas `case` posteriores) hasta que, o bien finaliza todo el `switch`, o bien se encuentra una instrucción `break` (en cuyo caso toda la instrucción `switch` acaba inmediatamente). Si ninguno de los valores coincide con el de la `expresion` entonces se ejecuta, si existe, la instrucción o bloque asociado a la etiqueta `default`. El programa continúa a partir de la instrucción que sigue al `switch`. Su diagrama de flujo puede verse en la figura 8.1(d).

El siguiente es un ejemplo de uso de la instrucción **switch**.

**Ejemplo 8.3.** Considérese el siguiente problema: Dado el número de un mes del año, se desea saber a qué estación pertenece.

El análisis por casos de este problema se puede realizar, sabiendo que cada estación dura aproximadamente 3 meses, agrupando los meses por estaciones de este modo:

- *Primavera*: marzo, abril y mayo (meses 3, 4 y 5).
- *Verano*: junio, julio y agosto (meses 6, 7 y 8).
- *Otoño*: septiembre, octubre y noviembre (meses 9, 10 y 11).
- *Invierno*: diciembre, enero y febrero (meses 12, 1 y 2).

En la figura 8.3, se resuelve el problema usando una instrucción **switch**. El programa muestra la estación del año a la que pertenece un número de mes dado o un mensaje de error en caso de que el número de mes no sea válido. El método **deNumMesaEstacion**, siguiendo el análisis por casos anterior, devuelve el **String** **Invierno** si **numMes** es 1, 2 o 12; **Primavera** si **numMes** es 3, 4 o 5; **Verano** si **numMes** es 6, 7 u 8; **Otoño** si **numMes** es 9, 10 u 11; y devuelve **Error** en cualquier otro caso.

Por ejemplo, para el mes 7 (julio), el programa muestra por pantalla:

|                                         |
|-----------------------------------------|
| Entrada/Salida Estándar                 |
| Introduce un mes del año (en número): 7 |
| Mes: 7 ---> Verano                      |

Toda instrucción **switch** puede traducirse a una instrucción **if...else...** equivalente. Pero no toda instrucción **if...else...** puede traducirse a una instrucción **switch** equivalente.

La instrucción **switch** del método **deNumMesaEstacion** de la figura 8.3 es equivalente a la instrucción **if...else...** múltiple que sigue. Sin embargo, la instrucción **if...else...** del método **deNotaALetra** de la figura 8.2 no puede escribirse como una instrucción **switch** puesto que la expresión de un **switch** no puede evaluarse a un valor de tipo **double**.

```


/**
 * Clase TestSwitchInt: ejemplo de uso de la instrucción switch.
 * @author Libro IIP-PRG
 * @version 2011
 */
import java.util.*;
public class TestSwitchInt {
 /** Dado un valor entero que representa un mes,
 * obtiene la estación a la que pertenece.
 * @param numMes - int representando un mes.
 * @return String - mes equivalente en letra.
 */
 public static String deNumMesaEstacion(int numMes) {
 String estacion = "";
 switch (numMes) {
 case 1:
 case 2:
 case 12: estacion = "Invierno"; break;
 case 3:
 case 4:
 case 5: estacion = "Primavera"; break;
 case 6:
 case 7:
 case 8: estacion = "Verano"; break;
 case 9:
 case 10:
 case 11: estacion = "Otoño"; break;
 default: estacion = "Error"; break;
 }
 return estacion;
 }
 /** Método principal.
 * @param args - String[]
 */
 public static void main(String[] args) {
 Scanner tec = new Scanner(System.in);
 System.out.print("Introduce un mes del año (en número): ");
 int mes = tec.nextInt();
 String estacion = deNumMesaEstacion(mes);
 if (estacion.equals("Error"))
 System.out.println("Mes no válido");
 else System.out.println("Mes: " + mes + "-->" + estacion);
 }
}


```

Figura 8.3: Ejemplo de uso de la instrucción switch.

```
if (numMes==12 || numMes==1 || numMes==2)
 estacion = "Invierno";
else if (numMes>=3 && numMes<=5)
 estacion = "Primavera";
else if (numMes>=6 && numMes<=8)
 estacion = "Verano";
else if (numMes>=9 && numMes<=11)
 estacion = "Otoño";
else estacion = "Error";
```

Desde la versión 7.0 de Java, se permite el uso de un **String** como expresión del **switch**.

**Ejemplo 8.4.** Considérese el siguiente problema: Dado un **String** que representa un mes del año, se desea obtener su equivalente en número.

Se distinguen 13 casos posibles, uno por cada mes más el caso en el que el mes no es válido. El problema se puede resolver usando una instrucción **switch** cuya **expresión** es el mes, con 12 **case** cuyas expresiones serán los nombres de los meses y un caso **default** para tratar el mes no válido.

El programa de la figura 8.4 muestra el número del mes basándose en el valor del **String** **mes**. En el método **deMesaNúmero**, el **String** en la expresión del **switch** se compara con las expresiones de cada **case** usando el método **equals** de la clase **String**. El valor de **mes** se convierte a minúsculas (con el método **toLowerCase**), y todos los **String** de las etiquetas de cada **case** están también en minúsculas.

Por ejemplo, si el usuario teclea **Agosto**, el programa muestra por pantalla:

|                                                        |
|--------------------------------------------------------|
| Entrada/Salida Estándar                                |
| Introduce un mes del año: Agosto<br>Agosto es el mes 8 |

Una aplicación habitual de la instrucción **switch** es en la gestión de un menú. Un menú es una lista de opciones, de entre las cuales el usuario puede seleccionar una de ellas. El programa tiene que responder a cada opción posible de una forma distinta. Si las opciones están numeradas 1, 2, ..., entonces el número de la opción elegida puede ser utilizado en una instrucción **switch** para seleccionar la respuesta correcta. En el código que sigue, cuando se elige, por ejemplo, la opción 2 se muestra por pantalla **Perímetro = 31.4** y si se elige una opción distinta de las tres posibles, se muestra por pantalla **Opción no válida**.

```
/*
 * Clase TestSwitchString: ejemplo de uso de la instrucción
 * switch con un String.
 * @author Libro IIP-PRG
 * @version 2011
 */
import java.util.*;
public class TestSwitchString {
 /** Dada una cadena de caracteres que representa un mes,
 * obtiene su equivalente en número.
 * @param mes - String representado un mes.
 * @return int - mes equivalente en número.
 */
 public static int deMesaNumero(String mes) {
 int numMes = 0;
 switch (mes.toLowerCase()) {
 case "enero": numMes = 1; break;
 case "febrero": numMes = 2; break;
 case "marzo": numMes = 3; break;
 case "abril": numMes = 4; break;
 case "mayo": numMes = 5; break;
 case "junio": numMes = 6; break;
 case "julio": numMes = 7; break;
 case "agosto": numMes = 8; break;
 case "septiembre": numMes = 9; break;
 case "octubre": numMes = 10; break;
 case "noviembre": numMes = 11; break;
 case "diciembre": numMes = 12; break;
 default: numMes = 0; break;
 }
 return numMes;
 }
 /** Método principal.
 * @param args - String[]
 */
 public static void main(String[] args) {
 Scanner tec = new Scanner(System.in);
 System.out.print("Introduce un mes del año: ");
 String mes = tec.nextLine();
 int numeroMes = deMesaNumero(mes);
 if (numeroMes==0) System.out.println("Mes no válido");
 else System.out.println(mes + " es el mes " + numeroMes);
 }
}
```

Figura 8.4: Ejemplo de uso de la instrucción switch con un String.

```
Circulo c = new Circulo(5.0,"verde",0,0);
...
System.out.println(" MENÚ");
System.out.println("1. Área de un círculo");
System.out.println("2. Perímetro de un círculo");
System.out.println("3. Datos de un círculo");
System.out.print("\nElige una opción: ");
int opc = teclado.nextInt();

switch(opc) {
 case 1: System.out.println("Área = " + c.area());
 break;
 case 2: System.out.println("Perímetro = " + c.perimetro());
 break;
 case 3: System.out.println(c.toString());
 break;
 default: System.out.println("Opción no válida");
}
```

## 8.2 El operador ternario

El lenguaje Java (al igual que los lenguajes C y C++) introduce un operador especial, llamado *operador condicional* o *ternario*, con un uso parecido al de una instrucción condicional. La forma general de una expresión en la que aparece dicho operador es:

$$\text{exprbool} ? \text{expr1} : \text{expr2}$$

donde

- **exprbool** es cualquier expresión de tipo **boolean** y
- **expr1** y **expr2** son expresiones cualesquiera del mismo tipo.

Recuérdese que lo que se presenta es un operador y no una instrucción y que un operador sólo puede aparecer formando parte de una expresión y no tiene sentido de forma aislada.

La evaluación de toda la expresión anterior se efectúa del modo siguiente: en primer lugar se evalúa la **exprbool** que se encuentra a la izquierda del interrogante. Si el resultado de dicha evaluación es **true** entonces el valor de toda la expresión es el valor de la **expr1**, en caso contrario, el valor de toda la expresión es el valor de la **expr2**.

Por ejemplo, la siguiente instrucción, en la que interviene el operador ternario, asigna a la variable `max` el valor mayor de entre los de `a` y `b`.

```
int a, b, max;
...
max = a>b ? a : b;
...
```

Es equivalente a la siguiente instrucción:

```
int a, b, max;
...
if (a>b) max = a;
else max = b;
...
```

En el siguiente ejemplo el operador ternario es uno de los argumentos de una instrucción `System.out.println` que muestra por pantalla el resultado de dividir `a` entre `b` o muestra un 0 si `a` es cero o `b` es cero:

```
int a, b;
...
System.out.println("Resultado: " + ((a!=0 && b!=0) ? a/b : 0));
...
```

## 8.3 Algunos ejemplos

### *Comprobación de si una fecha es correcta*

El problema a resolver es comprobar si una fecha dada en el formato *día, mes y año*, es una fecha correcta, es decir, si se cumple que  $año > 0$ ,  $1 \leq mes \leq 12$  y  $1 \leq día \leq$  número de días del mes *mes*. En función del valor *mes*, se puede saber el número total de días del mes, teniendo en cuenta que los meses 1, 3, 5, 7, 8, 10 y 12 son meses de 31 días; los meses 4, 6, 9 y 11 son de 30 días y el mes 2 es de 29 o 28 días según el año sea o no bisiesto. Un año es bisiesto si es divisible por 4 y no por 100 o es divisible por 400.

Se dispone de una clase `Fecha` para representar fechas en dicho formato, que incluye un método privado `bisiesto` para comprobar si el año de la fecha actual es o no bisiesto. El problema se puede resolver añadiendo un método `esCorrecta` a dicha clase que compruebe si una fecha es correcta. En este método se comprobará que los atributos `dia`, `mes` y `año` del objeto `this` tienen los valores adecuados, es decir, aquéllos que hacen que la fecha sea correcta ( $año > 0$ ,  $1 \leq mes \leq 12$  y  $1 \leq dia \leq$  número de días del mes *mes*).

En la figura 8.5, se muestra la solución del método `esCorrecta`. En primer lugar, se declara e inicializa la variable `correcta` a `false` (línea 21). Este valor sólo se cambia a `true` si se comprueba que la fecha es correcta (línea 34). La instrucción principal del cuerpo del método es una instrucción condicional simple (línea 22), cuya condición comprueba que los valores de los atributos `dia`, `mes` y `año` son todos válidos (nótese que se comprueba  $1 \leq \text{dia} \leq 31$ ). Si la condición es falsa, el método directamente devuelve `false`. Sino, la instrucción `switch` (línea 24) se utiliza para determinar el número de días totales del mes `mes`, asignando a la variable `diasMes` el valor adecuado para el mes 2 (líneas 25 a 27), los meses 4, 6, 9 y 11 (líneas 28 a 31) y el resto de meses (línea 32). A continuación, si se cumple que el valor de `dia` es menor o igual que el de `diasMes`, la fecha es correcta y el método devuelve `true`. En caso contrario, devuelve `false`.

La instrucción `switch` anterior es equivalente a la siguiente instrucción `if...else...` múltiple:

```
if (mes==2)
 if (bisiesto()) diasMes = 29;
 else diasMes = 28;
else if (mes==4 || mes==6 || mes==9 || mes==11) diasMes = 30;
else diasMes = 31;
```

### *Comprobación de si una fecha es anterior a otra*

En este caso, se desea comprobar si una fecha (`dia1`, `mes1` y `año1`) es anterior a otra fecha dada (`dia2`, `mes2` y `año2`). Para comparar dos fechas, basta comparar componente a componente empezando por el año, como sigue:

- si  $año1 < año2$ , la primera fecha sí es anterior a la segunda.
- si  $año1 = año2$ , entonces se compara el mes:
  - si  $mes1 < mes2$ , la primera fecha sí es anterior a la segunda.
  - si  $mes1 = mes2$ , entonces se compara el día:
    - si  $dia1 < dia2$ , la primera fecha sí es anterior a la segunda.
    - si  $dia1 \geq dia2$ , la primera fecha no es anterior a la segunda.
  - si  $mes1 > mes2$ , la primera fecha no es anterior a la segunda.
- si  $año1 > año2$ , la primera fecha no es anterior a la segunda.

```

1 /**
2 * Clase Fecha: define una fecha en formato día, mes y año.
3 * @author Libro IIP-PRG
4 * @version 2011
5 */
6 public class Fecha {
7 private int dia, mes, año; // atributos
8 ...
9 /** Comprueba si el año de la fecha actual es o no bisiesto.
10 * @return boolean - devuelve true si año es bisiesto;
11 * en caso contrario, devuelve false.
12 */
13 private boolean esBisiesto() {
14 return ((año%4==0) && (año%100!=0)) || (año%400==0);
15 }
16 /** Comprueba si la fecha actual es o no correcta.
17 * @return boolean - devuelve true si la fecha actual es correcta;
18 * en caso contrario, devuelve false.
19 */
20 public boolean esCorrecta() {
21 boolean correcta = false;
22 if ((año>0) && (mes>=1 && mes<=12) && (dia>=1 && dia<=31)) {
23 int diasMes;
24 switch(mes) {
25 case 2: if (bisiesto()) diasMes = 29;
26 else diasMes = 28;
27 break;
28 case 4:
29 case 6:
30 case 9:
31 case 11: diasMes = 30; break;
32 default: diasMes = 31; break;
33 }
34 if (dia<=diasMes) correcta = true;
35 }
36 return correcta;
37 }
38 /** Comprueba si la fecha actual es anterior a otra fecha dada.
39 * @return boolean - devuelve true si la fecha actual es
40 * anterior a la fecha dada; en caso
41 * contrario, devuelve false.
42 */
43 public boolean esAnterior(Fecha f) {
44 boolean anterior=false;
45 if (año<f.año) anterior=true;
46 else if (año==f.año)
47 if (mes<f.mes) anterior=true;
48 else if ((mes==f.mes)&&(dia<f.dia)) anterior=true;
49 return anterior;
50 }
51 }
```

Figura 8.5: Clase Fecha.

Este análisis por casos se puede simplificar intentando agrupar los casos en los que la primera fecha es anterior a la segunda, de la siguiente forma:

- si  $año1 < año2$ , la primera fecha sí es anterior a la segunda.
- si  $año1 = año2$ , entonces se compara el mes:
  - si  $mes1 < mes2$ , la primera fecha sí es anterior a la segunda.
  - si  $mes1 = mes2$  y  $dia1 < dia2$ , la primera fecha sí es anterior a la segunda.
- en cualquier otro caso, la primera fecha no es anterior a la segunda.

Para resolver este problema se puede añadir a la clase **Fecha** un método **esAnterior** que compare la fecha **this** con una segunda fecha pasada como parámetro. A partir del análisis por casos anterior, la solución es inmediata, como se puede observar en la figura 8.5. En primer lugar, se declara e inicializa la variable **anterior** a **false** (línea 44). Se usa una instrucción **if...else...** anidada en la que el valor de **anterior** sólo se cambia a **true** si se comprueba que la fecha **this** es anterior a la fecha **f** (líneas 45, 47 y 48). En esos casos, el método devuelve **true**. En el resto de casos, devuelve **false**.

El método **esAnterior** también puede implementarse usando una expresión lógica, sin hacer uso de una instrucción condicional, como sigue:

```
public boolean esAnterior(Fecha f) {
 return (año<f.año) ||
 ((año==f.año) &&
 ((mes<f.mes) || ((mes==f.mes) && (dia<f.dia))));
}
```

En la figura 8.6, se muestra una clase **TestFecha** que prueba el comportamiento de los métodos **esAnterior** y **esCorrecta**, comprobando si dadas dos fechas correctas, la primera es anterior a la segunda. En el programa principal **main**, se pide al usuario que introduzca una fecha (líneas 15 a 18). Se crea **f1**, un objeto de tipo **Fecha**. Sólo si la fecha que representa **f1** es correcta (invocando al método **esCorrecta**, línea 21), se pide al usuario que introduzca una segunda fecha. Se crea **f2**, un objeto de tipo **Fecha**. Sólo si la fecha que representa **f2** es correcta (invocando de nuevo al método **esCorrecta**, línea 29), se comprueba si la primera fecha es o no anterior a la segunda (invocando al método **esAnterior**, línea 32), mostrándose por pantalla el mensaje correspondiente en cada caso.

```
1 /**
2 * Clase TestFecha: permite probar las funcionalidades de la clase
3 * Fecha.
4 * @author Libro IIP-PRG
5 * @version 2011
6 */
7 import java.util.*;
8 public class TestFecha {
9 /**
10 * Método principal.
11 * @param args - String[]
12 */
13 public static void main(String[] args) {
14 Scanner tec = new Scanner(System.in);
15 System.out.println("Introduce una fecha:");
16 System.out.print("Día? "); int dia = tec.nextInt();
17 System.out.print("Mes? "); int mes = tec.nextInt();
18 System.out.print("Año? "); int año = tec.nextInt();
19
20 Fecha f1 = new Fecha(dia,mes,año);
21 if (f1.esCorrecta()){
22 System.out.println("Primera fecha correcta.");
23 System.out.println("\nIntroduce otra fecha:");
24 System.out.print("Día? "); dia = tec.nextInt();
25 System.out.print("Mes? "); mes = tec.nextInt();
26 System.out.print("Año? "); año = tec.nextInt();
27
28 Fecha f2 = new Fecha(dia,mes,año);
29 if (f2.esCorrecta()){
30 System.out.println("Segunda fecha correcta.");
31 System.out.println("\nLa primera fecha ");
32 if (!f1.esAnterior(f2)) System.out.print("no ");
33 System.out.println("es anterior a la segunda.");
34 }
35 else System.out.println("Segunda fecha incorrecta.");
36 }
37 else System.out.println("Primera fecha incorrecta.");
38 }
39 }
```

Figura 8.6: Clase TestFecha.

El siguiente es un ejemplo de ejecución de la clase TestFecha:

Entrada/Salida Estándar

Introduce una fecha:

Día? 15

Mes? 10

Año? 2011

Primera fecha correcta.

Introduce otra fecha:

Día? 5

Mes? 10

Año? 2011

Segunda fecha correcta.

La primera fecha no es anterior a la segunda.

## 8.4 Problemas propuestos

1. ¿Cuál es la salida que produce este fragmento de código si `x` de tipo `int` vale 0? ¿Y si vale 1?

```
if (x==0)
 System.out.print("x es 0");
else System.out.print("x es ");
 System.out.print(x);
```

2. ¿Qué valor se asigna a `consumo` en la sentencia `if` siguiente si `velocidad` es 120?

```
if (velocidad > 80)
 consumo = 10.00;
else if (velocidad > 100)
 consumo = 12.00;
else if (velocidad > 120)
 consumo = 15.00;
```

3. En una tienda de electrodomésticos, por liquidación, se aplican distintos descuentos en función del total de las compras realizadas:

- Si  $total < 500\text{€}$ , no se aplica descuento.
- Si  $500\text{€} \leq total \leq 2000\text{€}$ , se aplica un descuento del 30 %.
- Si  $total > 2000\text{€}$ , entonces se aplica un descuento del 50 %.

Para implementar el problema en Java, se debe usar **una única variable** (`total` de tipo `double`) que almacena el total de las compras realizadas (antes de aplicar el descuento) y también el total a cobrar (tras aplicar el descuento correspondiente). Es decir, la variable `total` es, a la vez, dato y resultado. Se debe resolver el problema con **una única instrucción condicional (anidada)**. Completar dicha instrucción si la condición es `(total>=500)` o `(total<=2000)`.

4. Supóngase el siguiente fragmento de código, donde `x` es una variable `int` y `c` es una variable `char` que han sido convenientemente inicializadas:

```
if (x<0 && c=='x') System.out.println("Caso 1");
else if (x<0 && c!= 'x') System.out.println("Caso 2");
else if (x>=0 && c=='y') System.out.println("Caso 3");
else if (x>=0 && c!= 'y') System.out.println("Caso 4");
```

Se debe reescribir dicho código con la siguiente estructura, colocando las condiciones e instrucciones `System.out.println()` adecuadas, de forma que dados cualesquiera `x` y `c` el resultado escrito en la salida estándar coincida:

```
if (x < 0)
```

```
else
```

5. Para cada uno de los siguientes 4 bloques de código Java que contienen instrucciones condicionales, deducir el valor final de `x` si el valor inicial de `x` es 0:

1)  
if (x >= 0)  
 x++;  
else if (x >= 1)  
 x = x+2;

3)  
if (x < 0)  
 x = x+2;  
else x++;  
x--;

2)  
if (x >= 0)  
 x++;  
if (x >= 1)  
 x = x+2;

4)  
if (x > 0)  
 if (x <= 1)  
 x++;  
 else x--;

6. Sean `x` e `y` dos variables enteras, y sean las instrucciones condicionales siguientes:

1)      if (x==0)  
              if (y==0) System.out.println("x e y valen 0");  
              else System.out.println("Sólo x vale 0");  
    else  
        if (y==0) System.out.println("Sólo y vale 0");  
        else System.out.println("x e y son distintos de 0");

2)      if (x==0 && y==0)  
              System.out.println("x e y valen 0");  
    else if (x==0 && y!=0)  
              System.out.println("Sólo x vale 0");  
    else if (x!=0 && y==0)  
              System.out.println("Sólo y vale 0");  
    else System.out.println("x e y son distintos de 0");

Se pide:

- a) Contar cuántas operaciones de comparación de enteros realizan 1) y 2) en cada uno de los cuatro casos de `x` e `y` (`x` e `y` valen 0, `x` vale 0 e `y` es distinto de cero, etc.).

- b) De acuerdo con ello, ¿cuál de los condicionales 1) o 2) es más adecuado en cuanto a minimizar el número de operaciones de comparación realizadas?

7. ¿Qué se muestra por pantalla tras ejecutar el siguiente fragmento de código?

```
switch(2){
 case 1: System.out.println(1); break;
 case 2: System.out.println(2);
 case 3: System.out.println(3); break;
 default: System.out.println(4);
}
```

8. ¿Cuál es la salida que produce este programa si `primOpcion` de tipo `int` vale 1? ¿Y si vale 2?

```
switch (primOpcion + 1) {
 case 1: System.out.print("Ensalada ");
 break;
 case 2: System.out.print("Paella ");
 break;
 case 3: System.out.print("Emperador ");
 case 4: System.out.print("Helado ");
 break;
 default: System.out.print("Buen provecho");
}
```

9. Escribir un método estático que, dados dos valores enteros cualesquiera, los muestre por pantalla ordenadamente, de mayor a menor.
10. Escribir un método estático que, dados tres valores enteros cualesquiera, los muestre por pantalla ordenadamente, de mayor a menor.
11. Escribir un programa en Java que, dados tres valores enteros `a`, `b` y `c`, implemente distintas soluciones al análisis por casos siguiente, haciendo uso de: operadores cortocircuitados e instrucciones condicionales.

```
a > b -> true
a < b -> false
a == b y a > c -> true
a == b y a < c -> false
a == b y a == c -> false
```

12. Escribir un método estático que, dadas las coordenadas del centro ( $x, y$ ) de una circunferencia, muestre por pantalla si dicho centro está situado en:

- el primer cuadrante  $x > 0$  e  $y > 0$ ,
- el segundo cuadrante  $x < 0$  e  $y > 0$ ,
- el tercer cuadrante  $x < 0$  e  $y < 0$ ,

- el cuarto cuadrante  $x > 0$  e  $y < 0$ ,
  - el eje de abscisas  $x \neq 0$  e  $y = 0$ ,
  - el eje de ordenadas  $x = 0$  e  $y \neq 0$  o
  - el origen de coordenadas  $x = 0$  e  $y = 0$ .
13. Añadir un método a la clase **Fecha** tal que devuelva **true** si la fecha actual es festivo y devuelva **false** en caso contrario. Se considerarán únicamente los siguientes días festivos: 1 y 6 de enero; 1 de mayo; 12 de octubre; 1 de noviembre y 25 de diciembre. **NOTA:** Se debe utilizar **obligatoriamente** una instrucción condicional **switch** para su resolución.
14. Añadir un método a la clase **Fecha** que determine el número de días del mes de la fecha actual.
15. Añadir un método a la clase **Fecha** que obtenga la fecha del día siguiente a la fecha actual, suponiendo que es correcta.
16. Escribir un programa en Java que lea la hora de un día en notación de 24 horas (00:00 a 23:59) y dé la respuesta en notación de 12 horas (las 00:00 son las 12 de la noche (midnight); de 00:01 a 11:59 es AM; las 12:00 son las 12 de mediodía (noon) y de 12:01 a 23:59 es PM). Por ejemplo, si la entrada es 00:15, la salida será 12:15 AM; si la entrada es 11:25, la salida será 11:25 AM; si la entrada es 12:10, la salida será 12:10 PM; si la entrada es 13:35, la salida será 01:35 PM. El programa pedirá al usuario que introduzca exactamente 5 caracteres. Así, por ejemplo, las nueve en punto se introducirá como 09:00.
17. Dada la clase **Hora** del problema 4, capítulo 5, reescribir el cuerpo del método **toString()** usando condicionales, de modo que sólo se anteponga un "0" a las horas y a los minutos en los casos en que sea necesario, es decir, cuando sean menores que 10.
18. Escribir un programa en Java que determine el menor número de billetes y monedas de curso legal que equivalen a una cierta cantidad de euros (cambio óptimo). Por ejemplo, el cambio óptimo de 1755.45 euros es tres billetes de 500, uno de 200, uno de 50, uno de 5, dos monedas de 20 céntimos y una de 5 céntimos.
19. Escribir un programa en Java que calcule el salario semanal de un empleado pidiendo el número de horas trabajadas a la semana y el pago por hora, teniendo en cuenta que las horas extra (las que superan las 40) se pagan a un 50 % más.
20. Considérese el texto siguiente: "En una empresa el cálculo de las vacaciones pagadas se efectúa de la manera siguiente: si una persona lleva menos de un año en la empresa, tiene derecho a dos días por cada mes de presencia, si

no, al menos a 28 días. Si es un directivo y si tiene menos de 35 años y si su antigüedad es superior a 3 años, obtiene 2 días suplementarios. Si tiene menos de 45 años y si su antigüedad es superior a los 5 años, se le conceden 4 días suplementarios”.

- Escribir el programa correspondiente, que implicará el diseño de la clase **Trabajador** con atributos: la antigüedad expresada en meses **ant**, la edad en años **edad**, y la condición de ser o no directivo **dir**. Diseñar un método en dicha clase para obtener como resultado el número de días de vacaciones pagadas **diasPag**.
  - Diseñar una Clase-Programa que lea los datos por teclado, introduciendo verificaciones sobre la coherencia de los mismos. Por ejemplo, la edad ha de ser inferior a 65 años y superior a 18 años, etc. Se construirá un objeto **Trabajador** previo a mostrar por pantalla el resultado.
21. Realizar un programa que calcule la tarifa de una autoescuela teniendo en cuenta el tipo de carnet (A,B,C o D) y el número de prácticas realizadas. Precios de las matrículas: A 150 euros, B 325 euros, C 520 euros, D 610 euros. Precios por práctica según carnet: A 15 euros, B 21 euros, C 36 euros, D 50 euros.
  22. Dados dos números enteros, **num1** y **num2**, diseña un método estático que escriba uno de los dos mensajes: “el producto de los dos números es positivo o nulo” o bien “el producto de los dos números es negativo”. No hay que calcular el producto.
  23. Considérese el juego siguiente: un jugador (jug. A) elige un número entre 1 y 16. Otro jugador (jug. B) puede hacer cuatro preguntas de la forma: ¿es 13 el número?, a lo que el primer jugador (jug. A) responderá diciendo: **igual**, **mayor** o **menor**, según el número propuesto sea igual, mayor o menor que el elegido.
    - Estudiar una estrategia ganadora para el juego.
    - Realizar un programa en que el usuario sea el jugador A y el ordenador el B.
  24. Se desea escribir un programa para calcular la raíz cuadrada real de un número real cualquiera pedido inicialmente al usuario. Como dicha operación no está definida para los números negativos es necesario tratar, de algún modo, dicho posible error sin que el programa detenga su ejecución. Escribir distintas soluciones al problema anterior, haciendo uso de: operadores cortocircuitados, instrucciones condicionales, el operador ternario, etc.
  25. Escribir un programa para simular una calculadora. Considerar que los cálculos posibles son del tipo **num1 operador num2**, donde **num1** y **num2** son dos números reales cualesquiera y **operador** es uno de entre: +, -, \*, /. El

programa pedirá al usuario en primer lugar el valor `num1`, a continuación el operador y finalmente el valor `num2`. Resolver utilizando tanto instrucciones `if...else...`, como `switch`.

26. El juego infantil “Piedra, papel, tijeras” se juega entre dos jugadores. En una partida de este juego, cada jugador opta por decir, de forma simultánea al otro jugador, una de las palabras siguientes: “Piedra”, “Papel” o “Tijeras”, de forma que:

- Si un jugador dice “Piedra” y el otro “Tijeras”, entonces gana el jugador que haya elegido “Piedra”, ya que la piedra puede romper a las tijeras.
- Si un jugador dice “Tijeras” y el otro elige “Papel”, entonces gana el jugador que haya elegido “Tijeras”, ya que las tijeras cortan el papel.
- Si un jugador dice “Papel” y el otro dice “Piedra”, entonces gana el jugador que haya elegido “Papel”, ya que con el papel se puede envolver a la piedra.
- Por último, si ambos jugadores eligen lo mismo, entonces el resultado final es un empate o tablas entre los dos jugadores.

Escribir un programa en Java que implemente el juego “Piedra, papel, tijeras”. En el programa, el papel de uno de los jugadores lo realizará el ordenador, mientras que el del otro lo realizará el usuario. Cuando el programa se ejecute deberá:

- a) Seleccionar al azar uno de los tres elementos: “Piedra”, “Papel” o “Tijeras”.
- b) Pedir al usuario que elija uno de ellos. La introducción debe de realizarse mediante una cadena de texto (o `String`), de forma que mayúsculas y minúsculas se considerarán irrelevantes. En el caso de que el usuario introduzca una palabra distinta de las tres posibles, entonces el programa acabará, diciendo que la palabra no ha sido reconocida.
- c) En función de lo que se haya seleccionado inicialmente de forma aleatoria en el paso primero del programa y de lo que haya elegido el usuario en el segundo paso, el programa anunciará lo que ha elegido cada uno de los contrincantes y, siguiendo las reglas del juego enunciadas antes, deberá señalar el vencedor, caso de que exista, o que se ha producido un empate, caso de no haberlo.

27. Se desea resolver de forma general, la siguiente ecuación de segundo grado:

$$ax^2 + bx + c = 0$$

donde,  $a$ ,  $b$  y  $c$ , que son los coeficientes de la ecuación, son números reales cualesquiera, y  $x$ , la variable, cuyo valor o valores se desea conocer, puede ser un número real o complejo.

La solución de la ecuación anterior depende de los valores de los coeficientes y, en función de los mismos, cabe distinguir entre los siguientes casos:

- si  $a$  es 0 (no hay término de segundo orden), entonces
  - si  $b$  es también 0, y dependiendo de si  $c$  es 0 o no, la ecuación tiene infinitas soluciones (es de la forma  $0 = 0$ ), o es imposible (es de la forma  $c = 0$ ),
  - si  $b$  es distinto de 0, entonces la ecuación es de primer grado, y la solución  $x$  es el número real  $-c/b$ .
- si  $a$  es distinto de 0, entonces la solución de la ecuación se puede obtener siempre aplicando la fórmula:

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

donde al valor  $(b^2 - 4ac)$  se denomina discriminante. En dicho caso:

- si el discriminante es 0 la solución es única y es un número real,
- si el discriminante es positivo las dos soluciones son números reales,
- si el discriminante es negativo las dos soluciones son números complejos.

Para resolver el problema, de forma general, mediante un programa, se puede seguir una estrategia como la siguiente:

- a) Pedir inicialmente los valores de los coeficientes al usuario, leyéndolos de teclado.
- b) En función de ellos, decidir si la ecuación:
  - Es imposible,
  - tiene un número infinito de soluciones,
  - es una ecuación de primer grado, con solución real, o
  - es una ecuación de segundo grado y:
    - tiene una única solución real,
    - tiene soluciones reales, o
    - tiene soluciones complejas.

Para ello, se recomienda considerar el análisis de casos que se ha esbozado anteriormente.

- c) Escribir en la salida la advertencia o solución hallada según corresponda.

Tanto los coeficientes,  $a$ ,  $b$  y  $c$ , como otros posibles números, como el discriminante, son números reales, por lo que se mantendrán a lo largo de la ejecución mediante variables de dicho tipo, por ejemplo se pueden declarar como variables de tipo `double`. Como se ha visto, según el tipo de ecuación, algunas soluciones pueden ser números complejos. Sin embargo, en el

lenguaje no existe el tipo de datos número complejo de forma predefinida, por lo que se hace necesario representar dichos números de alguna forma alternativa. Se puede utilizar, por su sencillez, la forma cartesiana, en la que cada número complejo se representa mediante un par de valores de tipo real (`double`), uno para la parte real, y otro para la imaginaria. De esta manera, si  $n$  es un número real negativo cualquiera, entonces el valor complejo  $\sqrt{n}$ , es equivalente a  $\sqrt{|n|}i$ . Por ejemplo, el valor complejo  $\sqrt{-16}$ , es equivalente a  $\sqrt{|-16|}i$ , esto es:  $4i$ .

28. Diseñar una clase para poder efectuar etiquetas a partir del nombre de una persona. La idea es que, dado el nombre de una persona como, por ejemplo: Sara Ricard Senabre, se puedan generar, utilizando alguno de los métodos que se deberán construir en la clase, alguna de las `String` siguientes:

- "Sara Ricard Senabre" (todo el nombre completo),
- "S. Ricard" ( inicial, punto, y primer apellido),
- "Ricard Senabre, Sara" (apellidos, coma, nombre),
- "S.R.S." (iniciales en mayúsculas acabadas en punto).

La clase `Etiqueta` debe, por lo menos, incluir:

- Atributos diferentes para mantener el `nombre` de la persona, así como el `apellidoPrimero` y el `apellidoSegundo`.
- Un constructor que genere un objeto `Etiqueta` a partir de tres `String` correspondientes, cada una de ellas, a las partes del nombre.
- Un constructor que genere un objeto `Etiqueta` a partir de una única `String` que contendrá el nombre completo, pero con espacios en blanco para separar entre sí los componentes (nombre y apellidos).
- Tres métodos que devuelvan, cada uno de ellos, una `String`, valor de la `Etiqueta`, según se ha mostrado en el ejemplo inicial.

¿Qué ocurre en la clase que se ha efectuado si se da alguna de las condiciones siguientes?

- No se han escrito con mayúscula originalmente los nombres y los apellidos,
- la persona no tiene segundo apellido,
- la persona tiene más de un nombre.

Modificar adecuadamente la clase realizada resolviendo de forma razonable, esto es, sin pérdida o cambio de información, las condiciones descritas.

## Más información

- [Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.  
URL: <http://math.hws.edu/javanotes/>. Capítulo 3 (3.1, 3.5 y 3.6).
- [Ora11d] Oracle. *The Java<sup>TM</sup> Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Language Basics - Control Flow Statements.
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.  
Capítulo 3.

# Capítulo 9

## Estructuras de control: iteración

Los lenguajes de programación cuentan con instrucciones, genéricamente denominadas *iteraciones* o *bucles*, que permiten repetir la ejecución de una instrucción un número de veces tan grande como sea preciso.

Por ejemplo, supóngase una expresión en Java que genere dos valores enteros aleatorios 0 y 1, simulando el lanzamiento de una moneda, 0 para “cara” y 1 para “cruz”. Para comprobar que ambos valores son igualmente probables se puede repetir el cálculo de la expresión y comparar la frecuencia de aparición de cada valor.

En este capítulo se presentan las instrucciones de repetición existentes en el lenguaje Java y que permiten resolver este problema. Además, como se verá a lo largo del capítulo, gracias a dichas instrucciones se puede escribir la resolución de problemas más complejos, como los algoritmos presentados en el capítulo 1.

Con la iteración se completan las principales estructuras de control, herramientas básicas que permiten expresar los algoritmos en el lenguaje de programación. Más adelante, en el capítulo 11, se presenta la *recursión*, el otro modelo fundamental de expresión de los algoritmos que proporciona el lenguaje Java.

### 9.1 Iteraciones. El bucle while

Considérese el siguiente método, que recibe un entero y devuelve la suma de sus cifras, para valores del argumento relativamente pequeños.

```
/** 0<=n<10000 */
public static int sumaCifras(int n) {
 int result = 0;
 if (n>0) {
 result += n%10;
 n = n/10;
 if (n>0) {
 result += n%10;
 n = n/10;
 if (n>0) {
 result += n%10;
 n = n/10;
 }
 }
 }
 return result;
}
```

La expresión del código anterior resulta poco natural y redundante, dado que hay una secuencia de instrucciones que se ejecutará un número de veces variable, según *n* sea 0, o tenga 1, 2 o 3 cifras, habiéndose repetido su escritura el máximo número de veces posible.

Además, el programa queda muy limitado, no valiendo para sumar todas las cifras de enteros de 4 o más cifras. Se debe escribir un código que, para un entero mayor o igual que 0 cualquiera, repita el cálculo anterior el número de veces necesario.

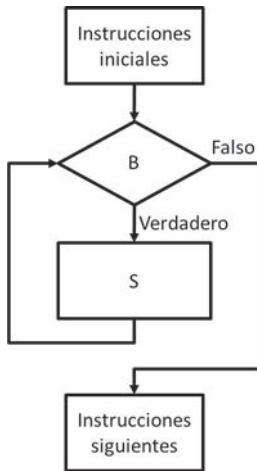
En Java, la iteración se puede escribir mediante la instrucción **while**, que tiene la forma

while (B) S

en donde:

- B puede ser cualquier expresión booleana y se le llama *guarda* del bucle,
- S puede ser cualquier instrucción válida de Java y se le llama *cuerpo* del bucle.

La ejecución de la instrucción **while** repite la ejecución del cuerpo del bucle mientras la guarda se evalúa a **true**. Cuando se llega a un estado en el que la guarda se evalúa a **false** la ejecución termina, como se muestra en el diagrama de flujo de la figura 9.1.



**Figura 9.1:** Diagrama de flujo de la instrucción `while`.

Cada vez que se ejecuta el cuerpo se dice que se ha ejecutado una *pasada* del bucle. Abusando del lenguaje, se llama también *iteración* a cada una de las pasadas del bucle, distinguiéndose por el contexto si se refiere al bucle o a una de sus repeticiones.

**Ejemplo 9.1.** El método `sumaCifras` anterior se puede escribir de una forma más adecuada con una instrucción `while`:

```

/** 0<=n */
public static int sumaCifras(int n) {
 int result = 0;
 while (n>0) {
 result += n%10;
 n = n/10;
 }
 return result;
}

```

A continuación se presenta una traza ejemplo de la ejecución del bucle, en el que el valor del argumento es 2735. En esta traza se muestra el estado por el que van pasando las variables del bucle después de ejecutar cada pasada.

| Estado de las variables  |      |        |
|--------------------------|------|--------|
| nº de pasadas ejecutadas | n    | result |
| 0 (inicio)               | 2735 | 0      |
| 1                        | 273  | 5      |
| 2                        | 27   | 8      |
| 3                        | 2    | 15     |
| 4                        | 0    | 17     |
|                          |      | n = 0  |

Inicialmente, y después de las tres primeras pasadas, se cumple la condición **n > 0**, por lo que el cuerpo del bucle se vuelve a ejecutar. Después de la 4<sup>a</sup> pasada, se alcanza el estado final del bucle, pues se llega a **n = 0**, y la guarda del bucle falla, o equivalentemente, **n** cumple la *condición de terminación* del bucle.

La ejecución del método termina entonces devolviendo el valor de **result**, 17 en este ejemplo.

Para repetir una instrucción un número prefijado de veces se puede escribir un bucle **while** que utilice un contador del número de pasadas realizadas. El bucle debe terminar cuando se haya completado el número de repeticiones deseado.

**Ejemplo 9.2.** El siguiente método repite el cálculo de un valor aleatorio 0 o 1 un número **n** de veces, para poder comparar la frecuencia de aparición de ambos valores. Cuanto mayor sea **n**, más se ajustarán las frecuencias obtenidas a la distribución de probabilidad.

```
/** Lanzamiento de una moneda n veces */
public static void pruebaMoneda(int n) {
 int cara = 0, cruz = 0;
 int repeticiones = 0;
 while (repeticiones < n) {
 int tirada = (int)(Math.random() * 2);
 switch (tirada) {
 case 0: cara++; break;
 case 1: cruz++; break;
 default: System.out.println("Error");
 }
 repeticiones++;
 }
 System.out.println("Frecuencias de 0 (cara) y 1 (cruz):");
 System.out.println(cara + " " + cruz);
}
```

## 9.2 Diseño de iteraciones

### 9.2.1 Estructura iterativa del problema

El ejemplo anterior pone de manifiesto una característica esencial de los problemas que admiten solución algorítmica o automatizable, y en consecuencia de los lenguajes de programación con los que expresar esta solución. Independientemente de lo complejo o grande que sea el problema, la solución debe cumplir las siguientes condiciones:

- Las instrucciones que la componen se deben poder describir de forma finita,
- su ejecución, por larga que sea, debe ser finita, abarcable en el tiempo.

Para resolver problemas suficientemente sencillos, como los ejemplos de los capítulos previos, bastaba con una serie de instrucciones de extensión perfectamente definida. En otros problemas, como el de la sección 9.1, el número de instrucciones elementales que se llegan a ejecutar es indeterminado.

Ante problemas indeterminadamente “largos” o “grandes”, resulta imposible expresar su solución como una secuencia indeterminadamente larga de instrucciones todas diferentes. Ello significa que necesariamente debe haber algún patrón de comportamiento que se repite y que permite darle a la resolución del problema una estructura iterativa.

**Ejemplo 9.3.** Supóngase que  $a$  y  $n$  son dos enteros no negativos. Se desea calcular el producto  $a \cdot n$  realizando únicamente sumas.

Para resolver el problema propuesto, se debe tener en cuenta que, por definición,

$$a \cdot n = \underbrace{a + a + a + \dots + a}_n \quad \text{si } n > 0; \quad a \cdot 0 = 0$$

que si se prefiere, se puede escribir de la siguiente forma:

$$a \cdot n = 0 + \underbrace{a + a + a + \dots + a}_n$$

La expresión anterior pone de manifiesto que la resolución del problema pasa por sumar  $a$  consigo misma un número  $n$  de veces,  $n > 0$ .

Ello sugiere de inmediato el uso de una iteración, en la que si **a**, **n** son variables con los valores a multiplicar, en sucesivas pasadas se acumulase el resultado de sucesivas sumas en una variable **producto**, declarada al efecto e inicializada a 0, de forma que:

|                                     |                               |
|-------------------------------------|-------------------------------|
| Tras la 1 <sup>a</sup> pasada:      | <b>producto</b> = <b>a</b>    |
| tras la 2 <sup>a</sup> pasada:      | <b>producto</b> = $2 \cdot a$ |
| tras la 3 <sup>a</sup> pasada:      | <b>producto</b> = $3 \cdot a$ |
| ...                                 | ...                           |
| tras la $n$ -ésima y última pasada: | <b>producto</b> = $n \cdot a$ |

Se puede declarar una variable **i** que permita contar el número de pasadas que se van haciendo, o lo que es lo mismo, el número de veces que se lleva acumulado el valor de **a** en **producto**. Entonces, la siguiente traza correspondería a la ejecución de una instrucción iterativa que implementase la estrategia anterior:

| Estado de las variables  |          |                 |
|--------------------------|----------|-----------------|
| nº de pasadas ejecutadas | <b>i</b> | <b>producto</b> |
| 0 (inicio)               | 0        | 0               |
| 1                        | 1        | $a$             |
| 2                        | 2        | $2 \cdot a$     |
| 3                        | 3        | $3 \cdot a$     |
| ...                      | ...      | ...             |
| $n$                      | $n$      | $n \cdot a$     |

$i < n$

En esta traza se observa que la transformación entre dos estados consecutivos cualesquiera se consigue con un mismo bloque de instrucciones:

- Incrementar **producto** en **a**,
- incrementar **i** en 1.

Con ello, en cualquiera de los estados se mantiene el siguiente patrón o relación entre las variables del bucle:

$$\text{producto} = i \cdot a$$

independientemente del número de pasadas que se lleguen a realizar. En efecto:

- Inicialmente **producto** e **i** lo cumplen al haberse iniciado ambas a 0,
- si se parte de un estado en que se cumple  $\text{producto} = i \cdot a$ , el incremento de **producto** en **a** y de **i** en 1 lleva a un estado en el que dicha relación se mantiene.

Esta relación tiene la información precisa para indicar que en cuanto  $i = n$  en **producto** se llega al resultado deseado, y el bucle debe terminar. Equivalentemente, el bucle debe continuar mientras  $i < n$ .

En resumen, se puede escribir el siguiente método:

```
/** a>=0, n>=0 */
public static int prod(int a, int n) {
 int i = 0, producto = 0;
 while (i<n) {
 producto = producto+a;
 i++;
 }
 return producto;
}
```

Cabe remarcar que **producto** se inicia a 0 para poder empezar a acumular valores en esta variable, siendo un estado particular del invariante o patrón general del bucle. Además, si el valor de **n** es 0, no se realiza ninguna pasada del bucle, y de inicio **producto** tiene el resultado requerido.

En general, la estrategia iterativa de resolución de un problema se puede expresar como una *relación invariante* entre las variables del bucle correspondiente, de forma que:

- Se cumple desde el inicio,
- se mantiene tras cada pasada del bucle,
- en el caso particular del estado final, es decir, junto con la condición de terminación, implica la obtención del resultado.

Esta relación entre las variables, explícita o implícitamente, estructura iterativamente la resolución del problema, y permite guiar la escritura de las componentes básicas del bucle:

1. *Inicialización de las variables.* Previamente a la entrada en el bucle, se inicializan adecuadamente las variables para ajustarlas a la relación invariante. Después de la inicialización, se debe poder empezar a aplicar las instrucciones del cuerpo.
2. *Guarda del bucle.* El estado final se distinguirá por la condición de terminación. La guarda del bucle deberá expresar por lo tanto que no se ha alcanzado esta condición.

3. *Cuerpo del bucle.* Es la instrucción que permite avanzar pasada a pasada hacia la resolución del problema. Se aplica desde el inicio a todos los estados para los que se cumple la guarda del bucle, y que se ajustan a un patrón común o invariante, de modo que se pasa desde cualquiera de ellos al estado siguiente mediante el mismo juego de instrucciones.

Estas consideraciones acerca de las componentes de la iteración no sólo tienen interés teórico, sino práctico. Si se toman en cuenta al diseñar una iteración, en general el desarrollo y depuración del código correspondiente será más sistemático y sencillo.

### 9.2.2 Terminación de la iteración

Como se ha subrayado anteriormente, una característica esencial de las soluciones algorítmicas es su ejecución en un tiempo finito. La instrucción de iteración, por su propia estructura es susceptible de errores que precisamente conducen a ejecuciones infinitas. Considérese de nuevo, por ejemplo, el problema del apartado anterior, en el que se ha introducido un pequeño cambio en la guarda del bucle:

```
/** 0<=n */
public static int sumaCifras(int n) {
 int result = 0;
 while (n>=0) {
 result += n%10;
 n = n/10;
 }
 return result;
}
```

Puede parecer que esta modificación sólo produce una iteración más, superflua por otra parte y que no modifica el resultado. Sin embargo su efecto es más importante, pues hace que el bucle no termine.

| Estado de las variables  |     |        |
|--------------------------|-----|--------|
| nº de pasadas ejecutadas | n   | result |
| 0 (inicio)               | 35  | 0      |
| 1                        | 3   | 5      |
| 2                        | 0   | 8      |
| 3                        | 0   | 8      |
| ...                      | ... | ...    |

En este algoritmo se mantiene invariante en cada estado que **result** es la suma de las cifras del argumento o valor inicial que se han eliminado de **n**. Si **n** > 0,

queda todavía alguna cifra en  $n$ , y una vez sumada, el valor de  $n$  se reduce a  $n/10$ , estrictamente más pequeño, con una cifra menos. En cambio, si se permite seguir ejecutando el bucle cuando  $n = 0$ ,  $n$  no cambia dado que  $0/10 = 0$ .

En el algoritmo correcto, en el que el cuerpo del bucle sólo se ejecuta cuando  $n > 0$ , se demuestra que la iteración termina teniendo en cuenta que:

- el valor de la variable entera  $n$  se mantiene acotado inferiormente, ya que si  $n > 0$  el resultado de aplicarle la división entera es  $n/10 \geq 0$ ;
- en cada pasada del bucle el valor de  $n$  se reduce a un valor entero estrictamente más pequeño, ya que si  $n > 0$  entonces  $n > n/10$ .

En resumen, a lo largo del bucle  $n$  es una cantidad entera que se mantiene acotada inferiormente y que decrece estrictamente en cada pasada. Se concluye que el bucle sólo puede realizar un número finito de pasadas.

En general, a cualquier medida dependiente de las variables del bucle, que sea entera, acotada inferiormente, y estrictamente decreciente con cada iteración, se le denomina *función limitadora o convergente*, dado que demuestra que el número de pasadas del bucle está limitado, es decir, el bucle acaba o converge.

**Ejemplo 9.4.** Considérese el siguiente problema: Dados dos enteros positivos se desea encontrar el máximo entero que divide a ambos. Un algoritmo que resuelve este problema es el conocido con el nombre de *algoritmo de Euclides de cálculo del máximo común divisor* (m.c.d.) y suele expresarse en los siguientes términos:

Dada una pareja  $a, b$  de enteros mayores que 0, se reemplaza el mayor de ellos por la resta del mayor menos el menor. Este proceso se repite sucesivamente hasta que la pareja de números coincide. Entonces, el valor de ambos es el máximo común divisor buscado.

Su funcionamiento se basa en la siguiente observación:

Sea  $m$  un divisor de  $a$  y  $b$ , es decir,  $a = m \cdot q_a$  y  $b = m \cdot q_b$ , con  $m > 0$ ,  $q_a > 0$ ,  $q_b > 0$ . Entonces:

$$|a - b| = m \cdot |q_a - q_b|$$

Dicho de otro modo, cualquier divisor de  $a$  y  $b$ , también divide simultáneamente a  $a$ ,  $b$ ,  $|a - b|$ . Además, el máximo  $m$  que divide a  $a$  y  $b$ , también es el máximo divisor común de  $a$ ,  $|a - b|$  y de  $b$ ,  $|a - b|$ .

Para expresar este algoritmo en Java se podría escribir una iteración en la que la referida pareja de números fuese un par de variables enteras  $a$ ,  $b$  iniciadas a  $a$  y  $b$  respectivamente. A continuación se muestra un ejemplo de cuál sería la evolución

de ambas variables si se les aplicase el algoritmo descrito, para el par de valores iniciales 247 y 152.

| Estado de las variables  |     |         |
|--------------------------|-----|---------|
| nº de pasadas ejecutadas | a   | b       |
| 0 (inicio)               | 247 | 152     |
| 1                        | 95  | 152     |
| 2                        | 95  | 57      |
| 3                        | 38  | 57      |
| 4                        | 38  | 19      |
| 5                        | 19  | 19      |
|                          |     | $a = b$ |

$a \neq b$

A lo largo del bucle, el par de valores que toman **a** y **b** mantiene el mismo m.c.d. En este ejemplo concreto, como se deduce del estado final, el m.c.d. buscado es 19.

En Java se puede escribir el siguiente método:

```
/** a>0, b>0 */
public static int mcd(int a, int b) {
 while (a != b)
 if (a>b) a = a-b;
 else b = b-a;
 return a;
}
```

Una posible pregunta que se plantea ante este algoritmo es si, como sucede con el ejemplo, para cualesquiera valores iniciales positivos de **a** y **b** siempre se acaba convergiendo a un estado en el que **a = b**.

La respuesta a la pregunta de si esto sucede en general, es que sí. El algoritmo lo fuerza haciendo que el máximo de **a** y **b** sea cada vez más pequeño, pero manteniéndolo acotado inferiormente: en efecto, **a** y **b** se mantienen  $> 0$  restando siempre el menor al mayor, y sólo si no son iguales. Por ejemplo, sean 2345 y 1, los respectivos parámetros reales del método. El algoritmo reduce **a** en 1 en cada pasada, y como no se llega a hacer 0, el número de pasadas no puede superar el propio valor inicial de **a**.

En definitiva, el argumento anterior ha permitido demostrar que el bucle termina en todos los casos, dado que si  $a \neq b$  se puede reducir el máximo de ambos manteniéndolos positivos, y este proceso no puede continuar indefinidamente, por lo que necesariamente se alcanza el estado final  $a = b$ .

En resumen, si se toma como convergente el máximo de  $a$  y  $b$ , es una cantidad:

- entera, acotada inferiormente (ambos  $a$  y  $b$  se mantienen  $> 0$ ),
- su valor disminuye en cada pasada del bucle,

con lo cual se demuestra que necesariamente se alcanza el estado final.

**Ejemplo 9.5.** Existe otra versión conocida del algoritmo de Euclides, que se puede considerar una variación de la anterior y cuyo propósito es reducir el número de pasadas realizadas en el cálculo. En el algoritmo anterior, antes de llegar a  $a = b$  se puede dar que uno de  $a, b$  sea un múltiplo del otro, triplicándolo o más. Supóngase sin pérdida de generalidad que fuese  $a$ , entonces el algoritmo anterior restaría  $b$  de  $a$  repetidas veces hasta igualarlos. Estos pasos se pueden abreviar comprobando directamente si el resto de  $a/b$  es 0, en cuyo caso se concluye que  $b$  es el m.c.d. buscado.

¿Qué sucede sin embargo si el resto es  $> 0$ ? En ese caso, si  $q$  y  $r$  son respectivamente el cociente y el resto de  $a/b$ :

$$a = q \cdot b + r, \quad q \geq 0, \quad b > r > 0$$

o lo que es lo mismo,  $a - q \cdot b = r > 0$ .

Como cualquier divisor de  $a, b$ , es divisor de  $a - q \cdot b$ , entonces es divisor de  $r$ . Y el cálculo del m.c.d de  $a, b$  se puede reducir al cálculo del m.c.d de  $b, r$ :

```
/** a>0, b>0 */
public static int mcd(int a, int b) {
 int r = a%b;
 while (r>0) {
 a = b;
 b = r;
 r = a%b;
 }
 return b;
}
```

Tal como ilustra la siguiente traza ejemplo para valores iniciales 114 y 209, la terminación del bucle queda patente si se toma como función limitadora  $b$ , que se mantiene  $> 0$ , y decreciente con cada pasada.

| Estado de las variables  |     |     |       |
|--------------------------|-----|-----|-------|
| nº de pasadas ejecutadas | a   | b   | r     |
| 0 (inicio)               | 114 | 209 | 114   |
| 1                        | 209 | 114 | 95    |
| 2                        | 114 | 95  | 19    |
| 3                        | 95  | 19  | 0     |
|                          |     |     | r > 0 |
|                          |     |     | r = 0 |

### 9.3 La instrucción for

La instrucción iterativa **for** tiene la siguiente sintaxis:

```
for ([Inicio]; B; [Avance]) S
```

en donde:

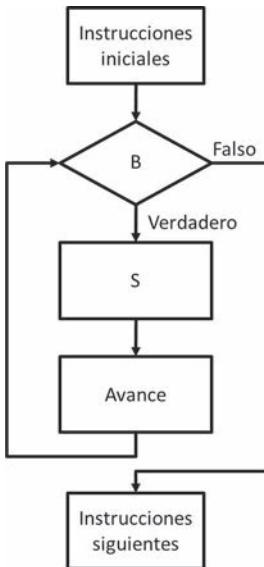
- Las partes entre corchetes son optativas,
- B es una expresión booleana, la guarda del bucle,
- S es una instrucción, el cuerpo del bucle,
- Inicio y Avance son unas listas, i1, i2, ..., in y a1, a2, ..., am respectivamente, cuyos elementos aparecen separados entre comas, que normalmente expresan la inicialización y la progresión del bucle.

El bucle **for** expuesto equivale al siguiente bucle **while**:

```
[i1; i2; ... in] // Inicio
while (B){
 S
 [a1; a2; ... am] // Avance
}
```

con la única salvedad de que el ámbito de las variables declaradas en **Inicio** se limita al propio bucle.

Esta forma alternativa del bucle se diseñó deliberadamente para que agrupase en una sola línea la inicialización, la condición de repetición y la parte final del cuerpo. Con ello se consigue a menudo expresar el bucle de una forma compacta, porque es frecuente que las instrucciones del final del cuerpo expresen el progreso hacia la terminación. Ello se puede resaltar escribiéndolo como la parte **Avance** del **for**.



**Figura 9.2:** Diagrama de flujo de la instrucción `for`.

En la figura 9.2 se muestra el diagrama de flujo de la instrucción `for`.

**Ejemplo 9.6.** Considérese la siguiente sucesión:

$$a_1 = 7,$$

$$a_i = a_{i-1} + i, \quad i \geq 2.$$

Supóngase que se desea un método:

```
/** n>=1 */
public static int sumaSuc(int n)
```

que obtenga la suma  $\sum_{i=1}^n a_i$  para  $n \geq 1$ .

Ello requiere el cálculo de los términos  $a_1, a_2, \dots, a_n$ , calculados mediante una iteración. Dado que para calcular cada término sólo es preciso recordar el anterior y el número de término, y que se pueden ir sumando a medida que se calculan, el bucle puede utilizar tres variables, que pasada a pasada mantendrán la siguiente relación invariante:

- $i$ , número del último término calculado,  $1 \leq i \leq n$ ,

- **term**, término  $i$ -ésimo, último término calculado,
- **suma**, suma de todos los términos calculados desde el 1 hasta el  $i$  inclusive.

A continuación se muestra una traza ejemplo para  $n = 6$  y a su derecha el bucle `while` correspondiente:

| Estado de las variables  |   |      |      |
|--------------------------|---|------|------|
| nº de pasadas ejecutadas | i | term | suma |
| 0 (inicio)               | 1 | 7    | 7    |
| 1                        | 2 | 9    | 16   |
| 2                        | 3 | 12   | 28   |
| 3                        | 4 | 16   | 44   |
| 4                        | 5 | 21   | 65   |
| 5                        | 6 | 27   | 92   |

```
public static int sumaSuc(int n){
 int term = 7, suma = 7, i = 1;
 while (i<n) {
 i++;
 term = term+i;
 suma = suma+term;
 }
 return suma;
}
```

La iteración se puede estructurar de forma ligeramente distinta, de modo que iniciados `term = 7` y `suma = 7`, se inicia `i = 2` para que el bucle se disponga directamente a calcular el siguiente término  $a_2$  sin incrementar previamente el valor de `i`. Es decir, ahora, tras cada pasada:

- **i** es el número del siguiente término a calcular,  $2 \leq i \leq n + 1$ ,
- **term** es el término  $i - 1$ -ésimo, último término calculado,
- **suma** es la suma de todos los términos calculados desde el 1 hasta el  $i - 1$  inclusive.

A continuación se muestra la traza para  $n = 6$  y el correspondiente bucle `while`:

| Estado de las variables  |   |      |      |
|--------------------------|---|------|------|
| nº de pasadas ejecutadas | i | term | suma |
| 0 (inicio)               | 2 | 7    | 7    |
| 1                        | 3 | 9    | 16   |
| 2                        | 4 | 12   | 28   |
| 3                        | 5 | 16   | 44   |
| 4                        | 6 | 21   | 65   |
| 5                        | 7 | 27   | 92   |

```
public static int sumaSuc(int n){
 int term = 7, suma = 7, i = 2;
 while (i<=n) {
 term = term+i;
 suma = suma+term;
 i++;
 }
 return suma;
}
```

El bucle de esta última versión se ajusta especialmente a un `for`, debido a que la última instrucción del cuerpo es el incremento de `i`, lo que permite tomarla como instrucción de avance:

```
public static int sumaSuc(int n) {
 int term = 7, suma = 7;
 for (int i=2; i<=n; i++) {
 term = term+i;
 suma = suma+term;
 }
 return suma;
}
```

De esta manera se consigue resumir en una sola línea la evolución de la variable `i`, leyéndose con claridad que en cada pasada se calcula uno de los términos de la sucesión a partir del anterior, acumulándose en la suma.

## 9.4 La instrucción do ... while

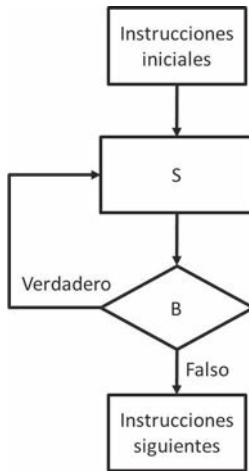
La sintaxis de la sentencia `do ... while` Java es:

```
do S while (B);
```

Se trata de una iteración en la cual se repite sucesivamente la ejecución de `S`, hasta que se llega a un estado en el que se deja de cumplir la expresión booleana `B`. La diferencia con respecto al `while` reside en:

- La guarda se comprueba después de ejecutar cada pasada, y no antes. Es por ello que en este bucle siempre se ejecuta al menos la primera pasada, como se observa en la figura 9.3;
- la sintaxis exige escribir el carácter ; como terminador de la instrucción.

**Ejemplo 9.7.** El siguiente método es un constructor de la clase `Círculo` presentada en el capítulo 2. Recibe como parámetro un `Scanner` abierto con la entrada estándar como fuente de caracteres, y permite al usuario crear un círculo con los valores de sus atributos introducidos desde teclado.



**Figura 9.3:** Diagrama de flujo de la instrucción `do ... while`.

```

public Circulo(Scanner teclado) {
 do {
 System.out.print("Teclee un valor del radio >0: ");
 this.radio = teclado.nextDouble();
 } while (this.radio<=0);

 System.out.print("Teclee las coordenadas del centro: ");
 this.centroX = teclado.nextDouble();
 this.centroY = teclado.nextDouble();
 teclado.nextLine();

 System.out.print("Teclee el color: ");
 this.color = teclado.nextLine();
}

```

La introducción de todos los valores se debe ejecutar al menos una vez; en el caso del radio, se ejecuta además las veces que sea preciso para asegurarse de que no quede `this.radio ≤ 0`.

## 9.5 Algunos ejemplos

### *Comprobación de si un número es primo: búsqueda del menor divisor*

Sea  $n$  un entero  $> 1$ . Se desea averiguar si  $n$  tiene algún divisor propio; en caso contrario  $n$  es primo. Es inmediato que 2 es primo, y que todos los pares no lo son. La discusión del problema se va a centrar pues en los impares  $> 1$ .

Sean  $\max$  y  $\min$  el máximo y el mínimo divisor de  $n$  respectivamente. Entonces

$$n = \max \cdot \min = \sqrt{n} \cdot \sqrt{n}$$

luego  $\max \geq \lfloor \sqrt{n} \rfloor$  y  $\min \leq \lfloor \sqrt{n} \rfloor$ . Ello significa que basta con buscar un divisor de  $n$  en los sucesivos impares  $3, 5, 7, \dots$ , sin superar el límite  $\lfloor \sqrt{n} \rfloor$ .

Sea **raiz** una variable iniciada a este límite. Entonces, una iteración que resuelva esta búsqueda puede utilizar una variable **i** que tome los valores de los sucesivos impares desde 3 en adelante sin sobrepasar **raiz**, y comprobando si **i** divide a **n**. Como es suficiente con encontrar un divisor, se buscará el más pequeño: en cada pasada, ninguno de los impares previos a **i** divide a **n**.

A continuación se presentan dos trazas ejemplos, que siguen esta estrategia, una para el primo  $n = 127$  (**raiz** =  $\lfloor \sqrt{127} \rfloor = 11$ ) y otra para el no primo  $n = 119$  (**raiz** =  $\lfloor \sqrt{119} \rfloor = 10$ ).

| $n = 127, \text{raiz} = 11$ |    | $n = 119, \text{raiz} = 10$         |   |
|-----------------------------|----|-------------------------------------|---|
| Estado de las variables     |    | Estado de las variables             |   |
| nº de pasadas ejecutadas    | i  | nº de pasadas ejecutadas            | i |
| 0 (inicio)                  | 3  | 0 (inicio)                          | 3 |
| 1                           | 5  | 1                                   | 5 |
| 2                           | 7  | 2                                   | 7 |
| 3                           | 9  |                                     |   |
| 4                           | 11 |                                     |   |
| 5                           | 13 |                                     |   |
|                             |    | $i \leq \text{raiz}, n \% i \neq 0$ |   |

Estas dos trazas son representativas de las dos posibles formas en las que puede acabar la búsqueda del mínimo divisor:

- Fracaso en la búsqueda: Se han examinado todos los impares entre 3 y **raiz** y ninguno de ellos divide a **n**, **i** llega a hacerse  $> \text{raiz}$  y se concluye que **n** es primo,
  - o
- éxito en la búsqueda: Se ha encontrado un divisor, se ha llegado a  $n \% i = 0$  y se concluye que **n** no es primo.

La condición de repetición debe expresar entonces que no se ha alcanzado el estado final, es decir:  $i \leq \text{raiz}$  y  $n \% i \neq 0$ .

Expresado en Java:

```
/** n>1 */
public static boolean esPrimo(int n) {
 boolean primo;
 if (n==2) primo = true;
 else if (n%2==0) primo = false;
 else { int i = 3, raiz = (int)Math.sqrt(n);
 while (i<=raiz && n%i!=0)
 i += 2;
 if (i>raiz) primo = true;
 else primo = false;
 }
 return primo;
}
```

### *Rotación de los caracteres de una línea*

Dada una variable entera  $n \geq 1$  se desea escribir en la salida una primera línea de  $n$  ‘A’ y  $n$  ‘Z’, y sucesivas líneas en las que el último carácter de cada línea se “desplace” al principio de la siguiente, “empujando” hacia la derecha al resto de caracteres, hasta que en la última línea escrita, todas las ‘A’ se hayan desplazado al final. A continuación se muestra un ejemplo con  $n = 4$ :

| Salida Estándar |
|-----------------|
| AAAAZZZZ        |
| ZAAAAXZZ        |
| ZZAAAXAZ        |
| ZZZAAAZAZ       |
| ZZZZAAAAA       |

Una posible resolución se basa en la observación de que se deben escribir  $n + 1$  líneas, todas ellas formadas por un primer grupo de ‘Z’, un segundo grupo de ‘A’, y un tercer grupo de ‘Z’, con el caso particular de que alguno de estos grupos puede tener 0 caracteres.

Si la variable  $i$  cuenta las líneas, suponiéndolas numeradas de 0 en adelante, la línea  $i$ -ésima se ajusta al siguiente patrón:

$\underbrace{\text{Z}\dots\text{Z}}_i \underbrace{\text{A}\dots\text{A}}_n \underbrace{\text{Z}\dots\text{Z}}_{n-i}$

con lo que cada línea tiene con respecto a la anterior una ‘Z’ más al inicio, desplazando el resto de caracteres hacia la derecha y quedando una ‘Z’ menos al final.

De acuerdo con este invariante, se puede escribir el siguiente código:

```
for (int i=0; i<=n; i++) {
 // Secuencia de Z de longitud i:
 for (int j=1; j<=i; j++) System.out.print('Z');
 // Secuencia de A de longitud n:
 for (int j=1; j<=n; j++) System.out.print('A');
 // Secuencia de Z de longitud n-i:
 for (int j=1; j<=n-i; j++) System.out.print('Z');
 System.out.println();
}
```

## 9.6 Problemas propuestos

1. Dados los siguientes bucles:

```
int i = 3;
while (i<=n) {
 System.out.println(i);
 i = i+3;
}
int i = 0;
while (i<n) {
 i = i+3;
 System.out.println(i);
}
```

¿Cuál de ellos escribe únicamente los múltiplos de 3, entre 3 y n inclusive?

- a) los dos,
  - b) sólo el de la izquierda,
  - c) sólo el de la derecha.
2. Escribir un método estático que calcule la suma de los  $n$  primeros números naturales ( $n \geq 0$ ); esto es  $\sum_{k=0}^n k$ .
3. Escribir a mano el resultado de ejecutar los siguientes bucles, donde  $n$  vale 5:

```
int i = n;
while (i>=0) {
 System.out.println(i*i);
 i--;
}
int i = 0;
while (i<=n) {
 System.out.println(i*i);
 i++;
}
for (int i=n; i>=0; i--)
 System.out.println(i*i);

int i = n+1;
while (i>0) {
 i--;
 System.out.println(i*i);
}
int i = 0;
while (i<=n) {
 System.out.println((n-i)*(n-i));
 i++;
}
for (int i=0; i<=n; i++)
 System.out.println((n-i)*(n-i));
```

4. En el siguiente fragmento de programa, ¿cuál es la salida si  $m = 0$ , si  $m = 1$ , y si  $m = 3$ ?

```
n = m;
for (i=0; i<n; i++)
 n--;
System.out.print(i);
```

5. Escribir un método estático que, dado un entero  $n \geq 0$  calcule  $\lfloor \sqrt{n} \rfloor$ , teniendo en cuenta que es el mínimo entero  $m \geq 0$  tal que  $n \cdot n \leq m$ . No se permite usar `Math.sqrt`, el cálculo se deberá hacer exclusivamente con datos enteros.

6. Escribir un método estático para calcular, mediante restas sucesivas, el cociente y resto de la división de dos números enteros  $a$  y  $b$ , con  $a \geq 0$  y  $b > 0$ . ¿Qué ocurriría si se ejecutase el método con  $b$  valiendo inicialmente 0? ¿Y si  $a$  fuese menor que 0?
7. Escribir un método estático que calcule iterativamente  $a^n$ , siendo  $a > 0$  y  $n \geq 0$ , teniendo en cuenta que:

$$a^n = \underbrace{a \cdot a \cdot a \cdot \dots \cdot a}_{n \text{ veces}} \quad \text{si } n > 0$$

$$a^0 = 1 \quad \text{si } n = 0$$

8. Escribir un método estático que, utilizando tan sólo incrementos y decrementos en uno, obtenga en una variable **resta**, la sustracción de dos números enteros no negativos.
9. Escribir un método estático que, dada una **String**, escriba en la salida estándar los caracteres de la cadena (accediendo a ellos con el método **charAt(int)**). Los caracteres se escribirán en líneas sucesivas, un carácter en cada línea. Como caso particular, si la cadena de caracteres es la vacía, no se escribirá nada.

Por ejemplo, para Java se escribirá:

Salida Estándar

```
J
a
v
a
```

10. Escribir un método estático que, dada una **String**, escriba en la salida estándar, consecutivos en la misma línea, los  $n$  caracteres de la palabra. Los caracteres deberán venir separados por ‘-’, de forma que, si la cadena de caracteres no está vacía, se escribirá el primer carácter, y después, repetidamente, se escribirán los caracteres restantes precedidos por un ‘-’.

Por ejemplo, para Java se escribirá:

Salida Estándar

```
J-a-v-a
```

11. ¿Cuál es la salida del siguiente bucle?

```
int suma=0;
while (suma<100) suma += 5;
System.out.println(suma);
```

12. Reescribir el siguiente bucle `while` con instrucciones `do ... while` y `for`.

```
int num=10;
while (num<=100) {
 System.out.println(num);
 num += 10;
}
```

13. ¿Cuál es la salida del siguiente bucle?

```
for (int i=1; i<4; i++) {
 System.out.print(i);
 System.out.print(" ");
 for (j=i; j>=1; j--)
 {
 System.out.print(j);
 System.out.print(" ");
 }
 System.out.print("\n");
}
```

14. ¿Cuál es la salida del siguiente bucle?

```
i = 1;
while (i*i<10) {
 j = i;
 while (j*j<100) {
 System.out.print(i + j);
 System.out.print(" ");
 j *= 2;
 }
 i++;
 System.out.print("\n");
}
System.out.print("\n*****");
```

15. Describir qué hace el siguiente segmento de código en Java. Se supone que `n` es una variable de tipo `int` previamente inicializada.

```
double resultado;
int i;
if (n<0) i = -n;
else i = n;
resultado = 0.0;
while (i>=1) {
 resultado += (1/i);
 i--;
}
```

16. Demostrar la terminación del siguiente bucle, donde  $x$  es una variable entera iniciada a algún valor positivo.

```
/* x>0 */
while (x%2==1)
 x = (3*x+1)/2;
```

Se sugiere tomar como función limitadora el número de ceros consecutivos que aparecen a la derecha en la representación en binario en  $x$ .

17. Escribir un programa que lea un  $n$  positivo de teclado y que escriba en la salida, línea a línea, los pares de enteros  $i, j$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq n$  y el valor que toma la expresión  $i + j + 2 \cdot i \cdot j$ . Dado que la suma y el producto son conmutativos, la expresión toma el mismo valor para el par  $(j, i)$  que para el par  $(i, j)$ , por lo que sólo se deberá calcular para uno de los pares. A continuación se muestra cuál debería ser el resultado del programa para  $n = 4$ :

| Salida Estándar |                   |
|-----------------|-------------------|
| Par 1,1:        | 1+1+2*1*1 vale 4  |
| Par 1,2:        | 1+2+2*1*2 vale 7  |
| Par 1,3:        | 1+3+2*1*3 vale 10 |
| Par 1,4:        | 1+4+2*1*4 vale 13 |
| Par 2,2:        | 2+2+2*2*2 vale 12 |
| Par 2,3:        | 2+3+2*2*3 vale 17 |
| Par 2,4:        | 2+4+2*2*4 vale 22 |
| Par 3,3:        | 3+3+2*3*3 vale 24 |
| Par 3,4:        | 3+4+2*3*4 vale 31 |
| Par 4,4:        | 4+4+2*4*4 vale 40 |

18. Dado un entero  $n > 0$  se desea un método estático que calcule otro entero con las cifras de  $n$  en orden inverso. Por ejemplo, si  $n = 5437$  debe calcular 7345, si  $n = 7$  debe calcular 7.
19. Escribir un método estático que dados  $a$  y  $n$  enteros no negativos, calcule  $a \cdot n$  utilizando únicamente sumas, productos y divisiones por dos. Para ello, se deberá aplicar repetidamente la siguiente propiedad:  $a \cdot n = (a \cdot 2) \cdot n/2$ , si  $n$  es par, ó  $a \cdot n = (a \cdot 2) \cdot n/2 + a$ , si  $n$  es impar. Por ejemplo:  $4 \cdot 14 = 8 \cdot 7 = 16 \cdot 3 + \underline{8} = 32 \cdot 1 + \underline{16} + \underline{8} = 32 + 16 + 8$ . Se sugiere utilizar una variable  $result$ , que vaya acumulando las cálculos parciales que aparecen subrayados en el ejemplo anterior.

20. Escribir un programa que se comporte como un reloj digital, mostrando en pantalla el siguiente mensaje:

| Salida Estándar |      |         |          |
|-----------------|------|---------|----------|
| dia             | hora | minutos | segundos |

El programa pedirá al usuario que introduzca el día de la semana y la hora, por ejemplo:

Sábado 23 59 58

Una vez comprobado que los datos son correctos, el programa mostrará por pantalla el día y la hora correcta (hora, minutos y segundos) para los 100 segundos siguientes. Para el ejemplo:

| Salida Estándar |    |    |    |
|-----------------|----|----|----|
| Sábado          | 23 | 59 | 59 |
| Domingo         | 0  | 0  | 0  |
| Domingo         | 0  | 0  | 1  |
| ...             |    |    |    |
| Domingo         | 0  | 1  | 38 |

21. Escribir un método estático que dados dos caracteres, muestre por pantalla los caracteres que hay entre los dos valores, incluyendo los extremos. Por ejemplo, dados los caracteres ‘a’ y ‘e’, los caracteres que se muestran por pantalla son ‘a’, ‘b’, ‘c’, ‘d’ y ‘e’.
22. Escribir un método estático que muestre por pantalla una figura de  $n$  líneas, escribiendo los caracteres blanco y asterisco. La primera línea debe ser toda de blancos y la última toda de asteriscos; cada línea debe tener dos blancos menos que la anterior, como en el siguiente ejemplo con  $n = 6$ .

| Salida Estándar |  |  |  |
|-----------------|--|--|--|
| **              |  |  |  |
| ***             |  |  |  |
| *****           |  |  |  |
| *****           |  |  |  |
| *****           |  |  |  |

23. Implementar un método estático que dada una `String` escriba en la salida estándar, línea a línea, la `String` y todas sus variantes, consistentes en ir desplazando sus caracteres una posición a la izquierda, y llevando el primer carácter al final. Por ejemplo, para Java se deberán escribir las siguientes variantes:

| Salida Estándar |
|-----------------|
| Java            |
| avaJ            |
| vaJa            |
| aJav            |

24. Para calcular una aproximación del valor de  $e^x$ , se puede utilizar el desarrollo en serie:  $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$ . La estrategia consiste en generar sucesivamente cada uno de los términos del desarrollo, acumulándolos en una variable a medida que se generan, hasta el momento en que el último término desarrollado tenga un valor suficientemente pequeño. Cabe notar que, además, es posible calcular cada término en función del inmediatamente anterior (excepto en el caso del primero que vale siempre 1).

Se pide: Realizar un método estático que reciba como parámetros el valor  $x$  y un  $\epsilon$  ( $\epsilon > 0$ ), y calcule utilizando la estrategia descrita, el valor de  $e^x$ , sumando todos los términos generados hasta que el último calculado sea menor que  $\epsilon$ . Utilizar el método para calcular el valor del número  $e$  (se recomienda un  $\epsilon$  muy pequeño, del orden de la precisión de `double`) y compararlo con la constante `Math.E` de Java.

## Más información

[Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.

URL: <http://math.hws.edu/javanotes/>. Capítulo 3 (3.1, 3.3 y 3.4).

[GG05] D. Gries and P. Gries. *Multimedia Introduction to Programming Using Java*. Springer, 2005. Capítulo 7.

[Ora11d] Oracle. *The Java<sup>TM</sup> Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Language Basics - Control Flow Statements.

[Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010. Capítulo 3.

## Capítulo 10

# Arrays: definición y aplicaciones

En ocasiones es conveniente poder almacenar y referenciar variables que representan una colección de valores, de forma que sea posible tratarlos de manera uniforme. Por ejemplo, imagínese que una estación meteorológica dispone de las medidas diarias de la temperatura media en una determinada zona geográfica a lo largo de un mes. A partir de dichas medidas la estación debe generar un informe indicando los días de máxima y mínima temperatura a lo largo de todo el mes y sus valores, la temperatura media del mes, los días en que la temperatura duplicó al valor medio y, por último, un listado de las distintas medidas y los días en que éstas se produjeron ordenado ascendentemente por el valor medido.

Para poder efectuar cada uno de los procesamientos indicados, es necesario poder tratar de forma individual cada una de las medidas; pero haciendo uso de algún tipo de notación que evite el problema de tener que escribir repetidamente los mismos cálculos con nombres de variables prácticamente idénticos.

Las matemáticas resuelven el problema anterior con el uso de índices. Así, si se considerara la variable *tempMed*, por ejemplo, como la colección de valores de la temperatura media diaria a lo largo del mes de octubre, entonces se podría referenciar cada uno de los valores individuales mediante un subíndice que indicara el día en que se produce dicha medida, así:  $tempMed_1, tempMed_2, \dots, tempMed_{20}$ , harían referencia, respectivamente, a la temperatura media registrada el primer, segundo y vigésimo día del mes. Así,  $tempMed_i$ , representaría la temperatura media registrada el día  $i$ -ésimo, siendo  $i$  un valor entero que indica el número de día.

En el lenguaje Java (así como en la mayoría de los lenguajes de programación) se introduce el concepto de *array*<sup>1</sup> o colección de valores del mismo tipo, con nombre común, y referenciables uno a uno, mediante un índice. En el resto del capítulo se examinará cómo definir y utilizar adecuadamente arrays en Java.

## 10.1 Arrays unidimensionales

Como ya se ha dicho, un *array* es una colección de componentes homogénea, esto es, todos sus componentes son del mismo tipo de datos, de un tamaño predefinido, con un nombre o identificador común y cuyo número no es modificable. Las características principales de una variable array son:

- es posible acceder y modificar cada una de las componentes individuales por su posición dentro del grupo o colección en tiempo constante,
- el número de componentes de un array se establece inicialmente, no siendo posible su modificación posterior, salvo volviendo a construir el array de nuevo.

### 10.1.1 Declaración y creación. Atributo length

Mediante la declaración de un array se define, en primer lugar, el tipo de cada una de sus componentes de la forma:

```
tipoBase[] elArray;
```

donde `tipoBase` es cualquier tipo de datos, ya sea elemental, referencia y, en particular, otro array como se verá en la sección 10.2. Por ejemplo:

```
double[] tempMed;
String[] texto;
Punto[] camino;
```

Como sucede con el resto de las referencias, cuando se declaran como variables de instancia o de clase los arrays se inicializan a `null`, mientras que quedan sin inicializar si se definen como variables locales en un método.

También es válido utilizar los corchetes después del nombre de variable, de la siguiente manera: `tipoBase elArray[]`. Sin embargo, la propia documentación

---

<sup>1</sup>Cabe señalar que el término inglés *array*, que se utiliza a lo largo del tema, se traduce ocasionalmente como “vector” y también como “arreglo”. Aquí se ha preferido mantener el término inglés original ya que el lenguaje Java utiliza el término “Vector” para referenciar a una clase predeterminada especial, mientras que el término arreglo prácticamente no se utiliza.

de Java recomienda *evitar esta sintaxis*, pues lo habitual al declarar una variable es poner el tipo (en este caso, `tipoBase[]`) y luego el identificador de la variable (`elArray`).

Para establecer el número de elementos del array hace falta, en el lenguaje Java, una operación explícita que establezca inicialmente dicho número, lo que se consigue haciendo uso del operador de construcción, `new`, que permite asignar espacio inicialmente al array. El operador `new` recibe el número (`num`, un entero no negativo) y tipo (`tipoBase`) de los elementos del array, de la siguiente manera:

```
elArray = new tipoBase[num];
```

Por ejemplo, mediante las declaraciones e inicializaciones siguientes, se definen y crean las variables de tipo array de `double`, de `String` y de `Punto` con 31, 100 y 20 componentes respectivamente:

```
double[] tempMed;
tempMed = new double[31];
String[] texto;
texto = new String[100];
Punto[] camino;
camino = new Punto[20];
```

Como se ha dicho, entre las dos instrucciones mostradas (la de declaración y la de creación de espacio inicial) el valor de la variable no está definido y, como es habitual en el lenguaje, es posible aunar ambas instrucciones en tan solo una:

```
double[] tempMed = new double[31];
String[] texto = new String[100];
Punto[] camino = new Punto[20];
```

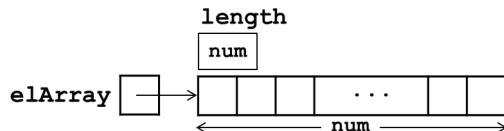
Una última característica significativa que cabe reseñar es que el valor `num`, mediante el cual se define el número de elementos del array en consideración, es una expresión numérica entera no negativa. Con ello es posible parametrizar el tamaño de los arrays en función de alguna característica del problema. Así, por ejemplo, se podría efectuar la siguiente declaración:

```
int numSemanas = 4;
double[] tempMed = new double[numSemanas*7+3];
```

Es muy habitual definir constantes previamente para luego utilizarlas como expresión del número de componentes como por ejemplo:

```
static final int NUM_DIAS=31;
static final int NUM_LINEAS=100;
static final int NUM_PUNTOS=20;
...
double[] tempMed = new double[NUM_DIAS];
String[] texto = new String[NUM_LINEAS];
Punto[] camino = new Punto[NUM_PUNTOS];
```

Ante una declaración y creación de un array, el compilador reserva en el montículo tantos bloques de memoria como sean necesarios para albergar de forma consecutiva `num` componentes de tipo `tipoBase`. Además, se reserva también espacio para un atributo denominado `length`, predefinido en cualquier array, público, final y de tipo entero, que indica el número de componentes del mismo. Así, una vez creado el array se puede consultar el número de componentes mediante la expresión `elArray.length`. En la figura 10.1 se representa gráficamente la reserva de memoria realizada al crear un array. En figuras sucesivas se obviará la representación del atributo `length`.



**Figura 10.1:** Reserva de memoria en un array.

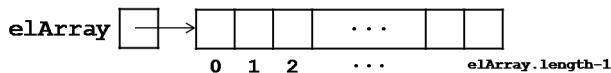
Nótese que, al igual que ocurre con cualquier variable, el comportamiento de la reserva de memoria es distinto si se trata de tipos elementales o de tipos referencia. Si se crea un array de `num` componentes cuyo `tipoBase` es un tipo elemental se reserva memoria para almacenar `num` valores de ese tipo elemental, sin embargo, si el `tipoBase` es un tipo referencia se reserva memoria para `num` referencias, con lo que en realidad aún no se ha reservado memoria para almacenar ningún objeto del `tipoBase`. Por ejemplo, con la declaración y creación del array `camino` de objetos `Punto` se reserva memoria para `NUM_PUNTOS` referencias pero no se crea ningún objeto `Punto`. Habría que crearlos uno a uno, como se hace en el método `main` de la clase de la figura 10.6.

### 10.1.2 Acceso a las componentes

Cada una de las componentes del array se indexan con valores entre 0 y `elArray.length-1`, ambos incluidos, con la siguiente sintaxis y representado gráficamente como se muestra en la figura 10.2:

```
// 0 <= expInt < elArray.length
tipoBase componente = elArray[expInt];
```

siendo `expInt` una expresión numérica entera que se evalúe entre 0 y la longitud del array menos uno. De no ser así, ocurrirá un error durante la compilación en el caso de que no sea un entero o, si el valor está fuera del rango válido, se lanzará una excepción `ArrayIndexOutOfBoundsException` durante la ejecución del programa.



**Figura 10.2:** Índices de las componentes de un array.

Por ejemplo, para mostrar por pantalla la primera y última líneas del array `texto` haríamos:

```
System.out.println(texto[0]);
System.out.println(texto[texto.length-1]);
```

Y si lo que se desea es mostrarlas todas se puede usar una estructura de repetición como, por ejemplo, mediante la instrucción `for` que sigue:

```
for (int i=0; i<texto.length; i++) System.out.println(texto[i]);
```

En ocasiones, por comodidad del programador y claridad de lectura del programa, la posición 0 del array se considera ficticia y aunque esté disponible no se usa. Por ejemplo, si definiésemos las mediciones de temperatura de octubre podría hacerse `double[] tempMed = new double[32]`; si bien `tempMed.length` es igual a 32, en realidad se usarían las componentes desde `tempMed[1]` hasta `tempMed[31]`.

Con cada una de las componentes del array `elArray` es posible efectuar todas las operaciones que podrían realizarse con variables individuales de tipo `tipoBase`. Por ejemplo, dadas las declaraciones anteriores se podría hacer:

```
tempMed[3] = 35.167;
System.out.println("temperatura media: " + tempMed[3]);
tempMed[5] = tempMed[3]*1.2;
int i = 6;
tempMed[i+1] = tempMed[i-1];
```

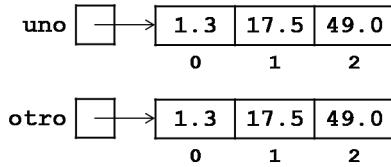
```
camino[0] = new Punto(0.0,0.0);
camino[1] = new Punto(5.0,2.0);
System.out.print("Distancia entre los dos primeros puntos: ");
System.out.printf("%10.2f",camino[0].distancia(camino[1]));

texto[0] = "No por mucho madrugar, amanece más temprano";
System.out.println(texto[0].toUpperCase() +
 " y su longitud es " + texto[0].length());
```

Nótese la diferencia entre `texto.length` y `texto[0].length()`. El primero es el atributo `length` del array `texto` mientras que el segundo es la llamada al método `length()` de la clase `String`.

En Java existe, además, la posibilidad de inicializar simultáneamente todas las componentes de un array, en lugar de hacerlo individualmente componente a componente, utilizando un *inicializador de array*, es decir, una lista de valores o expresiones separadas por comas y encerradas entre llaves, tomando de dicha lista tanto la longitud del array como el contenido de cada componente. Por ejemplo, tras la ejecución del segmento de código siguiente, los arrays `uno` y `otro` contienen los mismos valores, como se muestra en la figura 10.3:

```
int i = 7;
double[] uno = new double[3];
uno[0] = 1.3; uno[1] = 2.5*i; uno[2] = 0.0+i*i;
double[] otro = {1.3, 2.5*i, 0.0+i*i};
```



**Figura 10.3:** Inicialización de arrays.

### 10.1.3 Uso

Las variables de tipo array se pueden usar del mismo modo y en los mismos lugares que el resto de variables, esto es, como variables de instancia o de clase, como parámetros o resultado de un método o como variables locales. Nótese que, como ocurre con el resto de variables de tipo referencia, la asignación y comparación de arrays sólo actúa sobre las propias referencias y que, además, se les puede asignar el valor `null`.

Por ejemplo, en la clase `Temperaturas` se define un array como atributo de instancia:

```
public class Temperaturas {
 private double[] tempMed;
 private String mes;
 ...
 public Temperaturas(int numDias, String nom) {
 tempMed = new double[numDias];
 mes = nom;
 }
 public void setTemperaturas() {...}
 ...
}
```

También se puede definir un array como variable local de un método. Por ejemplo, en el método `main` de la clase `PruebaMayusculas` se define el array `texto`:

```
public class PruebaMayusculas {
 static final int NUM_LINEAS=100;
 public static void main(String[] args) {
 String[] texto = new String[NUM_LINEAS];
 ...
 // las 100 líneas del fichero se almacenan en el array
 for (int i=0; i<texto.length; i++)
 System.out.println(texto[i].toUpperCase());
 }
}
```

Nótese que el método `main` tiene como parámetro un array de `String` que recibe los argumentos de la línea de comandos en la invocación del programa. Un uso habitual del atributo `length` es el de determinar, en tiempo de ejecución, el número de argumentos que recibe el `main`; por ejemplo:

```
public static void main(String[] args) {
 if (args.length>0) {
 System.out.println("hay argumentos:");
 for (int i=0; i<args.length; i++)
 System.out.println("argumento " + i + ": " + args[i]);
 }
 else System.out.println("no hay argumentos");
}
```

Los métodos también pueden definir parámetros formales que sean arrays o devolverlos como resultado. Así, por ejemplo, se pueden inicializar los elementos de un array dentro de un método (para lo cual ya debe estar creado) o crearlo e inicializarlo devolviéndolo como resultado, como ocurre en los métodos `inicializa` y `creaInicializa`, respectivamente.

```
public static void inicializa(double[] a) {
 int i = 7;
 a[0] = 1.3; a[1] = 2.5*i; a[2] = 0.0+i*i;
}

public static double[] creaInicializa() {
 double[] a = new double[3];
 int i = 7;
 a[0] = 1.3; a[1] = 2.5*i; a[2] = 0.0+i*i;
 return a;
}
```

Una vez definidos, estos métodos se podrían utilizar como sigue:

```
double[] uno = new double[3];
inicializa(uno);
double[] otro = creaInicializa();
```

Nótese que, en cuanto a la sintaxis de la definición de parámetros formales y reales, el caso de los arrays es idéntico al del resto de tipos. Para utilizar un array como argumento en la llamada a un método se utiliza el nombre de la variable. Y como ocurre con el resto de tipos en la definición formal del parámetro se indica el tipo que en el caso del array es **tipoBase[]**.

Recuérdese que se aplican las mismas reglas del paso de parámetros que las que se utilizan en el paso de tipos referencia (véase el capítulo 5). Así, el paso de parámetros se realiza siempre *por valor*. Los parámetros formales son variables locales al método, que se inicializan con los valores que tienen los argumentos cuando se produce la llamada al método. Por esta razón no sería correcto crear el array **a** dentro del método **inicializa** que inicializaba el array en el ejemplo anterior.

En el caso de los tipos simples no hay duda, ya que el dato que se pasa como parámetro es una copia del original. Pero, sin embargo, en el caso de los tipos referencia (como pueden ser los objetos y los arrays) de lo que se tiene copia es de la propia referencia (que no se verá modificada al acabar el método) y, por el contrario, un objeto referenciado por dos referencias iguales es el mismo objeto y, por tanto, al realizar modificaciones en dicho objeto los cambios son permanentes.

En la clase **PasoParam** de la figura 10.4 se ejemplifica una vez más el uso de arrays como argumentos, parámetros formales y resultados de un método. En la figura 10.5 se representa esquemáticamente el estado de la memoria durante la ejecución del **main** de dicha clase. Nótese que aunque las variables referencia array no se modifican, no sucede lo mismo con las componentes que contienen.

Finalmente, los métodos **copiar** y **copiarPunto** de la clase **CopiaArrays** de la figura 10.6 son ejemplos de copia de arrays de tipo simple y de arrays de tipo refe-

```

/**
 * Clase PasoParam: ejemplo del uso de arrays como argumentos,
 * parámetros formales y resultados de un método.
 * @author Libro IIP-PRG
 * @version 2011
 */
public class PasoParam {
 /** Método principal.
 * @param args - String[]
 */
 public static void main(String[] args) {
 double[] elArray = {5.0, 6.4, 3.2, 0.0};
 metodo1(elArray);
 // elArray no ha sido modificado en absoluto
 metodo2(elArray);
 // la primera componente de elArray vale ahora 0.1
 }

 public static void metodo1(double[] copia) { copia=new double[4]; }

 public static void metodo2(double[] copia) { copia[0]=0.1; }
}

```

**Figura 10.4:** Ejemplo de arrays como argumentos, parámetros y resultados de un método.

rencia, respectivamente. Nótese que la copia se realiza componente a componente pero, en el caso del array `camino` de objetos `Punto`, para obtener la componente  $i$ -ésima del array `copia` es necesario crear un nuevo objeto `Punto` a partir de la componente  $i$ -ésima del array original.

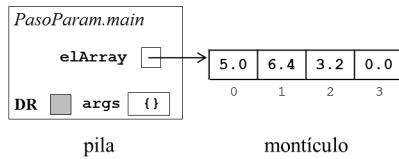
## 10.2 Arrays multidimensionales

Volviendo al ejemplo inicial de la toma de medidas de temperatura en cierta zona geográfica, si se deseara extenderlo para abarcar no sólo los días de un mes sino los de todo un año se podría definir un array de 366 elementos (por si el año es bisiesto) que mantuviera de forma correlativa los datos de temperatura de una zona día a día. Con ello, por ejemplo, el dato correspondiente al día 3 de febrero ocuparía la posición 33 del array (comenzando en la posición 0), mientras que el correspondiente al 2 de julio ocuparía la 183 (o la posición 184 si el año fuera bisiesto). Una representación alternativa consistiría en utilizar una matriz de dimensiones  $12 \times 31$ . Esto permitiría una descripción más ajustada a la realidad y, sobre todo, simplificaría los cálculos de la posición real de cada día en la estructura

```

public static void main(String[] args){
 double elArray = {5.0,6.4,3.2,0.0};
 metodo1(elArray);
 // elArray no se ha modificado
 metodo2(elArray);
 // elArray[0] vale 0.1
}
...

```

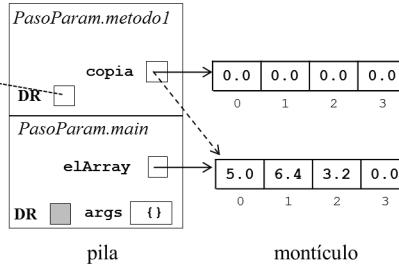


(a) Estado de la memoria antes de la llamada a `metodo1`.

```

public static void main(String[] args){
 double elArray = {5.0,6.4,3.2,0.0};
 metodo1(elArray);
 // elArray no se ha modificado
 metodo2(elArray);
 // elArray[0] vale 0.1
}
static void metodo1(double[] copia) {
 copia = new double[4];
}
...

```

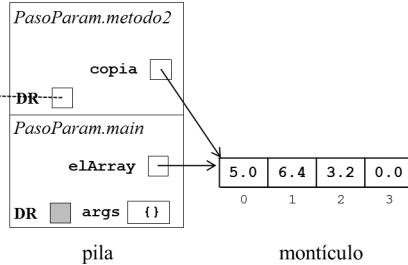


(b) Estado de la memoria justo antes de acabar la ejecución de `metodo1`.

```

public static void main(String[] args){
 double elArray = {5.0,6.4,3.2,0.0};
 metodo1(elArray);
 // elArray no se ha modificado
 metodo2(elArray);
 // elArray[0] vale 0.1
}
...
static void metodo2(double[] copia) {
 copia[0] = 0.1;
}

```

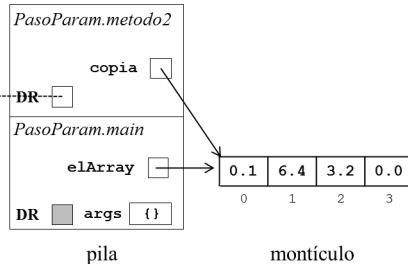


(c) Estado de la memoria recién invocado `metodo2`.

```

public static void main(String[] args){
 double elArray = {5.0,6.4,3.2,0.0};
 metodo1(elArray);
 // elArray no se ha modificado
 metodo2(elArray);
 // elArray[0] vale 0.1
}
...
static void metodo2(double[] copia) {
 copia[0] = 0.1;
}

```



(d) Estado de la memoria justo antes de acabar la ejecución de `metodo2`.

**Figura 10.5:** Estados de la memoria al ejecutar la clase `PasoParam`.

```

/**
 * Clase CopiaArrays: ejemplo de copia de arrays de tipo simple
 * y de arrays de tipo referencia.
 * @author Libro IIP-PRG
 * @version 2011
 */
public class CopiaArrays {
 /** Método principal.
 * @param args - String[]
 */
 public static void main(String[] args) {
 double[] a = {5.0, 6.4, 3.2, 0.0};
 double[] copia = copiar(a);
 // las componentes del array copia son copias de las de a

 Punto[] camino = new Punto[20];
 for(int i=0; i<camino.length; i++)
 camino[i] = new Punto();
 // se han creado los 20 objetos Punto del array camino
 Punto[] copiaP = copiarPunto(camino);
 // las componentes de copiaP son copias de las de camino
 }

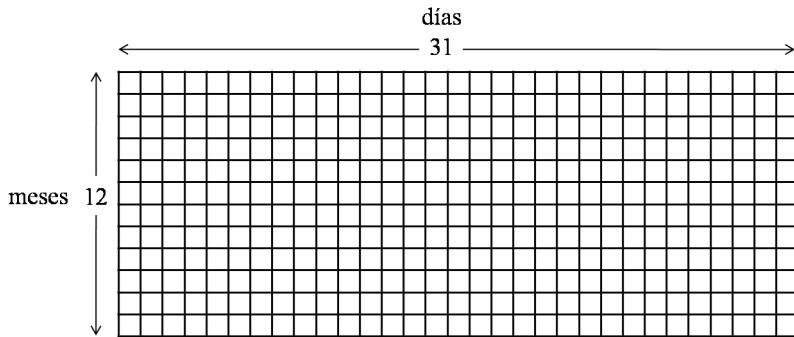
 /** Copia de array de tipo simple. */
 public static double[] copiar(double[] orig) {
 double[] copia = new double[orig.length];
 for (int i=0; i<orig.length; i++)
 copia[i] = orig[i];
 return copia;
 }

 /** Copia de array de tipo referencia. */
 public static Punto[] copiarPunto(Punto[] orig) {
 Punto[] copia = new Punto[orig.length];
 for (int i=0; i<orig.length; i++)
 // si se hace copia[i]=orig[i] se copia la referencia,
 // no el dato de tipo Punto del array.
 // Por eso hay que volver a construir cada componente.
 copia[i] = new Punto(orig[i].getX(),orig[i].getY());
 return copia;
 }
}

```

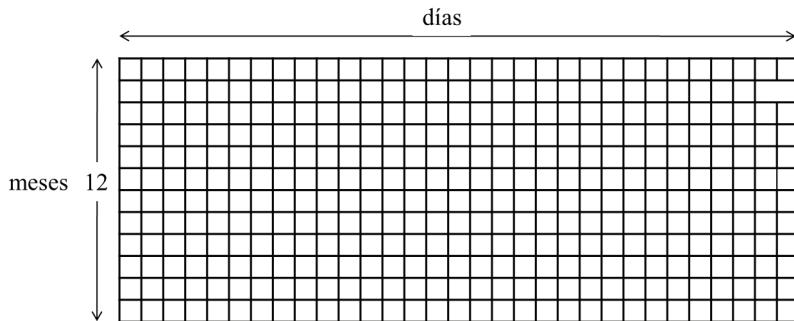
**Figura 10.6:** Ejemplo de copia de arrays de tipo simple y tipo referencia.

de datos. Como puede verse en la figura 10.7, se representaría en Java como un array de 12 componentes que, a su vez, fueran arrays de 31 elementos cada uno.



**Figura 10.7:** Representación de los datos del año. Primera versión.

Aún mejor sería utilizar un array de 12 componentes que, a su vez, fueran arrays de longitud igual al número de días de cada mes, como se muestra en la figura 10.8. Así, el 3 de febrero estaría en la componente 1,2 y el 2 de julio en la 6,1. Aunque mejor sería incluso no usar ni la fila ni la columna 0, con lo que el 3 de febrero estaría en la componente 2,3 y el 2 de julio estaría en la componente 7,2.



**Figura 10.8:** Representación de los datos del año. Segunda versión.

Como se ha comentado en Java las componentes de un array pueden ser, a su vez, otros arrays. Las componentes de estos últimos arrays pueden ser tanto datos elementales, con lo que los arrays son *bidimensionales*, (también denominados *matrices* si tienen todos la misma longitud), o pueden ser arrays de nuevo. En general, no hay límite al anidamiento que puede presentar una estructura de ese tipo, con lo que se puede obtener arrays de tantas dimensiones como se desee (denominados *n-dimensionales* o *multidimensionales*).

### 10.2.1 Declaración y creación

La sintaxis de Java para declarar un array multidimensional sería:

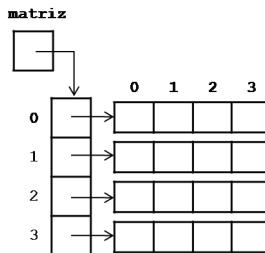
```
tipoBase[][]...[] elArrayMulti;
```

Por ejemplo, el código siguiente declara `m1`, `m2` y `m3` como arrays bidimensionales; sin embargo, aunque las tres sean correctas, se utilizará la primera forma de definición que es la recomendada en el propio tutorial de Java [Ora11d].

```
double[][] m1; // Válido.
double[] m2[]; // Válido, no recomendado.
double m3[][]; // Válido, no recomendado.
```

Para crear un array bidimensional se utiliza el operador `new` (igual que en el caso unidimensional). Por ejemplo, la instrucción siguiente declara y crea en Java un array bidimensional `matriz` de  $4 \times 4$  elementos de tipo `double`, representado en memoria como se muestra en la figura 10.9.

```
double[][] matriz = new double[4][4];
```

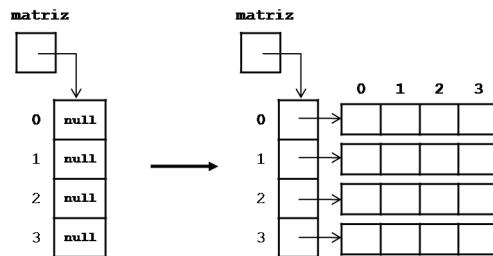


**Figura 10.9:** Representación de una matriz bidimensional.

En realidad los arrays multidimensionales en Java son arrays de arrays y cada array tiene las características de un array de una dimensión menos, por lo que es posible declarar al comienzo tan solo una dimensión e inicializar posteriormente el resto. Se puede considerar, por ejemplo, esta otra forma de declaración para el ejemplo anterior:

```
double[][] matriz = new double[4][];
// se inicializa matriz a un array de 4 componentes,
// arrays, a su vez, aún no inicializados.
matriz[0] = new double[4];
matriz[1] = new double[4];
matriz[2] = new double[4];
matriz[3] = new double[4];
```

Inicialmente se declararía el array **matriz** como un array de cuatro componentes inicializadas a **null**; después, cada una de esas componentes se inicializaría, a su vez, como un array de longitud 4 como puede verse en la figura 10.10.



**Figura 10.10:** Declaración e inicialización en dos fases de una matriz bidimensional.

Como ocurría en la declaración de los arrays unidimensionales, el número de elementos con que se inicializa un array o subarray cualquiera en el caso multidimensional puede ser una expresión numérica entera con valor, en el momento de la creación, mayor o igual que 0, como en el ejemplo siguiente:

```
int alto = 2, ancho = 3, prof = 2, valor = 1;
double[][][] tridimensional;
tridimensional = new double[alto][ancho][prof*(valor+1)];
```

Obviamente, es posible inicializar cada uno de los subarrays con un tamaño diferente (aunque el **tipoBase** debe ser siempre el mismo para todos los componentes) como puede verse en el ejemplo siguiente.

**Ejemplo 10.1.** Se desea almacenar las medidas diarias (de tipo **double**) de la temperatura media en una determinada zona geográfica a lo largo de un año (no bisiesto, por simplificar). Para ello, se puede declarar un array bidimensional con el tipo base **double** donde las filas son los meses y las columnas los días de cada mes.

En concreto, se define un array bidimensional **tempMed** con 12 filas y cada una de ellas tendrá un número de columnas adecuado a los días del mes que representa (entre 28 y 31). La componente 0 del array corresponde al mes de enero y, por tanto, su longitud **tempMed[0].length** tendrá que ser igual a 31 y así sucesivamente. Nótese que se utiliza un array especial **NUM\_DIAS** para almacenar el número total de días de cada mes.

```

final int[] NUM_DIAS = {31,28,31,30,31,30,31,31,30,31,30,31};
// NUM_DIAS[i] = n° días del mes (i+1), 0<=i<=11

double[][] tempMed = new double[12][];
// tempMed[i] representa el mes (i+1), 0<=i<=11

for (int i=0; i<tempMed.length; i++)
 tempMed[i] = new double[NUM_DIAS[i]];
// el n° elementos de tempMed[i] = NUM_DIAS[i], 0<=i<=11

```

Así, `tempMed[i][j]` representará la temperatura media del día (`j+1`) del mes (`i+1`), para  $0 \leq i \leq 11$  y  $0 \leq j < \text{NUM\_DIAS}[i]$ .

Si para facilitar la legibilidad del código se deseara que los índices del array se correspondieran con el número de mes y el número de día, la declaración y creación del array sería:

```

final int[] NUM_DIAS = {0,31,28,31,30,31,30,31,31,30,31,30,31};
// NUM_DIAS[0] = 0 y NUM_DIAS[i] = n° días del mes i, 1<=i<=12

double[][] tempMed = new double[13][];
// tempMed[0] = null y tempMed[i] representa el mes i, 1<=i<=12

for (int i=1; i<tempMed.length; i++)
 tempMed[i] = new double[NUM_DIAS[i]+1];
// el n° elementos de tempMed[i] = NUM_DIAS[i]+1, 1<=i<=12

```

Nótese que todas las longitudes del array se han incrementado en uno. De este modo, `tempMed[i][0]=0` y `tempMed[i][j]` representará la temperatura media del día `j` del mes `i`, para  $1 \leq i \leq 12$  y  $1 \leq j \leq \text{NUM\_DIAS}[i]$ .

### 10.2.2 Acceso a las componentes

El acceso a las componentes de un array multidimensional se efectúa de forma similar al caso unidimensional, pero tomando en consideración la existencia de múltiples dimensiones. Por ejemplo, siendo `elArrayBi` un array bidimensional:

```
elArrayBi[exp1][exp2]
```

donde `exp1` es una expresión entera que se tiene que evaluar a un valor comprendido entre 0 y `elArrayBi.length` y, a su vez, `exp2` es también una expresión entera que se tiene que evaluar a un valor comprendido entre 0 y `elArrayBi[exp1].length`. La expresión `elArrayBi[exp1][exp2]` se tratará como una variable del tipoBase con el que se haya declarado el array.

Por ejemplo, si se declara una matriz de dimensiones  $4 \times 4$  y se quiere almacenar una serie de valores enteros, se puede hacer:

```
int[][] matriz = new int[4][4];
matriz[0][0]=1; matriz[0][1]=4; matriz[0][2]=2; matriz[0][3]=3;
matriz[1][0]=3; matriz[1][1]=1; matriz[1][2]=4; matriz[1][3]=2;
matriz[2][0]=2; matriz[2][1]=3; matriz[2][2]=1; matriz[2][3]=4;
matriz[3][0]=4; matriz[3][1]=2; matriz[3][2]=3; matriz[3][3]=1;
```

Como ocurría con los arrays unidimensionales, a los n-dimensionales se les puede asignar también un valor mediante el uso de un *inicializador de array*, como, por ejemplo, se realiza a continuación:

```
// definición e inicialización de matriz
int[][] matriz = {{1,4,2,3},{3,1,4,2},{2,3,1,4},{4,2,3,1}}
```

En ambos casos, se podría representar el contenido de `matriz` como se ve en la figura 10.11.

|   |   | matriz |   |   |   |
|---|---|--------|---|---|---|
|   |   | 0      | 1 | 2 | 3 |
| 0 | 0 | 1      | 4 | 2 | 3 |
|   | 1 | 3      | 1 | 4 | 2 |
| 2 | 2 | 3      | 1 | 4 |   |
| 3 | 4 | 2      | 3 | 1 |   |

Figura 10.11: Representación habitual de una matriz bidimensional.

Por ejemplo, para mostrar por pantalla un valor o una fila o toda la matriz se haría del modo siguiente:

```
System.out.println("En la 2a fila, 4a columna de matriz" +
 " está almacenado el valor: " + matriz[1][3]);

System.out.println("Los datos de la segunda fila son:");
for (int c=0; c<matriz[1].length; c++)
 System.out.printf("%3d, ",matriz[1][c]);
System.out.println();

System.out.println("Los datos de la toda la matriz son:");
for (int f=0; f<matriz.length; f++) {
 for (int c=0; c<matriz[f].length; c++)
 System.out.printf("%3d, ",matriz[f][c]);
 System.out.println();
}
```

## 10.3 Tratamiento secuencial y directo de un array

Por lo general se consideran dos tipos de tratamiento de los elementos de una secuencia de datos, los denominados *tratamiento secuencial* y *directo*. Esto se puede particularizar para el caso en el que se utilice un array para representar dicha secuencia de datos.

En el *tratamiento secuencial* se accede a los elementos de la estructura (o de una parte de ella) posicionalmente, uno detrás del otro. Los problemas de recorrido y búsqueda secuencial son ejemplos de problemas que se resuelven con este tipo de tratamiento.

En el *tratamiento directo* se accede a los elementos por su posición, independientemente de qué se haya podido hacer antes. Ejemplos de problemas que se resuelven con este tipo de tratamiento son los que usan la estructura como un conjunto de contadores, como referencias posicionales, la búsqueda binaria, los algoritmos de ordenación, etc.

Como los arrays son estructuras de acceso directo, cualquiera de los dos tipos de tratamiento es posible.

### 10.3.1 Acceso secuencial: recorrido y búsqueda

Muchos de los problemas que se plantean cuando se utilizan arrays pueden clasificarse en dos grandes grupos de problemas genéricos: los que conllevan el *recorrido de un array* y los que suponen la *búsqueda de un elemento*, de entre los del array, que cumple cierta característica dada. La importancia de este tipo de problemas proviene de que surgen no sólo en el ámbito de los arrays sino también en muchas otras organizaciones de datos de uso frecuente. Las estrategias básicas de resolución que se verán a continuación son también extrapolables a esos otros ámbitos.

#### *Esquemas de recorrido*

Se clasifican como *problemas de recorrido* todos aquellos que para su resolución exigen algún tratamiento de todos o de un número previamente determinado de los elementos del array.

Se entiende por *recorrido de un array* a desde una posición `inicio` hasta una posición `fin`,  $0 \leq \text{inicio} \leq \text{fin} < \text{a.length}$ , a la visita única y sistemática de los elementos de `a` comprendidos entre esas dos posiciones. Este recorrido puede realizarse secuencialmente de manera ascendente, descendente o de forma combinada. Para implementar este acceso secuencial se utiliza un índice que permite acceder a las componentes del array.

El esquema general de un *recorrido ascendente* de un array **a** desde la posición **inicio** hasta la posición **fin**, usando un bucle **while**, se muestra a continuación:

```
int i = inicio;
while (i<=fin) {
 tratar(a[i]);
 avanzar(i);
}
```

donde **tratar(a[i])** indica la operación a realizar con el elemento *i*-ésimo del array y **avanzar(i)** representa el incremento del índice. Nótese que la inicialización, la guarda y el avance del bucle deben asegurar que cuando se accede a un elemento del array, el índice siempre está dentro del rango de índices del array.

El siguiente es el esquema equivalente usando un bucle **for**:

```
for (int i=inicio; i<=fin; avanzar(i))
 tratar(a[i]);
```

El esquema general de un *recorrido descendente* de un array **a** desde la posición **fin** hasta la posición **inicio**, usando un bucle **while**, se muestra a continuación:

```
int j = fin;
while (j>=inicio) {
 tratar(a[j]);
 retroceder(j);
}
```

donde **tratar(a[j])** indica la operación a realizar con el elemento *j*-ésimo del array y **retroceder(j)** representa el decremento del índice.

El siguiente es el esquema equivalente usando un bucle **for**:

```
for (int j=fin; j>=inicio; retroceder(j))
 tratar(a[j]);
```

En las figuras 10.12(a) y 10.12(b) se muestra gráficamente el estado de las variables del bucle a lo largo de la ejecución en un recorrido ascendente y descendente, respectivamente. Se han sombreado en gris los elementos del array **a** que ya han sido visitados. Nótese que inicialmente ninguno de los elementos está sombreado mientras que, al finalizar el recorrido, todos los elementos del array desde la posición **inicio** a la posición **fin** están sombreados, indicando que todos ellos se han tratado.

```

int i = inicio;
 i
a [] -> [] [] [] ... [] [] []
 inicio ... fin a.length-1

while (i <= fin) {
 tratar(a[i]);
 avanzar(i);
 i
a [] -> [] [] [] [] ... [] [] []
 inicio ... fin a.length-1
}

a [] -> [] [] [] [] ... [] [] []
 inicio ... fin a.length-1

```

(a) Recorrido ascendente.

```

int j = fin;
 j
a [] -> [] [] [] ... [] [] []
 inicio ... fin a.length-1

while (j >= inicio) {
 tratar(a[j]);
 retroceder(j);
 j
a [] -> [] [] [] ... [] [] [] []
 inicio ... fin a.length-1
}

a [] -> [] [] [] [] ... [] [] []
 inicio ... fin a.length-1

```

(b) Recorrido descendente.

**Figura 10.12:** Estado de las variables del bucle durante la ejecución de un recorrido.

Por ejemplo, suponiendo que el array `Punto[] camino` contiene, en posiciones sucesivas, la secuencia de puntos desde un cierto origen a un cierto destino, el recorrido de origen a destino se puede realizar como sigue:

```
for (int i=0; i<=camino.length-1; i++)
 System.out.println(camino[i]);
```

Y el recorrido inverso se realizará del siguiente modo:

```
for (int j=camino.length-1; i>=0; j--)
 System.out.println(camino[j]);
```

Véase que se trata de una instanciación del esquema de recorrido en el que se tiene `inicio = 0` y `fin = camino.length-1`.

El recorrido de arrays se utiliza para resolver problemas en los que necesariamente se deben conocer todos los datos para poder determinar la solución; por ejemplo, obtener el máximo o el mínimo de un conjunto de números, obtener la suma o el producto de todos los números de un conjunto dado, obtener la media, etc. A continuación, se presentan algunos de estos ejemplos.

**Ejemplo 10.2.** Los métodos siguientes devuelven la media aritmética de un array `a` de enteros (no vacío) o del subarray comprendido entre las posiciones `izq` y `der` ( $0 \leq izq \leq der < a.length$ ).

```
/** Cálculo de la media aritmética de a, array no vacío de int. */
public static double media(int[] a) {
 double suma = 0;
 for (int i=0; i<a.length; i++) suma+=a[i];
 return suma/a.length;
 // o directamente invocando al método media(int[],int,int):
 // return media(a,0,a.length-1);
}

/** Cálculo de la media aritmética del subarray de int
 * a[izq...der], 0<=izq<=der< a.length. */
public static double media(int[] a, int izq, int der) {
 double suma = 0;
 for (int i=izq; i<=der; i++) suma+=a[i];
 return suma/(der-izq+1);
}
```

**Ejemplo 10.3.** Los siguientes métodos permiten obtener la posición del valor máximo de un array de `double` y de `String`. Cabe señalar que la única diferencia es la forma de comparar los datos de dichos arrays.

```
/** Cálculo de la posición del máximo del array a. */
public static int maximo(double[] a) {
 int posMax = 0;
 // el recorrido se inicia en 1, ya que
 // la posición 0 ya ha sido tratada
 for (int i=1; i<a.length; i++)
 if (a[i]>a[posMax]) posMax = i;
 return posMax;
}

/** Cálculo de la posición del máximo del array a. */
public static int maximo(String[] a) {
 int posMax = 0;
 // el recorrido se inicia en 1, ya que
 // la posición 0 ya ha sido tratada
 for (int i=1; i<a.length; i++)
 if (a[i].compareTo(a[posMax])>0) posMax = i;
 return posMax;
}
```

Nótese que si el valor máximo del array está repetido, el método `maximo` devuelve la posición del máximo de menor índice. Para obtener la de índice mayor se podría o bien modificar la condición de la instrucción condicional usando  $\geq$  o bien realizar un recorrido descendente como sigue:

```
/** Cálculo de la posición del máximo del array a. */
public static int maximo(double[] a) {
 int posMax = a.length-1;
 // el recorrido se inicia en a.length-2, ya que
 // la posición a.length-1 ya ha sido tratada
 for (int i=a.length-2; i>=0; i--)
 if (a[i]>a[posMax]) posMax = i;
 return posMax;
}

/** Cálculo de la posición del máximo del array a. */
public static int maximo(String[] a) {
 int posMax = a.length-1;
 // el recorrido se inicia en a.length-2, ya que
 // la posición a.length-1 ya ha sido tratada
 for (int i=a.length-2; i>=0; i--)
 if (a[i].compareTo(a[posMax])>0) posMax = i;
 return posMax;
}
```

**Ejemplo 10.4.** Los siguientes son problemas que requieren el recorrido de arrays multidimensionales. El método `suma` realiza la suma de matrices de la misma dimensión y el método `matrizPorVect` realiza el producto de una matriz por un vector.

```
/** Cálculo de la suma de dos matrices de reales de la misma
 * dimensión. */
public static double[][] suma(double[][] m1, double[][] m2) {
 double[][] res = new double[m1.length][m1[0].length];
 for (int i=0; i<m1.length; i++)
 for (int j=0; j<m1[i].length; j++)
 res[i][j] = m1[i][j] + m2[i][j];
 return res;
}

/** Cálculo del producto de matriz por vector de reales.
 * El número de columnas de la matriz debe coincidir con
 * el tamaño del vector. */
public static double[] matrizPorVect(double[][] m, double[] v) {
 double[] res = new double[m.length];
 for (int i=0; i<m.length; i++)
 for (int j=0; j<v.length; j++)
 res[i] += m[i][j] * v[j];
 return res;
}
```

### *Esquemas de búsqueda*

Se denominan *problemas de búsqueda* a los que requieren determinar si existe algún elemento del array que cumpla una propiedad dada. Con respecto a los problemas de recorrido presentan la diferencia de que no es siempre necesario visitar todos los elementos del array, ya que el elemento buscado puede encontrarse inmediatamente, encontrarse tras haber recorrido todo el array, o incluso no encontrarse.

Este tipo de problemas implica operaciones que pueden obtener la solución sin necesidad de conocer todos los datos: localizar el primero que cumpla cierto requisito, tratar todos los datos hasta que se cumpla cierta condición, etc.

En Java, las búsquedas en arrays se concretan mediante el uso de variables enteras para acceder a sus distintas posiciones, de forma similar a los recorridos, y con variables de tipo `boolean` para comprobar la condición de terminación.

El esquema general de una *búsqueda ascendente* de un elemento que cumpla una cierta propiedad en un array `a` desde la posición `inicio` hasta la posición `fin`,  $0 \leq \text{inicio} \leq \text{fin} < \text{a.length}$ , usando un bucle `while`, se muestra a continuación:

```
int i = inicio, j = fin;
boolean encontrado = false;
while (i <= j && !encontrado) {
 if (propiedad(a[i])) encontrado = true;
 else avanzar(i);
}
// Resolución de la búsqueda
if (encontrado) ... // a[i] cumple la propiedad
else ... // ningún elemento cumple la propiedad
```

donde `propiedad(a[i])` comprueba si el elemento `i`-ésimo del array cumple la propiedad enunciada y `avanzar(i)` representa el incremento del índice.

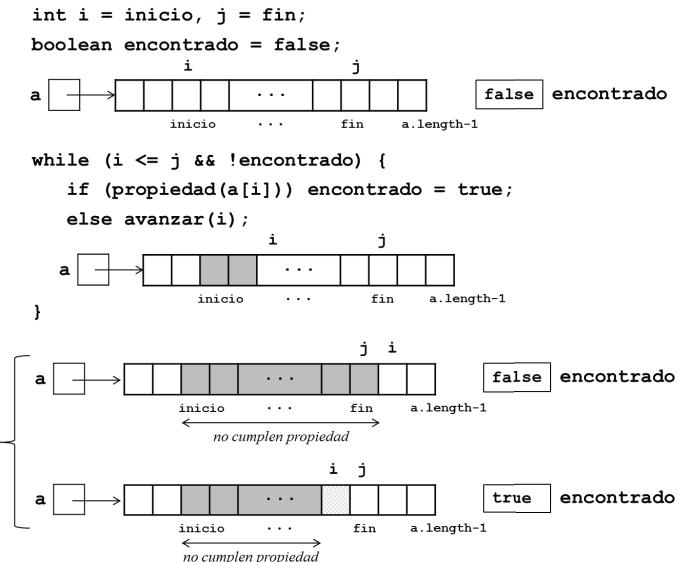
El esquema general de una *búsqueda descendente* de un elemento que cumpla una cierta propiedad en un array `a` desde la posición `fin` hasta la posición `inicio`, usando un bucle `while`, se muestra a continuación:

```
int i = inicio, j = fin;
boolean encontrado = false;
while (j >= i && !encontrado) {
 if (propiedad(a[j])) encontrado = true;
 else retroceder(j);
}
// Resolución de la búsqueda
if (encontrado) ... // a[j] cumple la propiedad
else ... // ningún elemento cumple la propiedad
```

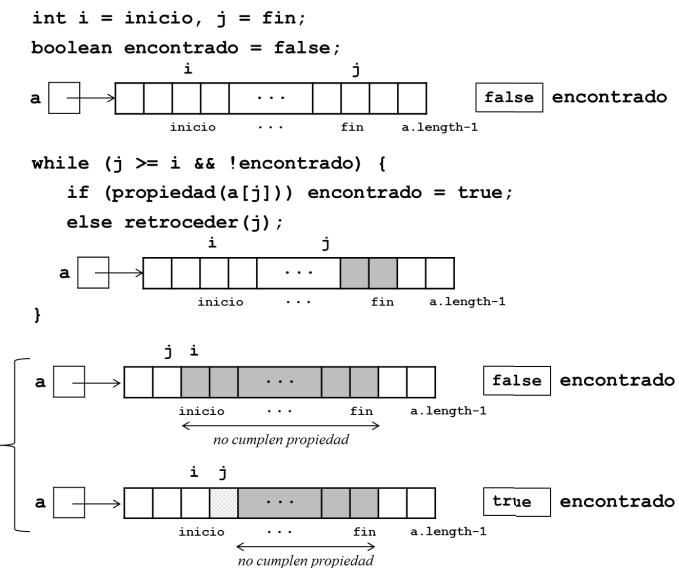
donde `propiedad(a[j])` comprueba si el elemento `j`-ésimo del array cumple la propiedad enunciada y `retroceder(j)` representa el decremento del índice.

En las figuras 10.13(a) y 10.13(b) se muestra gráficamente el estado de las variables del bucle a lo largo de la ejecución en una búsqueda ascendente y descendente, respectivamente. El sombreado en gris de algunos de los elementos del array `a` indica que ya se han visitado y no cumplen la propiedad a comprobar. Después del bucle, se representan las dos situaciones por las que una búsqueda puede acabar:

- *Búsqueda sin éxito:* ninguno de los elementos del array desde la posición `inicio` a la posición `fin` cumple la propiedad y `encontrado=false`.
- *Búsqueda con éxito:* el elemento `a[i]` (`a[j]` en la búsqueda descendente) cumple la propiedad y `encontrado=true`.



(a) Búsqueda ascendente.



(b) Búsqueda descendente.

Figura 10.13: Estado de las variables del bucle durante la ejecución de una búsqueda.

Una variante del esquema de búsqueda anterior es el siguiente que no utiliza la variable `encontrado` de tipo `boolean`. Nótese que en este caso es importante el orden de las dos comparaciones de la guarda del bucle y el uso del operador `&&` (*and* cortocircuitado).

```
int i = inicio, j = fin;
while (i<=j && !propiedad(a[i])) avanzar(i);
// El bucle acaba porque:
// i<=fin y a[i] cumple propiedad o i=fin+1

// Resolución de la búsqueda
if (i<=j) ... // a[i] cumple la propiedad
else ... // ningún elemento cumple la propiedad
```

Un esquema alternativo consiste en salir del bucle cuando se cumple la propiedad, acabando el código en ejecución. Por ejemplo, suponiendo que se devuelve la posición del elemento que cumple la propiedad o se devuelve -1 si ningún elemento la cumple, el esquema sería:

```
int i = inicio, j = fin;
while (i<=j) {
 if (propiedad(a[i])) return i; // búsqueda con éxito
 avanzar(i);
}
// i>j y no hay ningún elemento que cumpla la propiedad
return -1;
```

Si se sabe de antemano que existe algún elemento `a[i]` que cumple la propiedad, la guarda del bucle se puede simplificar. Se trata de una búsqueda denominada *búsqueda con garantía de éxito*:

```
int i = inicio;
while (!propiedad(a[i])) avanzar(i);

// Resolución de la búsqueda:
// a[i] cumple la propiedad
```

Es posible incluir deliberadamente un elemento que cumpla la propiedad. Dicho elemento se denomina *centinela*. En ese caso, la búsqueda se llama *búsqueda con centinela*.

Los siguientes son ejemplos de problemas en los que, para su resolución, se aplica alguno de los esquemas de búsqueda anteriores.

**Ejemplo 10.5.** Los siguientes métodos de clase devuelven la posición del dato en el array si se encuentra uno igual o devuelven -1 si no se encuentra. En el caso de que hayan valores repetidos, al tratarse de una búsqueda ascendente, el primero devuelve la posición de índice menor mientras que el segundo, por ser una búsqueda descendente, devuelve la posición de índice mayor.

```
/** Busca dato en un array de String SIN garantía de éxito.
 * Si el dato se encuentra devuelve su índice y si no
 * devuelve -1. */
public static int buscarPos(String[] a, String dato) {
 int i = 0;
 while (i<a.length && !a[i].equals(dato)) i++;
 if (i<a.length) return i;
 else return -1; // o bien directamente: return -1;
}

/** Busca un dato en un array de double SIN garantía de éxito.
 * Si el dato se encuentra devuelve su índice y si no
 * devuelve -1. */
public static int buscarPos(double[] a, double dato) {
 int i = a.length-1;
 while (i>=0 && a[i]!=dato) i--;
 return i;
}
```

Los métodos que siguen resuelven el problema anterior pero con garantía de éxito.

```
/** Busca dato en un array de String CON garantía de éxito.
 * Devuelve su índice. */
public static int buscarPosEsta(String[] a, String dato) {
 int i = 0;
 while (!a[i].equals(dato)) i++;
 return i;
}

/** Busca un dato en un array de double CON garantía de éxito.
 * Devuelve su índice. */
public static int buscarPosEsta(double[] a, double dato) {
 int i = a.length-1;
 while (a[i]!=dato) i--;
 return i;
}
```

**Ejemplo 10.6.** Los métodos siguientes son otros ejemplos de búsqueda en los que el resultado de la misma puede ser también el propio valor encontrado o un valor lógico.

```
/** Comprueba si hay algún elemento del array mayor que dato. */
public static boolean estaMayor(double[] a, double dato) {
 int i = 0;
 while (i<a.length && a[i]<=dato) i++;
 return (i<a.length);
}

/** Devuelve, si existe, la posición del primer elemento del
 * array mayor que la suma de los anteriores.
 * En caso contrario, devuelve -1. */
public static int buscaPosMaySuma(int[] a, int dato) {
 int i = 0, suma = 0;
 while (i<a.length && a[i]<=suma) { suma+=a[i]; i++; }
 if (i<a.length) return i;
 else return -1;
}

/** Devuelve el primer String de longitud n del array.
 * Si no existe, devuelve la cadena vacía. */
public static String buscaLong(String[] a, int n) {
 int i = 0;
 while (i<a.length && a[i].length()!=n) i++;
 if (i<a.length) return a[i];
 else return "";
}
```

**Ejemplo 10.7.** Dado un array de enteros **a**, el siguiente método encuentra, si existe, la primera secuencia de tres elementos consecutivos ordenados de menor a mayor. Más exactamente, busca el primer índice *i* tal que  $a[i] \leq a[i+1] \leq a[i+2]$ .

La estrategia que se propone para resolver el problema limita la búsqueda de dicho índice al rango  $0..a.length-3$ , y en cada iteración se examina  $a[i]$  y su relación con los dos siguientes elementos. El avance del índice se realiza según el siguiente análisis de casos:

- Si  $a[i] \leq a[i+1]$ :
  - si  $a[i+1] \leq a[i+2]$ : *i* es el índice buscado,
  - en caso contrario,  $a[i+1] > a[i+2]$ : se debe seguir desde *i*+2.
- En caso contrario,  $a[i] > a[i+1]$ : se debe seguir desde *i*+1.

```
/** Devuelve, si existe, el menor índice i tal que
 * a[i]<=a[i+1]<=a[i+2]. En caso contrario, devuelve -1. */
public static int secuencia3(int[] a) {
 int i = 0, fin = a.length - 3;
 boolean encontrado = false;
 while (i<=fin && !encontrado) {
 if (a[i]<=a[i+1])
 if (a[i+1]<=a[i+2]) encontrado = true;
 else i = i+2;
 else i = i+1;
 }
 if (encontrado) return i; else return -1;
}
```

**Ejemplo 10.8.** Dado el array bidimensional `tempMed`, definido en segundo lugar en el ejemplo 10.1, el siguiente código busca una fecha en la que el valor almacenado sea igual a 40°C:

```
int i = 1, j = 0; boolean encontrado = false;
while (i<tempMed.length && !encontrado) {
 j = 1;
 while (j<tempMed[i].length && !encontrado) {
 encontrado = (tempMed[i][j]==40);
 j++;
 }
 i++;
}
if (encontrado) {
 System.out.print("40° medido el día " + (j-1));
 System.out.println(" del mes " + (i-1));
}
else System.out.println("Ningún día con 40°");
```

### *Esquemas combinados*

La solución de problemas reales exige a veces la realización combinada de técnicas de recorrido y búsqueda. A continuación se muestran dos ejemplos que ilustran este tipo de problemas.

**Ejemplo 10.9.** Dado un array `String[] a`, se desea conocer si existen componentes duplicados, determinando para cada `String` del array la primera repetición. El resultado debe ser un `String` en el que en cada línea conste el dato y las dos posiciones en las que aparece.

La estrategia a seguir será un recorrido del array en el que para cada elemento se realizará, a su vez, la búsqueda de una componente igual a él desde su posición en adelante.

```
/** Devuelve un String con cada dato y las posiciones en las
 * que aparece duplicado. */
public static String listaDuplicados(String[] a) {
 String res = "";
 for (int i=0; i<a.length-1; i++) {
 int j = i+1;
 while (j<a.length && !a[i].equals(a[j])) j++;
 if (j<a.length)
 res+=a[i] + " duplicado en: " + i + " y " + j + "\n";
 }
 return res;
}
```

**Ejemplo 10.10.** Dado un array `String[] a`, se desea obtener una nueva versión ordenada del mismo, esto es: obtener un nuevo array con todas las componentes (`String`) del primero, pero ordenadas alfabéticamente de forma ascendente.

Para resolver este problema se puede seguir la estrategia siguiente: iterar tantas veces como datos (`a.length` iteraciones) y, en cada iteración:

1. obtener el dato alfabéticamente menor (de los que queden, esto es, de los que aún no se hayan situado en el array solución),
2. escribir dicho dato en la posición correspondiente en el array solución,
3. marcar el dato para no considerarlo en iteraciones posteriores (usando un array auxiliar de `boolean`).

Este método es poco eficiente debido principalmente a la necesidad de comprobar si los elementos han sido ya escritos o no en el array destino. El problema general de la ordenación en arrays será tratado más adelante de manera adecuada.

```
/** Devuelve un array ordenado ascendentemente con las String
 * del array que recibe como argumento. */
public static String[] toArrayOrdenado(String[] a) {
 // array de String a devolver:
 String[] aux = new String[a.length];
 // array para marcar los ya escritos en aux:
 boolean[] marcado = new boolean[a.length];
 int pos = 0; // posición de escritura en aux
```

```
for (int i=0; i<a.length; i++) {
 // buscar el primer elemento aún no escrito en
 // aux, es una búsqueda con garantía de éxito:
 int posMin = 0;
 while (!marcado[posMin]) posMin++;
 // recorrer el resto del array buscando el mínimo:
 for (int j=posMin+1; j<a.length; j++)
 if (!marcado[j] &&
 a[j].compareTo(a[posMin])<0) posMin = j;
 // posMin contiene la posición de la String menor:
 aux[pos] = a[posMin];
 pos++; // escribir el mínimo en aux
 marcado[posMin] = true; // y marcarlo como escrito
}
return aux;
}
```

### 10.3.2 Acceso directo

Una de las principales ventajas de los arrays es que permiten el acceso directo a los elementos, es decir, que se puede acceder con tiempo constante<sup>2</sup> a cualquier componente siempre que se conozca su posición. Este hecho da lugar a toda una serie de algoritmos que resuelven problemas de manera eficiente. En esta sección se presentan algunos de ellos.

#### Búsqueda binaria o dicotómica

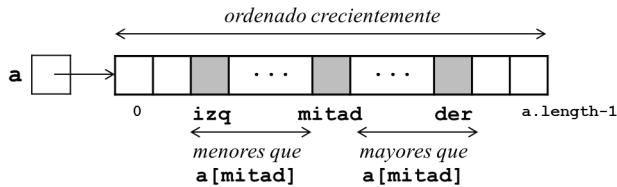
Un ejemplo representativo de acceso directo en un array es la *búsqueda binaria o dicotómica*. Es un caso particular del problema de búsqueda sobre un array ordenado.

El algoritmo de búsqueda binaria está basado en el hecho de que el array *a* está ordenado crecientemente o decrecientemente. En lo que sigue se supone una ordenación creciente del array, aunque el razonamiento para una ordenación decreciente sería análogo. Dado un elemento *x* a buscar en *a[inicio..fin]*, esto es, desde la posición *inicio* hasta la posición *fin*,  $0 \leq \text{inicio} \leq \text{fin} < \text{a.length}$ , se le compara con el elemento central *a[mitad]*. Si dicho valor es igual al buscado, entonces *x* ha sido encontrado. Si en cambio, *a[mitad]* es superior, entonces *x*, de encontrarse, sólo puede aparecer en *a[inicio..mitad-1]*: debe buscarse *x* en este subarray. Por último, si *a[mitad]* es inferior al valor buscado, *x* sólo puede aparecer en *a[mitad+1..fin]*. Aplicando reiteradamente esta estrategia, se puede reducir el tamaño de la zona de búsqueda de *x* en el array, cuyos extremos inferior

---

<sup>2</sup>Esto es, el tiempo de acceso a cualquier elemento es el mismo y está acotado.

y superior se delimitarán respectivamente con unas variables `izq` y `der` (véase figura 10.14).



**Figura 10.14:** Búsqueda binaria.

Para llevar a cabo la estrategia se deberá en cada iteración:

- Determinar el elemento central `mitad = (izq+der)/2`.
- Acceder al elemento central `a[mitad]`, comprobando que si:
  - `x = a[mitad]`, entonces: búsqueda finalizada,
  - `x < a[mitad]`, entonces: efectuar `der = mitad - 1`,
  - `x > a[mitad]`, entonces: efectuar `izq = mitad + 1`.

Además, inicialmente se tendrá que: `izq = inicio` y `der = fin`, donde `inicio` y `fin` son los extremos del subarray donde se realizará la búsqueda. Si la búsqueda se quiere extender a todo el array `inicio = 0` y `fin = a.length-1`. Siendo, además, la condición de finalización de la iteración:

- En el caso de una *búsqueda con éxito*, la condición es `x==a[mitad]`.
- En el caso de una *búsqueda sin éxito*, la condición es `izq=der+1` (que se puede redefinir, de forma más eficiente, como `izq>der`), ya que inicialmente, `izq<=der`, y en una iteración sólo se incrementa o decrementa uno de ambos valores en 1.

El algoritmo de búsqueda binaria propuesto es el siguiente:

```
/** 0<=inicio<=fin<a.length */
public static int busBinaria(int[] a, int inicio, int fin, int x) {
 int izq = inicio, der = fin, mitad = 0;
 boolean encontrado = false;
 while (izq<=der && !encontrado) {
 mitad = (izq+der)/2;
 if (x==a[mitad]) encontrado = true;
 else if (x<a[mitad]) der = mitad-1;
 else izq = mitad+1;
 }
 if (encontrado) return mitad;
 else return -1;
}
```

Si se quiere realizar la búsqueda en todo el array, esto es con valores `inicio = 0` y `fin = a.length-1`, se puede definir un método que invoque al método anterior como sigue:

```
public static int busBinaria(int[] a, int x) {
 return busBinaria(a,0,a.length-1,x);
}
```

Aunque la búsqueda binaria sólo se puede aplicar sobre arrays ordenados, en general, es mucho más eficiente que la búsqueda lineal, como se verá detalladamente en el capítulo 13.

### *Representación de un conjunto de naturales*

La representación más sencilla de un conjunto de naturales en el intervalo  $[0..n]$  es mediante un array `conjunto` de elementos de tipo `boolean` tal que se cumpla que `conjunto[i]=true` si  $i$  pertenece al conjunto y `conjunto[i]=false` en caso contrario. En la figura 10.15 se muestra la representación gráfica del conjunto  $\{0,1,2,4,8\}$  con  $n = 8$ .

|          |   |      |      |      |       |      |       |       |       |      |
|----------|---|------|------|------|-------|------|-------|-------|-------|------|
| conjunto | → | 0    | 1    | 2    | 3     | 4    | 5     | 6     | 7     | 8    |
|          |   | true | true | true | false | true | false | false | false | true |

**Figura 10.15:** Representación de un conjunto de naturales.

Se podría definir una clase `Conjunto` (véase figura 10.16) con dos atributos: el array `conjunto` de elementos de tipo `boolean` y un entero `ultimo` representando el valor máximo del intervalo.

### *Moda de un multiconjunto de naturales*

Otro ejemplo clásico de algoritmo que aprovecha el acceso directo de los arrays, es el que permite calcular la *moda* de un conjunto de naturales con elementos repetidos, es decir, de un multiconjunto. La moda de un multiconjunto se define como el elemento del mismo que más veces se repite.

Sea `a` un array con los datos del multiconjunto que se supone son naturales en el intervalo  $[0..n]$ <sup>3</sup>. El problema del cálculo de la moda se puede resolver mediante un recorrido secuencial ascendente de este array, utilizando un array auxiliar de contadores, por ejemplo `frecuencia`, de forma que `frecuencia[i]` es el número de veces que aparece  $i$  en el array `a`. Cuando termina el recorrido del array `a`, la moda se obtiene buscando el máximo del array `frecuencia`.

---

<sup>3</sup>Nótese que estos valores permiten indexar un array en Java.

```

/**
 * Clase Conjunto: permite representar un conjunto de naturales.
 * @author Libro IIP-PRG
 * @version 2011
 */
public class Conjunto {
 private boolean[] conjunto;
 private int ultimo;

 /** Constructor de un conjunto vacío, que
 * contendrá naturales en el rango [0..ult]. */
 public Conjunto(int ult) {
 conjunto = new boolean[ult+1];
 ultimo = ult;
 }

 /** Comprueba la pertenencia al cjto. actual
 * de un x dado, 0<=x<=ultimo. */
 public boolean pertenece(int x) { return conjunto[x]; }

 /** Consulta el cardinal del cjto. actual. */
 public int cardinal() {
 int card = 0;
 for (int i=0; i<conjunto.length; i++)
 if (conjunto[i]) card++;
 return card;
 }

 /** Añade al cjto. actual un x dado, 0<=x<=ultimo. */
 public void añade(int x) { conjunto[x] = true; }

 /** Genera de forma aleatoria los elementos del cjto. actual. */
 public void setAlAzar() {
 for (int i=0; i<conjunto.length; i++)
 conjunto[i] = (Math.random()>=0.5);
 }
 ...
}

```

**Figura 10.16:** Clase Conjunto.

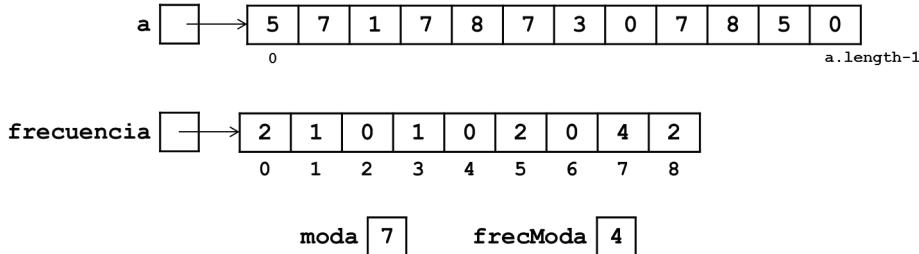
En la figura 10.17 se muestra la representación gráfica del problema del cálculo de la moda del multiconjunto  $C = \{5,7,1,7,8,7,3,0,7,8,5,0\}$  con  $n = 8$ .

```

/** Calcula la moda de un array que contiene
 * naturales comprendidos en el rango [0..n]. */
public static int modaDeaN(int[] a, int n) {
 // se construye un array entre 0 y n
 int[] frecuencia = new int[n+1];
 // recorrido de a y obtención de frecuencias
 for (int i=0; i<a.length; i++) frecuencia[a[i]]++;
}

```

```
// la moda es el máximo del array frecuencia
int moda = 0;
for (int i=1; i<frecuencia.length; i++)
 if (frecuencia[i]>frecuencia[moda]) moda = i;
return moda;
}
```

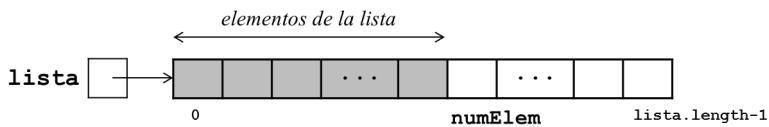


**Figura 10.17:** Moda de un multiconjunto de naturales.

Nótese que el array `frecuencia` podría utilizarse para representar un histograma de las frecuencias de aparición de las componentes del array original.

#### 10.4 Representación de una secuencia de datos dinámica usando un array

La representación más directa de una secuencia de datos se puede realizar utilizando un array. Es habitual que la secuencia sea dinámica en el sentido de que se puedan añadir o eliminar elementos. Aunque el array es una estructura de tamaño prestablecido y fijo, se puede simular sobre ella una lista dinámica, definiendo inicialmente un array `lista` de un tamaño máximo estimado `MAX_ELEM` y añadiendo una variable `numElem` que indica el número de elementos real de la lista. En todo momento, se mantiene que los elementos de la lista están situados en posiciones consecutivas del array desde 0 hasta `numElem-1` (véase figura 10.18).



**Figura 10.18:** Representación de una lista de elementos.

Con esta representación crear una lista vacía sería:

```
tipoBase[] lista = new tipoBase[MAX_ELEM];
int numElem = 0;
```

Añadir un elemento  $x$  a la lista sería:

```
if (numElem<MAX_ELEM) lista[numElem++] = x;
else // tratar el caso de lista llena
```

Borrar un elemento  $x$  de la lista respetando el orden de inserción, requeriría desplazar una posición a la izquierda todos los elementos posteriores al eliminado, como sigue:

```
int i = 0;
while (i<numElem && lista[i] distinto de x) i++;
if (i<numElem) {
 for (int j=i+1; j<numElem; j++)
 lista[j-1] = lista[j];
 numElem--;
}
else // búsqueda sin éxito
```

En el caso en que no fuera necesario preservar el orden, se podría sustituir el elemento a eliminar por el último (el que ocupa la posición  $\text{numElem}-1$ ).

Del mismo modo, se podrían representar listas ordenadas, teniendo en cuenta ahora que para insertar un elemento puede ser necesario el desplazamiento de parte de los datos una posición a la derecha, del siguiente modo:

```
int i = numElem-1;
while (i>=0 && lista[i] mayor que x) {
 lista[i+1] = lista[i];
 i--;
}
lista[i+1] = x;
numElem++;
```

**Ejemplo 10.11.** Supóngase definida la clase `Circulo` introducida en el capítulo 2. Se plantea ahora el problema de tratar una secuencia de círculos de forma que se puedan realizar, por ejemplo, las operaciones siguientes:

- crear una secuencia vacía de círculos,

- añadir un círculo a la secuencia,
- recuperar el círculo que se encuentra en una cierta posición del array,
- consultar el número de círculos de la secuencia,
- borrar la primera aparición de un círculo de un cierto color,
- consultar la posición en el array de un círculo de cierto color y radio,
- calcular la suma de las áreas de todos los círculos de la secuencia,
- obtener una representación en modo texto de la secuencia de círculos.

La clase **SecuenciaDeCirculos** (véase figura 10.19) se puede definir con tres atributos: un array de **Circulo** (**elArray**), un valor entero que determina la capacidad máxima del array (**CAPACIDAD\_DEL\_ARRAY**) y un valor entero con el número de círculos (**talla**).

```
/**
 * Clase SecuenciaDeCirculos: utilizando un array para almacenar
 * los círculos.
 * Atributos:
 * - talla, valor entero que representa el número de círculos.
 * - elArray, los círculos se almacenan desde 0 hasta talla-1.
 * - CAPACIDAD_DEL_ARRAY, número total de círculos que puede haber.
 * @author Libro IIP-PRG
 * @version 2011
 */
public class SecuenciaDeCirculos {
 private int talla;
 private Circulo elArray[];
 private static final int CAPACIDAD_DEL_ARRAY = 10;

 /** Constructor: crea una secuencia vacía de círculos. */
 public ArrayDeCirculos() {
 elArray = new Circulo[CAPACIDAD_DEL_ARRAY];
 talla = 0;
 }

 /** Añade un círculo al final de la secuencia.
 * @param c Círculo a añadir.
 * @return boolean true si se añade con éxito o
 * false si el array está lleno.
 */
 public boolean insertar(Circulo c) {
 boolean cabe = true;
 if (talla<elArray.length) elArray[talla++] = c;
 else cabe = false;
 return cabe;
 }
}
```

**Figura 10.19:** Clase SecuenciaDeCirculos.

```

/** Recupera el círculo que ocupa la posición indicada.
 * @param pos posición dentro de la secuencia
 * (el primer círculo ocupa la posición 0).
 * @return Círculo círculo que ocupa la posición pos o
 * null si pos<0 || pos>=talla.
 */
public Círculo recuperar(int pos) {
 if (pos>=0 && pos<talla) return elArray[pos];
 return null;
}

/** Consulta el número de círculos de la secuencia.
 * @return int número de círculos de la secuencia.
 */
public int talla() { return talla; }

/** Elimina la primera aparición de un círculo de color
 * col de la secuencia.
 * @param col String color del círculo a eliminar.
 * @return boolean true si se elimina con éxito o
 * false si no hay ningún círculo de ese color.
 */
public boolean eliminar(String col) {
 boolean ok = false;
 for (int i=0; i<talla && !ok; i++)
 if (elArray[i].getColor().equals(col)) {
 ok = true;
 for(int j=i; j<talla-1; j++)
 elArray[j] = elArray[j+1];
 talla--;
 }
 return ok;
}

/** Devuelve la posición de la primera aparición de un
 * círculo de color col y radio r de la secuencia.
 * @param col String color del Círculo a buscar.
 * @param r double radio del Círculo a buscar.
 * @return int posición del círculo en la secuencia
 * (el primer círculo ocupa la posición 0)
 * o -1 si no está en la secuencia.
 */
public int indiceDe(String col, double r) {
 int pos = -1;
 for (int i=0; i<talla && pos===-1; i++)
 if (elArray[i].getColor().equals(col) &&
 elArray[i].getRadio()==r) pos = i;
 return pos;
}

```

Figura 10.19: Clase SecuenciaDeCírculos (cont.).

```
/** Devuelve la suma de las áreas de todos los círculos
 * de la secuencia.
 * @return double suma de las áreas de todos los círculos.
 */
public double area() {
 double res = 0.0;
 for (int i=0; i<talla; i++) res += elArray[i].area();
 return res;
}

/** Devuelve una descripción de la secuencia de círculos.
 * @return String cadena de texto con la descripción de
 * la secuencia de círculos.
 */
public String toString() {
 String res = "Secuencia de " + talla + " círculos:\n";
 if (talla>0)
 for (int i=0; i<talla; i++)
 res += elArray[i].toString() + "\n";
 else res = "Secuencia vacía\n";
 return res;
}
```

Figura 10.19: Clase SecuenciaDeCirculos (cont.).

## 10.5 Problemas propuestos

1. Dado un array de enteros `v`, escribir un método de clase que:
  - a) Cuente las apariciones de un valor dado `x` en el array `v`.
  - b) Dadas dos posiciones, `izq` y `der`, del array,  $0 \leq izq \leq der \leq v.length - 1$ , duplique el valor de los elementos del array situados entre dichas posiciones.
  - c) Dadas dos posiciones, `izq` y `der`, del array,  $0 \leq izq \leq der \leq v.length - 1$ , invierta todos los elementos del array situados entre dichas posiciones, esto es, al finalizar la ejecución del método el array contendrá en su posición `izq` el elemento que inicialmente ocupaba la posición `der`, en su posición `izq+1` el elemento que inicialmente ocupaba la posición `der-1` y así sucesivamente.
  - d) Dadas dos posiciones, `izq` y `der`, del array,  $0 \leq izq \leq der < v.length - 1$ , desplace una posición hacia la derecha todos los elementos de `v` comprendidos entre las posiciones `izq` y `der`, ambas inclusive.
  - e) Dadas dos posiciones, `izq` y `der`, del array,  $0 < izq \leq der \leq v.length - 1$ , desplace una posición hacia la izquierda todos los elementos de `v` comprendidos entre las posiciones `izq` y `der`, ambas inclusive.
  - f) Cuente el número de elementos impares en posiciones pares.
  - g) Cuente el número de elementos que son menores que un valor dado `x` hasta una posición `n` del array.
  - h) Determine si dicho array está ordenado ascendenteamente.
  - i) Determine la posición, si existe, de la primera subsecuencia del array que comprenda, al menos tres números consecutivos en posiciones consecutivas del array.
  - j) Dado un entero `x` no negativo, determine si la suma de los elementos del array es mayor que `x`, recorriendo el mínimo imprescindible de elementos.
  - k) Dados dos valores enteros `x` y `n`, devuelva la posición de la primera subsecuencia de `n` enteros mayores que `x`, o devuelva `-1` en caso de que no exista dicha subsecuencia en el array.
  - l) Cuente cuántos ceros consecutivos hay al final del array, realizando la menor cantidad de comprobaciones posibles.
  - m) Obtenga la posición del último elemento impar del array, si existe.
  - n) Sume los elementos que aparecen tras el primer impar, si existe.

2. ¿Cómo habría que modificar el código siguiente para que fuera equivalente al primer esquema de búsqueda ascendente presentado en la sección 10.3.1?

```
boolean encontrado = false;
for (int i=inicio; i<=fin && !encontrado; avanzar(i))
 if (propiedad(a[i])) encontrado = true;

// Resolución de la búsqueda
if (encontrado) ...
else ...
```

3. Dado cierto array de enteros *v*, considérese el siguiente segmento de código:

```
int i;
for (i=0; i<v.length && v[i]==0; i++);
if (v[i] != 0) return i;
else return -1;
```

¿Qué ocurre si se ejecuta el segmento de código anterior cuando todos los elementos del array *v* valen inicialmente 0?

4. ¿Qué hace el siguiente código?

```
public static int f(int x, int[] a) {
 int encontrado = -1, i = 0, j = a.length-1;
 while (i<=j && encontrado===-1) {
 if (a[i]==x) encontrado = i;
 else if (a[j]==x) encontrado = j;
 i++; j--;
 }
 return encontrado;
}
```

5. Escribir un método estático que determine si un array de palabras (*String*) es capicúa, esto es, para determinar si la primera y última palabras del array son la misma, la segunda y la penúltima palabras también lo son, y así sucesivamente. El método retornará *true* si el array es capicúa o *false* en caso contrario.
6. Dados dos arrays de *double*, implementar un método de clase que compruebe si el primero es prefijo del segundo, esto es, que todos los elementos del primer array se encuentren en el mismo orden al comienzo del segundo.
7. Escribir un método de clase para sustituir en cierto array *a* de caracteres todas las apariciones de la pareja de letras ‘no’ por la pareja ‘si’.

8. Sea **a** un array de **int**,  $a.length > 0$ , cuyas componentes tienen valores comprendidos entre 0 y 9 inclusive. Se debe escribir un método de clase con el siguiente perfil:

```
public static void cifras(int[] a)
```

que escriba en la salida estándar las primeras componentes del array en las que no hay elementos consecutivos repetidos. Por ejemplo:

- Si **a** es {8,8,4,3}, se escribe 8.
- Si **a** es {4,0,5,9,9}, se escribe 4059.
- Si **a** es {0,9,4,5,9}, se escribe 09459.
- Si **a** es {1,7,1,0,0,8,7}, se escribe 1710.

9. Se pide una variante del método anterior, con perfil:

```
public static int cifras(int[] a)
```

tal que, siendo **a** un array con las mismas características, calcule y devuelva un **int** cuyas cifras sean las primeras componentes del array en las que no hay elementos consecutivos repetidos. Por ejemplo:

- Si **a** es {1,3,4,4,0,5,1}, se devuelve 134.
- Si **a** es {0,4,2,8,8,7}, se devuelve 428.
- Si **a** es {8,7,8,5,5}, se devuelve 8785.
- Si **a** es {8,8,4,3}, se devuelve 8.

10. Dado **a** un array de **int**, se debe diseñar un método de clase con la siguiente cabecera:

```
public static int suma(int[] a, int i, int j)
```

que devuelva la suma de los elementos que sean simétricos e iguales en el array **a** comprendidos entre las posiciones **i** y **j**,  $0 \leq i \leq j < a.length$ . Se entiende que, en el caso de que el número de elementos en **a[i..j]** sea impar, el elemento central es simétrico e igual a sí mismo. Por ejemplo:

- Si **a** es {1,2,3,2,1}, da como resultado 9.
- Si **a** es {1,2,3,2,5}, da como resultado 7.
- Si **a** es {1,2,3,5,1}, da como resultado 5.
- Si **a** es {1,2,3,2}, da como resultado 0.
- Si **a** es {1,2,3,1}, da como resultado 2.

11. Dados **a** y **b** dos arrays de enteros, se debe escribir un método de clase con el perfil:

```
public static void sumarEnB(int[] a, int[] b)
```

que modifique tantas componentes de **b** como sea posible, incrementando su valor con el de la misma componente de **a**. Más exactamente, si **n** es el mínimo de **a.length** y **b.length**, el método deberá sumar a cada una de las **n** primeras componentes de **b**, la misma componente de **a**.

12. ¿Qué muestra el siguiente código por pantalla?

```
class Traza{
 public static void main(String[] args) {
 int[] array1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
 int[] array2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
 f(array1);
 f(array2);
 }

 public static void f(int[] a) {
 int i = 0, j = a.length-1;
 if (j%2==0) j--;
 while (i<a.length && j>=0) {
 System.out.print(" " + a[i] + " " + a[j]);
 i+=2;
 j-=2;
 }
 System.out.println();
 }
}
```

13. Se pide la implementación de un método de clase con el siguiente perfil:

```
public static boolean detectar(char[] s1, char[] s2)
```

que recibe como argumentos dos arrays de tipo **char** y devuelve un valor de tipo **boolean**. El método devolverá **true** si la cadena de caracteres almacenada en **s1** está contenida en la que hay en **s2**, aunque no esté contenida en un solo bloque. Es decir, puede estar fragmentada pero en el mismo orden. Por ejemplo, “Castor” está dentro de “Ayer, en Castellón hubo tormenta”. Sin embargo, no está dentro de “Ayer hubo tormenta en Castellón”.

14. Implementar una clase de utilidades **Matrices** que permita realizar operaciones útiles con matrices de reales. Debe incluir los siguientes métodos estáticos para:

- a) Calcular el producto de dos matrices, **a** y **b**, de dimensiones **m×n** y **n×p**, respectivamente.

- b) Dada una matriz  $a$  con  $n$  filas y  $m$  columnas, obtener la matriz traspuesta de  $a$ , una matriz  $b$  con  $m$  filas y  $n$  columnas. Se llama matriz traspuesta de una matriz  $a$  de dimensión  $n \times m$ , a la matriz que se obtiene al cambiar en  $a$  las filas por columnas o las columnas por filas. Se representa por  $a^t$  y su dimensión es  $m \times n$ .
- c) Dada una matriz  $a$  cuadrada (de dimensión  $n \times n$ ), obtener la matriz traspuesta de  $a$ , modificando la propia matriz  $a$ .
- d) Dada una matriz  $a$  cuadrada (de dimensión  $n \times n$ ), comprobar si es o no simétrica con respecto a su diagonal principal.
- e) Comprobar, dada una matriz  $a$ , si existe algún elemento en una posición  $[i, j]$  que sea igual a la suma de elementos de la submatriz de  $[0, 0]$  hasta  $[i-1, j-1]$ .
- f) Comprobar si una matriz  $a$  cuadrada (de dimensión  $n \times n$ ) es estrictamente diagonal dominante por filas. Una matriz  $a$  de dimensión  $n$  es estrictamente diagonal dominante por filas cuando, para todas las filas, el valor absoluto del elemento de la diagonal de esa fila es estrictamente mayor que la suma de los valores absolutos del resto de elementos de esa fila.
- g) Calcular la 1-norma de una matriz  $a$ . La 1-norma se define como el máximo de la suma del valor absoluto de los elementos de cada columna de la matriz. Se supondrá que la matriz es cuadrada, de dimensión  $n \times n$ .
- h) Multiplicar una matriz diagonal (tan sólo tiene elementos no nulos en su diagonal) por un entero dado. Se deben realizar el mínimo de recorridos posibles.
- i) Dado un array bidimensional, devolver un array con la fila que tenga la suma de sus elementos máxima.
- j) Dado un array bidimensional, devolver un array con la columna que tenga la suma de sus elementos mínima.
15. Dado un array de `char` llamado `palabra` y  $m$  una matriz de `char` en la que todas sus filas tienen la misma longitud que `palabra`, buscar si `palabra` coincide, carácter a carácter, con alguna fila de  $m$ . Si es el caso, el método debe devolver el índice de la primera fila que encuentre (la de menor índice). En caso contrario, debe devolver -1. El método no hará comparaciones de caracteres innecesarias.

16. ¿Qué calcula el siguiente método?

```
public static int calcular(int[][] m) {
 boolean enc = false;
 int n = m.length, i = 0;
 while (i < n && !enc) {
 int s = 0;
 for (int j=0; j < n-1; j++) s = s + m[i][j];
 if (m[i][n-1] <= s) enc = true;
 else i++;
 }
 return i;
}
```

17. Modificar el algoritmo de la búsqueda binaria de la sección 10.3.2 para aplicarlo a un array de **String** cuyos elementos están en orden lexicográfico.
18. Completar la clase **Conjunto** de la sección 10.3.2 con los métodos siguientes para calcular:
- la unión del conjunto actual (**this**) con otro conjunto dado,
  - la intersección del conjunto actual (**this**) con otro conjunto dado,
  - la diferencia entre el conjunto actual (**this**) y otro conjunto dado,
  - el complemento del conjunto actual (**this**).
19. Suponiendo que se han medido las temperaturas medias diarias de una ciudad durante un año y almacenado los resultados en el array bidimensional **tempMed**, definido en segundo lugar en el ejemplo 10.1, modificar el método **modaDe0aN** de la sección 10.3.2 para obtener por pantalla el histograma de frecuencia de temperaturas:
- Sabiendo que la temperatura mínima fue de 0 grados y la máxima de 38. El histograma constará de 39 líneas con las temperaturas de 0 a 38 y el número de días en que se alcanzó dicha temperatura.
  - Si no se sabe cuáles fueron las temperaturas máxima y mínima y sabiendo que una o ambas pueden ser negativas (por ejemplo, temperaturas máxima y mínima de Moscú -10°C y +38°C, respectivamente).
20. Dada una secuencia de palabras almacenada en un array de **String**, escribir métodos de clase para:
- Calcular el histograma de las longitudes de las palabras que contiene. Para ello se debe utilizar un array **histo** con 15 contadores tal que **histo[i]** contiene el número de palabras de longitud **i+1** que se han detectado en el texto para  $0 \leq i \leq 14$  y **histo[14]** contiene el número de palabras de longitud mayor o igual que 15.

- b) Mostrar el histograma por pantalla con asteriscos (para cada longitud de palabra *i*, dos líneas de asteriscos con tantos asteriscos por línea como indique `histo[i-1]`).
- c) Calcular la moda de las longitudes de las palabras de la secuencia, sabiendo que, en este caso, la moda es la longitud más frecuente.
21. Completar la clase `SecuenciaDeCirculos` de la sección 10.4 con las siguientes operaciones:
- Método modificador que elimine (si existe) un objeto `Círculo` pasado como parámetro. El método devolverá `true` si el borrado se realiza con éxito y, en caso contrario, devolverá `false`.
  - Método consultor que indique en qué posición se encuentra un objeto `Círculo` pasado como parámetro. El método devolverá `-1` si dicho objeto no se encuentra.
  - Método que dibuje todos los objetos `Círculo` del array `elArray` en un objeto `Pizarra` pasado como parámetro. Supóngase que las clases `Pizarra` y `Círculo` (definidas en el capítulo 2) se encuentran en el mismo paquete que la clase `SecuenciaDeCírculos`.
22. Se desea realizar una aplicación `GestorAgenda` para gestionar una agenda telefónica. Se deben definir las clases que siguen:
- La clase `ItemAgenda` para representar cada entrada de una agenda mediante los atributos `telefono` y `nombre` (ambos de tipo `String`) y con las siguientes operaciones:
    - Un método constructor de un objeto `ItemAgenda` a partir de un teléfono y un nombre dados.
    - Dos métodos consultores y dos métodos modificadores de los atributos de instancia.
    - Un método `toString` para transformar el `ItemAgenda` actual en un `String`. Por ejemplo,

Nombre: Pedro Sancho  
Teléfono: 667890231
  - La clase `Agenda` para representar una agenda telefónica mediante un array `agenda` de, como mucho, 250 elementos de tipo `ItemAgenda` y una variable `numElem` de tipo `int` para indicar el número de entradas realmente existentes en la agenda en un momento dado. Y con las siguientes operaciones:
    - Un método constructor de un objeto `Agenda` con 0 entradas.
    - Un método consultor del número de entradas.

- Un método modificador `añadir` que, dado un `ItemAgenda`, lo añada ordenadamente en la `Agenda` actual. Se debe tener en cuenta que:
  - Un `ItemAgenda` es válido si `telefono` consta de 9 dígitos siendo el primero no nulo y `nombre` es una secuencia de caracteres cualesquiera con, a lo sumo, 40 caracteres.
  - En la agenda no pueden haber `nombres` repetidos.
  - Los elementos de `agenda` están ordenados alfabéticamente por el campo `nombre` de cada `ItemAgenda`.

Así, el `ItemAgenda` se podrá añadir si cabe, es un `ItemAgenda` válido y no está repetido. Si no se puede añadir se mostrará por pantalla un mensaje indicando el motivo.

Añadir un nuevo `ItemAgenda` implicará:

- a) encontrar la posición `pos` que debe ocupar para que el array `agenda` quede ordenado,
- b) desplazar una posición hacia la derecha todos los elementos del array desde la posición `pos` hasta la posición `numElem-1`,
- c) insertar en `pos` el nuevo `ItemAgenda`,
- d) incrementar `numElem` en uno.

Al final, devolverá como resultado `numElem`.

- Un método consultor `buscar` que, dado un `nombre`, encuentre el `ItemAgenda` asociado a dicho `nombre` en el array `agenda`. Si existe, devolverá la posición que ocupa en el array. Si no existe, devolverá -1.
- Un método modificador `borrar` que, dado un `nombre`, eliminará del array `agenda` el `ItemAgenda` asociado a dicho nombre. Implicará buscarlo. Si no existe, se mostrará por pantalla un mensaje indicándolo y si existe, se compactará el array, desplazando una posición a la izquierda todos los elementos posteriores al elemento eliminado, decrementándose `numElem` en uno. Al final, se devolverá como resultado `numElem`.
- Un método `toString` para transformar la `Agenda` actual en un `String`. Por ejemplo,

```
Nombre: Ana López
Teléfono: 654321287
=====
Nombre: Pedro Sancho
Teléfono: 667890231
=====
...
```

- En la clase **GestorAgenda** se probará el comportamiento de las clases anteriores. En esta clase, se implementará una función **menu** con el siguiente perfil: **public static int menu()** que presenta el siguiente menú de opciones por pantalla:

```
***** AGENDA *****
1. Mostrar
2. Añadir
3. Buscar
4. Borrar
0. Terminar

Elige una opción:
```

y devuelve la opción válida elegida por el usuario de entre las opciones posibles.

En esta clase se debe, obligatoriamente, implementar el método **main** en el que se gestionan las operaciones de la agenda telefónica. Se pide:

- Crear una agenda.
- Presentar el menú comentado anteriormente para que permita al usuario, de forma iterada, seleccionar la operación que desea realizar con el gestor hasta que desee terminar.
- Gestionar adecuadamente la opción seleccionada hasta que decida terminar.
  - La opción **1. Mostrar** consiste en mostrar por pantalla todas las entradas de la agenda.
  - La opción **2. Añadir** consiste en:
    - Solicitar un nombre y un teléfono y crear un **ItemAgenda**.
    - Llamar al método **añadir** para añadirlo a la agenda.
  - La opción **3. Buscar** consiste en:
    - Solicitar un nombre.
    - Llamar al método **buscar**.
    - Mostrar por pantalla el mensaje que corresponda.
  - La opción **4. Borrar** consiste en:
    - Solicitar un nombre.
    - Llamar al método **borrar**.
  - Con la opción **0. Terminar** acaba la ejecución del programa.

23. Se desea realizar una aplicación `GestorHospital` para gestionar el ingreso y el alta de pacientes de un hospital. Para facilitar el desarrollo de este ejercicio, se supone disponible:

- La clase `Paciente` que permite representar un paciente mediante los atributos: `nombre` (`String`), `edad` (`entero`), `estado` (`entero` entre 1 -más grave- y 5 -menos grave-), y con las siguientes operaciones:
  - `public Paciente(String n, int e)`. Constructor de un objeto `Paciente` de nombre `n`, de `e` años y cuyo `estado` es un valor aleatorio entre 1 y 5.
  - `public int getEdad()`. Consultor que devuelve `edad`.
  - `public int getEstado()`. Consultor que devuelve `estado`.
  - `public void tratamiento()`. Modificador que incrementa en uno `estado`.
  - `public String toString()`. Transforma el paciente en un `String`. Por ejemplo,

Pepe Pérez Santarosa

46 5

Se deben definir las clases que siguen, teniendo en cuenta que sus atributos serán privados y sus métodos públicos o privados y sólo los que se indican.

- La clase `Hospital` contiene la información de las camas de un hospital, así como de los pacientes que las ocupan. Un `Hospital` tiene un número máximo de camas `MAXC = 200` y para representarlas se utilizará un array `listaP` de objetos de tipo `Paciente` junto con un atributo `numLibres` que indique el número de camas libres del hospital en un momento dado. El número de cada cama coincide con su posición en el array de pacientes (la posición 0 no se utiliza), de manera que `listaP[i]` es el `Paciente` que ocupa la cama `i` o es `null` si la cama está libre. Las operaciones de esta clase son:

- `public Hospital()`. Constructor de un hospital. Cuando se crea un hospital, todas las camas están libres.
- `public int getNumLibres()`. Consultor del número de camas libres `numLibres`.
- `public boolean hayLibres()`. Devuelve `true` si en el hospital hay camas libres y devuelve `false` en caso contrario.
- `public int primeraLibre()`. Devuelve el número de la primera cama libre de `listaP` si hay camas libres o un 0 si no las hay.
- `public void ingresarPaciente(String n, int e)`. Si hay camas libres, la primera de ellas (la de número menor) pasa a estar ocupada por el paciente de nombre `n` y edad `e`. Si no hay camas libres, muestra un mensaje por pantalla.
- `private void darAltaPaciente(int i)`. La cama `i` del hospital pasa a estar libre.

- `public void darAltas()`. Se suministra el tratamiento a todos los pacientes del hospital y a aquellos pacientes sanos (cuyo `estado` es 6) se les da el alta médica (se invoca a `darAltaPaciente`).
- `public String toString()`. Transforma el array de camas en un `String`. Por ejemplo,

```

1 María Medina Muñoz 30 4
2 Pepe Pérez Santarosa 46 5
3 libre
4 Juan López Ayala 50 1
5 libre
...
200 Andrés Sánchez Encarnación 29 3

```

- En la clase `GestorHospital` se probará el comportamiento de las clases anteriores. En esta clase, se implementará una función `menu` con el siguiente perfil: `public static int menu()` que presenta el siguiente menú de opciones por pantalla:

```

***** HOSPITAL *****
1. Ingresos
2. Altas
3. Visualización
0. Terminar

Elige una opción:

```

y devuelve la opción válida elegida por el usuario de entre las opciones posibles.

En esta clase se debe, obligatoriamente, implementar el método `main` en el que se gestiona el ingreso y el alta de pacientes del hospital. Se pide:

- Crear un hospital.
- Presentar el menú comentado anteriormente para que permita al usuario, de forma iterada, seleccionar la operación que desea realizar con el gestor hasta que deseé terminar.
- Gestionar adecuadamente la opción seleccionada hasta que decida terminar.
  - La opción 1. `Ingresos` consiste en:
    - Solicitar el nombre y la edad de un paciente.
    - Llamar al método `ingresarPaciente`.
  - La opción 2. `Altas` consiste en llamar al método `darAltas`.
  - La opción 3. `Visualización` consiste en mostrar por pantalla la ocupación de las camas del hospital o un mensaje indicando la ausencia de pacientes en el mismo.
  - Con la opción 0. `Terminar` acaba la ejecución del programa.

## Más información

[Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.  
URL: <http://math.hws.edu/javanotes/>. Capítulo 7 (7.1 y 7.2).

[Ora11d] Oracle. *The Java<sup>TM</sup> Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Language Basics - Arrays.

[Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.  
Capítulo 6.

# Capítulo 11

## Recursión

En este capítulo se introducen los algoritmos recursivos y su implementación como métodos recursivos. Mediante la recursión es posible efectuar la repetición de algunos cálculos sin utilizar estructuras de iteración explícitas. Por ello y debido además a que los algoritmos recursivos pueden facilitar la solución de algunos problemas, la recursión constituye una herramienta fundamental en la resolución de múltiples problemas computacionales.

En la naturaleza y en las matemáticas es frecuente encontrar ejemplos en los que una definición de objetos por enumeración es complicada. Un ejemplo típico es la definición del conjunto de los números naturales que, al ser infinito, es imposible hacer por enumeración. Los axiomas de Peano, basándose en la definición de *sucesor*, establecen los términos básicos de la definición de un número natural:

1. El 1 es un número natural.
2. Si  $n$  es un número natural, entonces el sucesor de  $n$  también es un número natural.

Como se puede ver, la generalidad de la definición de número natural se apoya a su vez en la definición de número natural (además de en la de sucesor). Sólo el caso del número 1 se define por sí mismo.

A este tipo de definiciones, en las que un concepto se define basándose en sí mismo (excepto posiblemente para una serie de casos triviales), se les denomina *definiciones recursivas*. También se suele hablar de *razonamiento inductivo*, ya que a partir de unos casos sencillos se es capaz de encontrar una definición general que se apoya en un caso más simple; en el ejemplo de los naturales, el caso más sencillo de partida es el del número 1, y utilizándolo es posible generar por inducción, empleando la definición de sucesor, el resto de números naturales.

La *definición recursiva* o *inductiva* de la solución de un problema se caracteriza por que, a partir de un conjunto finito de casos sencillos del problema para los que se conoce la solución, se realiza una *hipótesis* de cuál es la solución general de dicho problema:

- La *hipótesis de inducción* para el caso general se expresa en términos de la(s) solución(es) del mismo problema para el(los) caso(s) más sencillo(s).
- Debe ser validada o demostrada mediante un proceso de *prueba por inducción*, sin el cual no deja de ser más que una hipótesis.

De igual manera que hay definiciones recursivas, hay problemas que pueden resolverse mediante el uso de *algoritmos recursivos*. La ventaja que presenta el uso de algoritmos recursivos es que se definen de forma inductiva, algo que en muchos casos es una ventaja frente a la formulación deductiva de un problema. Así, ante un problema complejo en ocasiones es mucho más sencillo formular su solución de forma inductiva (en términos de soluciones para casos más simples del problema) que de forma deductiva (por enumeración de los pasos necesarios para obtenerla).

La idea clave es que para resolver un problema complejo mediante un algoritmo recursivo se descompone el problema en una serie de problemas más simples que se resuelven *empleando el mismo algoritmo*, para luego componer la solución del problema complejo en base a las soluciones de los problemas más simples. Evidentemente, llegará un momento en que el problema se habrá simplificado lo suficiente como para que su solución sea inmediata, lo cual pone fin a la resolución recursiva y da la finitud necesaria para que el proceso sea considerado algorítmico (recuérdese que todo algoritmo se caracteriza por ser finito).

Tanto en definiciones matemáticas como en algoritmos se puede ver que hay dos situaciones distintas a la hora de resolver el problema planteado:

- El problema o dato es lo suficientemente simple para poder ser resuelto de manera trivial (en el caso de los números naturales, es así para el 1); este caso (o conjunto de casos) se denomina *caso base*.
- El problema o dato requiere del uso de la resolución de una instancia más simple para hallar su solución; a este caso se le llama *caso general*.

Una definición recursiva, al igual que un algoritmo recursivo, puede presentar varios casos base y casos generales, aunque de momento la mayor parte de los ejemplos que se expondrán tendrán un único caso base y caso general.

Una vez planteado un algoritmo recursivo es necesario verificar:

1. La *terminación* del algoritmo, es decir, que en algún momento se llegará al caso base a partir de cualquier caso general planteado inicialmente.
2. La *corrección* del algoritmo, es decir, que para cualquier subproblema que se dé, la solución del algoritmo es correcta (que suele requerir una demostración matemática por *inducción* sobre algún parámetro del algoritmo).

## 11.1 Diseño de un método recursivo

A la hora de resolver un problema recursivo, se suele partir de una formulación formal del mismo que plantea la recursividad. Un ejemplo clásico es la definición recursiva de la función factorial ( $n!$ ) de un número  $n$ , entero mayor o igual que 0, que sigue la siguiente recurrencia:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n - 1)! & \text{si } n > 0 \end{cases}$$

A partir de esta formulación, se debe conseguir un algoritmo que resuelva el problema de obtener el factorial de un número  $n$  (entero mayor o igual que 0). La implementación de estos algoritmos recursivos en Java requiere el uso de *métodos recursivos*. Un método recursivo se caracteriza por ser un método donde se ejecuta al menos *una llamada a sí mismo*, generalmente variando los parámetros de la llamada.

Un método recursivo debe constar en su cuerpo de una o varias condiciones que permiten distinguir en qué caso (base o general) se encuentra y una o varias instrucciones que permiten obtener la solución del caso correspondiente. A continuación, se muestra una posible implementación recursiva del factorial, indicando qué partes del código constituyen cada una de las partes generales de un algoritmo recursivo.

```
/** n>=0 */
public static int factorial(int n) {
 if (n==0) // Condición del caso base
 return 1; // Instrucciones del caso base
 else
 return n * factorial(n-1); // Instrucciones del caso general
}
```

Así pues, a la hora de diseñar un método recursivo, han de tenerse en cuenta:

- Las condiciones de los distintos casos base y las condiciones de los distintos casos generales.
- Las acciones de los distintos casos base (que pueden ser vacías en algunos casos) y las acciones de los distintos casos generales (que como mínimo deben de incluir una llamada al mismo método, lo que dará la naturaleza recursiva).

Todo método recursivo debe tener un caso base que garantice el fin de la recursividad. Además, las llamadas que se producen en el caso general deben de garantizar que se van acercando cada vez más, de forma estricta, a la condición del caso base, a fin de garantizar la finitud del algoritmo. Todo esto lleva a plantear que los métodos recursivos siguen un esquema de código semejante al que sigue:

```
[modificadores] [static] TipoRetorno nomMetRecursivo(listaParams){
 TipoRetorno resMet, res1, res2, ..., resK;
 if (casoBase(listaParams))
 resMet = solucionBase(listaParams);
 else {
 res1 = nomMetRecursivo(anterior1(listaParams));
 res2 = nomMetRecursivo(anterior2(listaParams));
 ...
 resK = nomMetRecursivo(anteriorK(listaParams));
 resMet = combinar(listaParams, res1, res2, ..., resK);
 }
 return resMet;
}
```

en donde:

- **listaParams** es la lista de parámetros del método recursivo.
- **casoBase** verifica si la condición del caso base es cierta.
- **solucionBase** obtiene la solución trivial para un caso base.
- **anterior $i$**  indica la descomposición del caso actual (**listaParams**) en el  $i$ -ésimo caso más sencillo (estRICTAMENTE más cercano al caso base que el actual).
- **combinar** indica la recombinación de las soluciones de los casos más sencillos para obtener la solución del caso actual.

Como se puede ver, la estructura fundamental consiste en tener una instrucción condicional que permita distinguir entre caso base y general. Por otro lado, las llamadas recursivas deben efectuarse cada vez para casos estrictamente más simples (los métodos *anterior i* tienen que garantizar que los datos sobre los que se opera estén estrictamente más cercanos al caso base que los de la llamada previa al método recursivo).

Generalmente, cuando se desea resolver un problema utilizando un algoritmo (o método) recursivo se consideran las siguientes etapas:

1. Enunciar completamente el problema, declarando la cabecera del método que se va a construir, y estableciendo tanto las *condiciones de entrada* para las que deberá ejecutarse el algoritmo, como el *resultado esperado* de dicha ejecución.

Además de por motivos obvios, la definición del problema es importante porque a menudo durante la misma se pone de relieve la estructura recursiva que puede darse en el problema, o se facilita su determinación.

2. Análisis de casos. Consiste en hacer explícito el caso base y el caso general de la recursión, estableciendo para cada caso las instrucciones pertinentes para resolver el problema. Hay que comprobar que se cubre cualquier caso posible, esto es, que ante cualquier posible entrada del problema, se efectúa el caso base o el general.
3. Transcribir el algoritmo que se está describiendo a un método del lenguaje utilizado.
4. Determinar que en cada nueva llamada recursiva se cumple que:
  - El nuevo problema que se debe resolver es estrictamente más cercano al caso base que el problema original, y
  - los datos de entrada de la nueva llamada cumplen las condiciones de entrada en las que se ha establecido que se deberá ejecutar el algoritmo.

## 11.2 Tipos de recursión

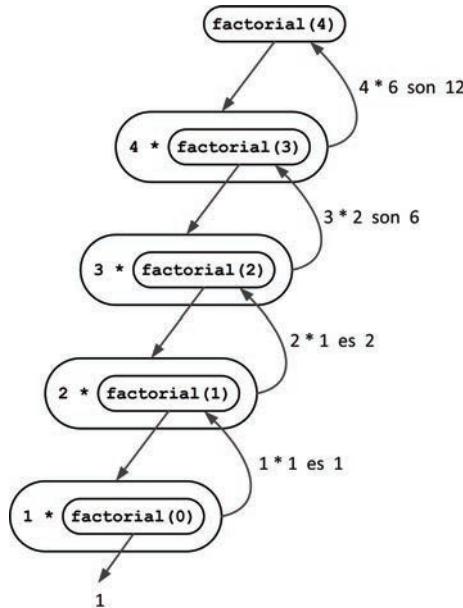
En un algoritmo (o método) recursivo, alguna de sus instrucciones hace referencia directa o indirecta a si mismo. Esto es, alguna de sus instrucciones es una llamada explícita a si mismo (en cuyo caso se denomina *recursión directa*), o bien es una llamada a otro algoritmo, a través del cual puede llegarse a hacer una llamada al primero (denominándose entonces *recursión indirecta*). Aunque ambas situaciones son posibles, este capítulo se centra en los algoritmos (o métodos) recursivos directos.

En función de cuántas llamadas recursivas se hacen y de cómo se recombinan las soluciones de los casos más simples se pueden distinguir varias categorías de algoritmos recursivos. Estas categorías suelen relacionarse con su complejidad computacional (capítulo 12) y la facilidad de convertirlos en algoritmos iterativos equivalentes. La clasificación es la siguiente:

- *Recursión lineal*: hay a lo sumo una sola llamada recursiva en cada ejecución del algoritmo, generando una *secuencia de llamadas* (figura 11.1). Según la recombinación usada, a su vez, se clasifica en:
  - *Final*: el resultado de la llamada recursiva es el propio resultado de la llamada actual; es decir, la solución del caso más simple es a su vez la del caso más complejo, con lo que no hay combinación de resultados.
  - *No final*: el resultado de la llamada recursiva se emplea para calcular el resultado de la llamada actual, arrojando un resultado posiblemente distinto al de la llamada recursiva.
- *Recursión múltiple*: hay más de una llamada recursiva en cada ejecución del algoritmo, y sus resultados han de combinarse para obtener la solución del caso actual. Se genera un *árbol de llamadas* (figura 11.3).

Atendiendo a esta clasificación, el método de cálculo del factorial que se ha presentado es un método lineal (se hace una única llamada recursiva a `factorial(n-1)` en cada ejecución) no final (el resultado de la llamada se multiplica por `n` para devolverse). En la figura 11.1 se muestra gráficamente la secuencia de llamadas realizadas al método `factorial`, a partir de la llamada inicial `factorial(4)`. La siguiente es una traza de la ejecución del método para calcular  $4!$ , en donde se indica el orden en el que se producen y el orden en el que finalizan las llamadas recursivas ejecutadas, así como el resultado de cada llamada.

| Llamadas ejecutadas       | Orden de ejecución | Orden de finalización | Resultado                      |
|---------------------------|--------------------|-----------------------|--------------------------------|
| <code>factorial(4)</code> | 1                  | 5                     | $4 * \text{factorial}(3) = 24$ |
| <code>factorial(3)</code> | 2                  | 4                     | $3 * \text{factorial}(2) = 6$  |
| <code>factorial(2)</code> | 3                  | 3                     | $2 * \text{factorial}(1) = 2$  |
| <code>factorial(1)</code> | 4                  | 2                     | $1 * \text{factorial}(0) = 1$  |
| <code>factorial(0)</code> | 5                  | 1                     | 1                              |



**Figura 11.1:** Secuencia de llamadas del método recursivo `factorial`.

### 11.3 Recursividad y pila de llamadas

La ejecución de un método recursivo es similar a la de cualquier otro método, aunque conviene tener presente que en el caso recursivo un método efectuará llamadas a sí mismo. Las características más relevantes de dicha ejecución pueden resumirse del modo siguiente:

- Cada vez que comienza la ejecución de un método, debido a una llamada al mismo, se apila un registro de activación en la pila de llamadas. Nótese que cada una de las diferentes ejecuciones recursivas de un mismo método está, por lo tanto, asociada a un registro de activación propio. Es decir, existen tantos registros de activación como llamadas pendientes. De todos ellos, en cada momento sólo hay uno activo, el que está en el tope de la pila.
- Un método puede, durante su ejecución, modificar la información local asociada al mismo sin que, por ello, quede alterada la información local asociada a otras llamadas pendientes, incluso del mismo método. Recuérdese que el paso de parámetros en Java se efectúa siempre por valor.
- Cuando una ejecución de un método finaliza, se liberan los recursos propios de dicha llamada. En particular, deja de existir su registro de activación (se desapila). La ejecución del programa pasa a reanudarse a partir de la

instrucción en la que se efectuó la llamada al método cuya ejecución ha finalizado. Nótese que en el caso recursivo, la ejecución puede reanudarse en una ejecución inmediatamente anterior del mismo método, que habrá dejado de estar pendiente.

- La información se transmite entre las distintas ejecuciones de los métodos a través de los parámetros cuando se efectúa la llamada, o por los valores devueltos cuando se efectúa el retorno.

Todo esto acarrea un uso intensivo de la pila de llamadas que, en ocasiones, crece de manera extraordinaria y puede llegar a provocar serios problemas por agotamiento de la memoria, produciéndose un *desbordamiento de la pila* (en inglés, *stack overflow*). La causa habitual del desbordamiento de la pila es la recursión infinita. Si un método realiza infinitas llamadas recursivas, en algún momento el tamaño de los registros de activación apilados llegará a sobrepasar el tamaño de la pila, provocando la excepción `StackOverflowError`.

Un ejemplo de cómo se comporta la pila para la ejecución del método `factorial` presentado previamente se muestra en la figura 11.2. Para facilitar la discusión, se ha reescrito el código anterior de la siguiente forma:

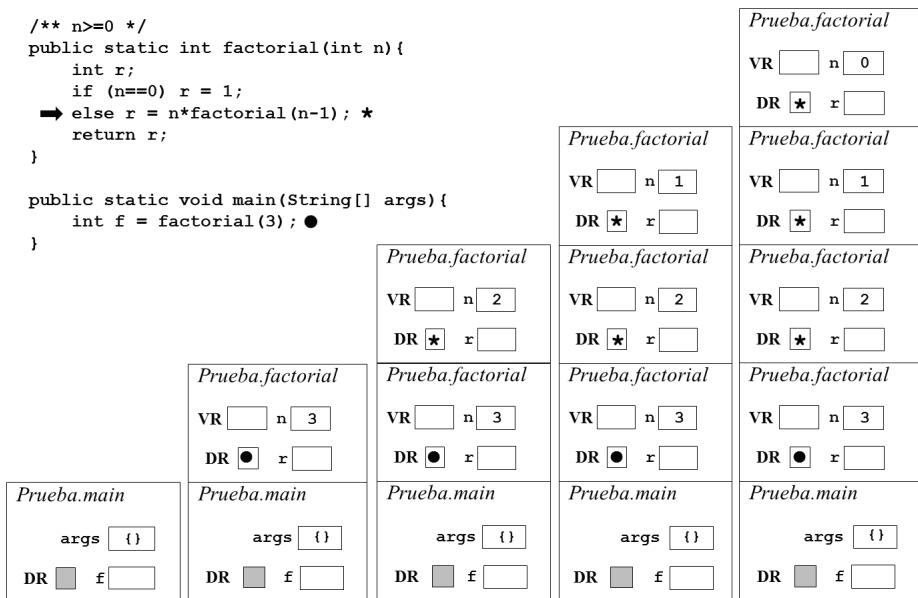
```
/** n>=0 */
public static int factorial(int n) {
 int r;
 if (n==0) r = 1;
 else r = n * factorial(n-1);
 return r;
}
```

En la figura 11.2(a) se muestran los estados de la pila tras cada llamada al método `factorial`. Se puede observar como va creciendo la pila al apilarse un registro de activación por cada llamada al método `factorial`. El primer estado se corresponde con el registro de activación del método `main`, justo antes de la llamada a `factorial(3)`. A continuación, cada columna representa la secuencia de registros de activación cada vez que se detiene la ejecución del método debido a su punto de ruptura (representado por ➞). En el último estado, el registro activo (el de la cima de la pila) se corresponde a la llamada a `factorial(0)`, es decir, la del caso base.

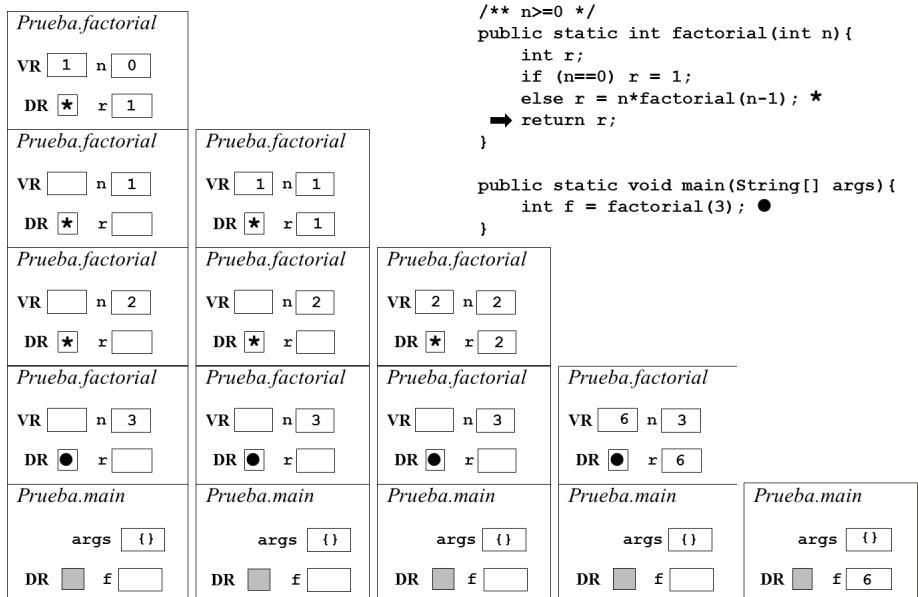
En la figura 11.2(b) se muestran los estados de la pila justo antes de devolver el resultado de cada llamada a `factorial`. La pila va decreciendo al ir desapilándose los registros de activación a medida que se van resolviendo las llamadas al método `factorial`, hasta llegar al estado en el que el único registro de la pila es el correspondiente al método `main` y se ha obtenido el resultado de `factorial(3)`.

```
/** n>=0 */
public static int factorial(int n){
 int r;
 if (n==0) r = 1;
 else r = n*factorial(n-1); *
 return r;
}

public static void main(String[] args){
 int f = factorial(3); ●
}
```



(a) Estados de la pila tras cada llamada a **factorial** hasta alcanzar el caso base.



(b) Estados de la pila justo antes de devolver el resultado de cada llamada a factorial.

**Figura 11.2:** Evolución de la ejecución y la pila de llamadas.

Si se compara el uso de la pila que provoca la llamada `factorial(n)` en la versión iterativa y recursiva del método, se puede concluir que dicho uso es mayor en el caso recursivo que en el iterativo. En la pila de llamadas de la versión iterativa de `factorial` coexisten simultáneamente en memoria, como mucho, el registro de activación del método `factorial` y el registro de activación del método `main`. En la pila de llamadas de la versión recursiva de `factorial` pueden llegar a coexistir simultáneamente  $n+1$  registros de activación de las distintas llamadas a `factorial` más el registro de activación del método `main`. Es decir, el consumo de memoria del método `factorial` iterativo es siempre el mismo mientras que el del `factorial` recursivo depende del valor de `n`.

## 11.4 Algunos ejemplos

Por simplicidad, los métodos que se presentan en esta sección son métodos estáticos; pero un método recursivo puede ser también un método de instancia.

### Potencia $n$ -ésima

Dado un número natural  $n \geq 0$  y un número real  $a \neq 0$ , la potencia  $a^n$  puede definirse de forma recursiva mediante la recurrencia:

- Si  $n = 0$ , entonces  $a^n = 1$ .
- Cuando  $n > 0$ , entonces  $a^n = a^{n-1} * a$ .

Con esta recurrencia, un posible método recursivo que lo soluciona sería:

```
/** n>=0 y a!=0 */
public static double potencia(double a, int n) {
 if (n==0) return 1;
 else return potencia(a,n-1) * a;
}
```

La condición del caso base es  $n == 0$  y la del caso general es la condición contraria (de ahí que se use el `else`). La acción del caso base consiste en devolver el valor 1. La acción del caso general consiste en devolver el resultado de la llamada recursiva multiplicado por  $a$  (es decir, es una recursión lineal no final), cambiando el valor del segundo parámetro por  $n - 1$ . Esto hace que en cada llamada el valor del segundo parámetro vaya decreciendo y se garantiza que en algún momento llegará a ser 0, alcanzando el caso base y finalizando el algoritmo. Esta prueba informal de terminación debe demostrarse matemáticamente para garantizar la finitud del algoritmo, al igual que debe hacerse con su corrección.

A continuación, se muestra una traza de la ejecución del método para calcular  $2^4$ .

| Llamadas ejecutadas | Orden de ejecución | Orden de finalización | Resultado              |
|---------------------|--------------------|-----------------------|------------------------|
| potencia(2,4)       | 1                  | 5                     | potencia(2,3) * 2 = 16 |
| potencia(2,3)       | 2                  | 4                     | potencia(2,2) * 2 = 8  |
| potencia(2,2)       | 3                  | 3                     | potencia(2,1) * 2 = 4  |
| potencia(2,2)       | 4                  | 2                     | potencia(2,0) * 2 = 2  |
| potencia(2,0)       | 5                  | 1                     | 1                      |

Al tratarse de una recursión lineal no final, cuando una llamada recursiva finaliza, se reanuda la ejecución en el punto del método en el que se efectuó la llamada, realizándose aún operaciones posteriores en dicho método para poder devolver el resultado y finalizar.

## Resto de la división entera

Dados dos números naturales  $a \geq 0$  y  $b > 0$ , el resto de su división entera  $a/b$  puede definirse de forma recursiva mediante la recurrencia siguiente:

- Si  $a < b$ , el resto de  $a/b$  es  $a$ .
- En otro caso, el resto de  $a/b$  es el resto de  $(a - b)/b$ .

El siguiente es un posible método recursivo que implementa esta recurrencia:

```
/** a>=0 y b>0 */
public static int resto(int a, int b) {
 if (a<b) return a;
 else return resto(a-b,b);
}
```

La condición del caso base es  $a < b$  y la del caso general es la contraria. La acción del caso base consiste en devolver el valor  $a$ . La acción del caso general consiste en devolver el resultado de la llamada recursiva (es decir, es una recursión lineal final), cambiando el valor del primer parámetro a  $a - b$ . Así, en cada llamada el valor del primer parámetro va decreciendo y se garantiza que en algún momento será inferior al segundo parámetro, alcanzando el caso base y finalizando el algoritmo. Esta prueba informal de terminación debe demostrarse matemáticamente para garantizar la finitud del algoritmo, al igual que debe hacerse con su corrección.

A continuación, se muestra una traza de la ejecución del método para los valores 75 y 18.

| Llamadas ejecutadas | Orden de ejecución | Orden de finalización | Resultado |
|---------------------|--------------------|-----------------------|-----------|
| resto(75,18)        | 1                  | 5                     | 3         |
| resto(57,18)        | 2                  | 4                     | 3         |
| resto(39,18)        | 3                  | 3                     | 3         |
| resto(21,18)        | 4                  | 2                     | 3         |
| resto(3,18)         | 5                  | 1                     | 3         |

Al tratarse de una recursión lineal final, cuando la ejecución de una llamada al método devuelve un valor como resultado (caso base), entonces dicho valor es también el de retorno del resto de llamadas (ya que sobre ninguno de ellos se efectúa otra operación posterior).

## Algoritmo de Euclides

En el capítulo 9, se presentaron dos versiones iterativas (ejemplos 9.4 y 9.5) del algoritmo de Euclides para el cálculo del máximo común divisor (m.c.d.) de dos números naturales  $a$  y  $b$  mayores que cero. La estrategia de resolución planteada en la segunda versión (ejemplo 9.5) se puede seguir para describir el algoritmo de Euclides de forma recursiva según la siguiente recurrencia:

- Si el resto de  $a/b$  es 0, el m.c.d. es  $b$ .
- En otro caso, el m.c.d. es el m.c.d. de  $b$  y el resto de  $a/b$ .

Por tanto, una posible implementación de esta recurrencia es:

```
/** a>0 y b>0 */
public static int euclides(int a, int b) {
 if (a%b==0) return b;
 else return euclides(b,a%b);
}
```

La condición del caso base es  $a \% b = 0$ , siendo el caso general el opuesto. En el caso base se devuelve como resultado el valor del segundo parámetro. En el caso general se devuelve el resultado de la llamada recursiva poniendo como primer parámetro  $b$  y como segundo parámetro  $a \% b$ ; esto garantiza que el segundo parámetro va siempre decreciendo (pues  $a \% b \in [0, b - 1]$ ) y se va acercando a la condición del caso base, pues en algún momento llegará como mínimo a valer 1 y en ese caso  $a \% b = 0$ . Es de nuevo un caso de recursión lineal final.

A continuación, se muestra una traza de la ejecución del método para los valores 152 y 247.

| Llamadas ejecutadas            | Orden de ejecución | Orden de finalización | Resultado |
|--------------------------------|--------------------|-----------------------|-----------|
| <code>euclides(152,247)</code> | 1                  | 7                     | 19        |
| <code>euclides(247,152)</code> | 2                  | 6                     | 19        |
| <code>euclides(152,95)</code>  | 3                  | 5                     | 19        |
| <code>euclides(95,57)</code>   | 4                  | 4                     | 19        |
| <code>euclides(57,38)</code>   | 5                  | 3                     | 19        |
| <code>euclides(38,19)</code>   | 6                  | 2                     | 19        |
| <code>euclides(19,19)</code>   | 7                  | 1                     | 19        |

## Sucesión de Fibonacci

La sucesión de Fibonacci fue definida por el matemático italiano Leonardo de Pisa, conocido como Fibonacci, que introdujo en Europa el sistema de numeración indo-árabigo actualmente utilizado. Dicha sucesión es la siguiente:

$$0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \dots$$

De manera informal, cada término de la sucesión de Fibonacci se define a partir de la suma de sus dos términos predecesores en la misma sucesión. Los dos términos iniciales, al no tener predecesores suficientes, se definen con un valor de 0 y 1, respectivamente. Por tanto, la sucesión de Fibonacci admite la siguiente formulación recursiva para el término  $n$ -ésimo de la sucesión (asumiendo que el término 0-ésimo es el primer término de la sucesión):

- Si  $n \leq 1$ , el valor del término  $n$ -ésimo es  $n$ .
- En otro caso, es la suma de los términos  $(n - 1)$ -ésimo y  $(n - 2)$ -ésimo.

Con esta definición, el método recursivo que puede plantearse para encontrar el término  $n$ -ésimo de la sucesión de Fibonacci es:

```
/** n>=0 */
public static int fibonacci(int n) {
 if (n<=1) return n;
 else return fibonacci(n-1) + fibonacci(n-2);
}
```

La condición del caso base es  $n \leq 1$ , siendo la del caso general la condición opuesta. A diferencia de los ejemplos previos, este método presenta en el caso general dos llamadas recursivas (la de  $(n - 1)$  y la de  $(n - 2)$ ), y el resultado devuelto se

construye a partir de la suma de los resultados de las dos llamadas; se trata, por tanto, de una recursión de tipo múltiple. En cada llamada se va tendiendo a términos inferiores de la sucesión, por lo que se garantiza (informalmente) que en algún momento se cumplirán las condiciones asociadas al caso base.

Se muestra, a continuación, una traza del método para calcular el término 4-ésimo de la sucesión de Fibonacci.

| Llamadas ejecutadas       | Orden de ejecución | Orden de finalización | Resultado                                    |
|---------------------------|--------------------|-----------------------|----------------------------------------------|
| <code>fibonacci(4)</code> | 1                  | 9                     | <code>fibonacci(3) + fibonacci(2) = 5</code> |
| <code>fibonacci(3)</code> | 2                  | 5                     | <code>fibonacci(2) + fibonacci(1) = 3</code> |
| <code>fibonacci(2)</code> | 3                  | 3                     | <code>fibonacci(1) + fibonacci(0) = 2</code> |
| <code>fibonacci(1)</code> | 4                  | 1                     | 1                                            |
| <code>fibonacci(0)</code> | 5                  | 2                     | 1                                            |
| <code>fibonacci(1)</code> | 6                  | 4                     | 1                                            |
| <code>fibonacci(2)</code> | 7                  | 8                     | <code>fibonacci(1) + fibonacci(0) = 2</code> |
| <code>fibonacci(1)</code> | 8                  | 6                     | 1                                            |
| <code>fibonacci(0)</code> | 9                  | 7                     | 1                                            |

En la figura 11.3 se muestra el árbol de llamadas al método `fibonacci` generadas por la llamada inicial `fibonacci(4)`, el orden en el que se realizan y el orden en el que finalizan dichas llamadas. Se puede observar cómo se repiten los mismos cálculos en llamadas recursivas idénticas. Por ejemplo, el subárbol generado por la llamada `fibonacci(2)` está duplicado. Para llamadas a `fibonacci` con valores de `n` más grandes (por ejemplo, para `fibonacci(30)`), la repetición de los cálculos será más frecuente, suponiendo un mayor consumo de tiempo y de memoria (por ejemplo, calcular `fibonacci(20)` requiere 21.891 llamadas y calcular `fibonacci(30)` requiere 2.692.537 llamadas).

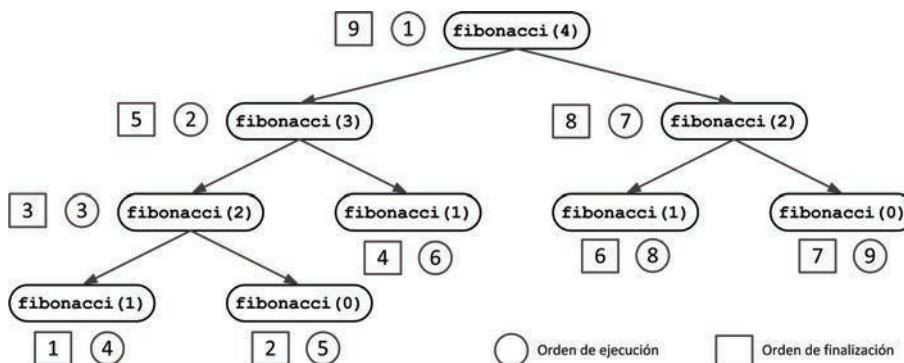


Figura 11.3: Árbol de llamadas del método recursivo `fibonacci`.

## 11.5 Recursión con arrays: recorrido y búsqueda

Todos los problemas con arrays, clasificados como *problemas de recorrido y búsqueda*, que se han estudiado en el capítulo 10, pueden también resolverse haciendo uso de esquemas recursivos en lugar de iterativos.

Para facilitar la discusión subsiguiente, se supondrá dada la siguiente definición en Java:

```
tipoBase[] a = new tipoBase[num];
```

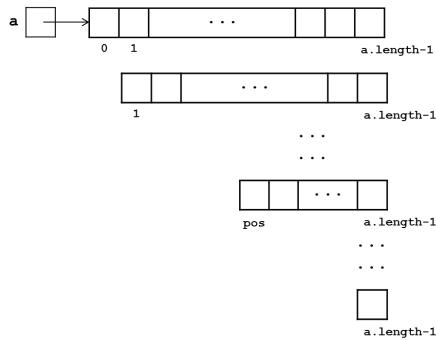
con la que se declara y crea un array **a** de **num** elementos de tipo **tipoBase**. Dada dicha declaración, se puede considerar que el array **a** representa una secuencia de **a.length** elementos de tipo **tipoBase** (**a[0]**, **a[1]**, **a[2]**, ..., **a[a.length-2]**, **a[a.length-1]**), denotada como **a[0..a.length-1]**.

Una secuencia como la anterior se puede definir recursivamente como la secuencia formada por la primera componente de **a**, **a[0]**, y la subsecuencia (que se denomina habitualmente *subarray*) definida por el resto de componentes de **a**, esto es, el array **a** menos su primera componente o subarray **a[1..a.length-1]**. Nótese que, a su vez, el subarray **a[1..a.length-1]** puede definirse recursivamente del mismo modo y así, sucesivamente, hasta que en una última descomposición factible el subarray correspondiente esté vacío, sin componentes. Esta descomposición de un array se denomina *descomposición recursiva ascendente*.

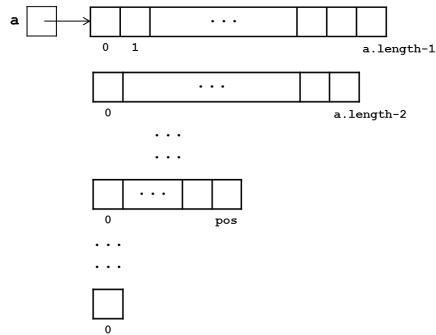
Análogamente, el array **a** se puede definir de forma recursiva considerando el subarray **a[0..a.length-2]** y su última componente **a[a.length-1]**, lo que se denomina *descomposición recursiva descendente*. En las figuras 11.4(a) y 11.4(b) se ilustra gráficamente la descomposición recursiva de un array **a** de manera ascendente y descendente, respectivamente.

Naturalmente, para poder diseñar un método recursivo que realice un recorrido o búsqueda sobre un (sub)array **a** se debe disponer de tres datos o parámetros formales del método: el propio (sub)array **a** y las posiciones que marcan el **inicio** y el **fin** del recorrido o la búsqueda a realizar sobre él en cada llamada. Estos tres parámetros definen en cada llamada la *talla* del problema o número de componentes del (sub)array **a[inicio..fin]** sobre las que se realiza el recorrido o la búsqueda, que viene dada por la expresión **t = fin-inicio+1**; por ejemplo, si en la llamada inicial **a** a un método recursivo **inicio = 0** y **fin = a.length-1** entonces se explora todo el array **a**, pero si **inicio = (a.length-1)/2** y se tiene que **fin = a.length-1** entonces el subarray de **a** a explorar es su última mitad.

Además de los tres parámetros mencionados, el diseño de un método recursivo exige determinar el tipo de descomposición recursiva que se realiza, ascendente o descendente, pues tanto el caso base como la forma de progresar hacia éste



(a) Ascendente



(b) Descendente.

**Figura 11.4:** Descomposición recursiva de un array  $a$ .

en el caso general varían según se elija uno u otro. Específicamente, en una descomposición ascendente de un array  $a$  el caso base se alcanza cuando `inicio = fin+1`, es decir, para una talla  $t = 0$ , y la forma de progresar hacia él en cada llamada es incrementar el valor de `inicio`, con lo que la talla  $t$  decrece en una unidad; así, el rango de variación de `inicio` será  $0 \leq inicio \leq a.length$ , mientras que `fin` mantiene constante su valor en la llamada inicial, `fin = a.length-1`. Sin embargo, en una descomposición descendente, el caso base se alcanza cuando `fin = inicio-1` y la forma de progresar hacia él en cada llamada será decrementar el valor de `fin`; así, el rango de variación de `fin` será  $-1 \leq fin \leq a.length-1$ , mientras que `inicio` mantiene constante su valor en la llamada inicial, `inicio = 0`.

A continuación, se presentan e instancian los esquemas recursivos de recorrido y búsqueda sobre un array. Cabe señalar que, por simplicidad, ambos esquemas se presentan como métodos estáticos y que, como se verá en los ejemplos, cuando se

definan como métodos de instancia de un objeto con un array como atributo de instancia, dicho array no será necesario como parámetro formal del método.

### 11.5.1 Esquemas recursivos de recorrido

En base a la definición de recorrido de un array `a` y la descomposición recursiva ascendente de `a`, el *esquema recursivo de recorrido ascendente* del array `a` desde una posición `izq` hasta una posición `der`,  $0 \leq izq \leq der < a.length$ , es el siguiente:

```
/** 0<=inicio<=der+1 y fin=der */
public static void recorrer(tipoBase[] a, int inicio, int fin) {
 if (inicio>fin) tratarVacio();
 else {
 tratar(a[inicio]);
 recorrer(a,inicio+1,fin);
 }
}
```

donde `tratarVacio()` indica la operación a realizar para un (sub)array sin elementos y `tratar(a[inicio])` indica la operación a realizar con el elemento que ocupa la posición `inicio` del array; siendo la primera llamada o *llamada inicial* `recorrer(a, izq, der)`, esto es, inicialmente `inicio = izq` y `fin = der`. Nótese que si se trata de un recorrido de todos los elementos del array `a`, en la llamada inicial `inicio = 0` y `fin = a.length-1`, es decir, la talla inicial es `t = a.length`. En este caso, es habitual definir un método público homónimo, denominado *guía* o *lanzadera*, que realiza la llamada inicial, con el fin de ocultar la estructura recursiva del array `a` que muestran los parámetros `inicio` y `fin` de la cabecera del método recursivo `recorrer` anterior que ahora se define privado.

```
public static void recorrer(tipoBase[] a) {
 recorrer(a,0,a.length-1);
}
```

El esquema presentado se puede simplificar sustituyendo cualquier referencia al parámetro formal `fin` por su valor en la llamada inicial `der`, como sigue:

```
/** 0<=inicio<=der+1 */
public static void recorrer(tipoBase[] a, int inicio) {
 if (inicio==der+1) tratarVacio();
 else {
 tratar(a[inicio]);
 recorrer(a,inicio+1);
 }
}
```

El *esquema recursivo de recorrido descendente* es el siguiente:

```
/** inicio=izq y izq-1<=fin<a.length */
public static void recorrer(tipoBase[] a, int inicio, int fin) {
 if (fin<inicio) tratarVacio();
 else {
 tratar(a[fin]);
 recorrer(a,inicio,fin-1);
 }
}
```

donde `tratarVacio()` indica la operación a realizar para un (sub)array sin elementos y `tratar(a[fin])` indica la operación a realizar con el elemento que ocupa la posición `fin` del array; siendo la llamada inicial `recorrer(a,izq,der)`, esto es, inicialmente `inicio = izq` y `fin = der`. Nótese que, al igual que en el recorrido ascendente, si se trata de un recorrido de todos los elementos del array `a`, en la llamada inicial `inicio = 0` y `fin = a.length-1`, es decir, la talla inicial es `t = a.length`; pudiéndose definir también, en este caso, un método guía idéntico al del esquema ascendente.

La versión simplificada en la que `inicio` desaparece como parámetro formal, sería:

```
/** izq-1<=fin<a.length */
public static void recorrer(tipoBase[] a, int fin) {
 if (fin==izq-1) tratarVacio();
 else {
 tratar(a[fin]);
 recorrer(a,fin-1);
 }
}
```

En general, la elección del esquema ascendente o descendente, o de la descomposición recursiva subyacente del array, debe realizarse del modo que mejor sirva a los propósitos del diseño ya que forma parte de éste y se basa en el análisis de casos que se realiza para resolver de la manera más natural y eficaz posible cada problema particular que se plantee; los ejemplos que siguen servirán para ilustrarlo.

**Ejemplo 11.1.** En la clase `SecuenciaDeCírculos` (figura 10.19 del capítulo 10) se definió un método iterativo `área` para calcular la suma de las áreas de todos los círculos de una secuencia de círculos; dicha secuencia se representaba mediante un array de `Círculo` (`elArray`) y un valor entero indicando el número de círculos (`talla`).

Considérese ahora el mismo problema pero en su versión recursiva. Para resolverlo, se siguen a continuación los pasos señalados en la sección 11.1 para abordar el diseño de un algoritmo recursivo:

1. Enunciado del problema. Se puede replantear el problema inicial de forma que se haga referencia explícita a la estructura recursiva del array que representa la secuencia de círculos en los términos siguientes: “determinar la suma de las áreas de todos los círculos del array `elArray[0..talla-1]`”. Siguiendo el esquema general recursivo de recorrido ascendente, se puede establecer como cabecera del método la siguiente:

```
/** 0<=inicio<=talla y fin=talla-1 */
private double area(int inicio, int fin) {
 double resMetodo, resLlamada;
 ...
 return resMetodo;
}

// area(inicio, fin) == $\sum_{i = \text{inicio}}^{\text{fin}} \text{elArray}[i].\text{area}()$
```

donde, a diferencia del esquema general, se puede observar, por una parte, que al tratarse de un método de instancia no es necesario pasar `elArray` como parámetro y, por otra parte, que devuelve un resultado explícito de tipo `double`.

El método guía público que lo lanza sería:

```
public double area() { return area(0,talla-1); }
```

2. Análisis de casos:

- Caso base. Como la suma de las áreas de cero círculos es cero, si `elArray` está vacío o su talla es 0, `inicio>fin` y `tratarVacio()` es `resMetodo = 0;`
- Caso general. Cuando `inicio≤fin`, la suma de las áreas de los círculos del subarray `elArray[inicio..fin]` se puede expresar como la suma del área de su primer círculo `elArray[inicio].area()` y la de las áreas de los círculos del subarray `elArray[inicio+1..fin]`. Equivalentemente,

```
resLlamada = area(inicio+1,fin);
resMetodo = elArray[inicio].area() + resLlamada;
```

3. Transcripción del algoritmo a un método en Java, en el que no se utilizará el parámetro `fin`, pues queda claro tras el análisis de casos que no es necesario:

```
/** 0<=inicio<=talla */
private double area(int inicio) {
 double resMetodo, resLlamada;
 if (inicio==talla) resMetodo = 0;
 else {
 resLlamada = area(inicio+1);
 resMetodo = elArray[inicio].area() + resLlamada;
 }
 return resMetodo;
}
// area(inicio) == $\sum_{i = \text{inicio}}^{talla-1} \text{elArray}[i].area()$
```

Alternativamente, escrito de forma más compacta se tendría:

```
private double area(int inicio) {
 if (inicio==talla) return 0;
 else return elArray[inicio].area() + area(inicio+1);
}
```

y su método guía público sería el siguiente:

```
public double area() { return area(0); }
```

4. Validación del diseño. Al examinar el código se observa lo siguiente:

- en el caso general, en cada llamada la talla del problema decrece una unidad porque `inicio` se incrementa en una unidad; por tanto, como antes de efectuarse la llamada `inicio<talla`, tras incrementarse puede llegar a alcanzar el valor `talla`;
- en cualquiera de los dos casos establecidos, el valor del parámetro formal siempre cumple la precondition: `inicio` toma valores en el rango `[0..talla]` porque su valor en la llamada inicial, `area(0)`, se incrementa siempre en uno hasta llegar en el caso base a `talla`.

**Ejemplo 11.2.** Se quiere añadir un método recursivo a la clase `SecuenciaDeCírculos` que muestre por pantalla en orden inverso el área de todos los círculos de la secuencia. La forma más natural de resolver este problema será basarse en una descomposición descendente del array y seguir el esquema general recursivo de recorrido descendente.

1. Enunciado del problema. Para hacer referencia explícita a la estructura recursiva del array que representa la secuencia de círculos, se replantea el problema como sigue: “mostrar por pantalla en orden inverso el área de todos los círculos del array `elArray[0..talla-1]`”. Siguiendo el esquema general

recursivo de recorrido descendente, se puede establecer como cabecera del método la siguiente:

```
/** inicio=0 y -1<=fin<talla */
private void mostrarArea(int inicio, int fin)
```

donde, a diferencia del esquema general, se puede observar que, al tratarse de un método de instancia, no es necesario pasar `elArray` como parámetro.

El método guía público que lo lanza sería:

```
public void mostrarArea() {
 return mostrarArea(0,talla-1);
}
```

## 2. Análisis de casos:

- Caso base. Como no hay ningún círculo, no hay ningún área que mostrar, si `elArray` está vacío o su talla es 0, `fin<inicio` y `tratarVacio() == ;`
- Caso general. Cuando `fin≥inicio`, mostrar por pantalla en orden inverso el área de los círculos del subarray `elArray[inicio..fin]` se puede expresar como mostrar por pantalla el área de su último círculo `elArray[fin].area()` y las áreas de los círculos del subarray `elArray[inicio..fin-1]`. Equivalentemente,

```
System.out.println(elArray[fin].area());
mostrarArea(inicio,fin-1);
```

## 3. Transcripción del algoritmo a un método en Java, en el que no se utilizará el parámetro `inicio`, pues queda claro tras el análisis de casos que no es necesario:

```
/** -1<=fin<talla */
private void mostrarArea(int fin) {
 if (fin===-1) ;
 else {
 System.out.println(elArray[fin].area());
 mostrarArea(fin-1);
 }
}
```

Alternativamente, escrito de forma más compacta se tendría:

```
private void mostrarArea(int fin) {
 if (fin>=0) {
 System.out.println(elArray[fin].area());
 mostrarArea(fin-1);
 }
}
```

y su método guía público sería el siguiente:

```
public void mostrarArea() { return mostrarArea(talla-1); }
```

4. Validación del diseño. Al examinar el código se observa lo siguiente:

- en el caso general, en cada llamada la talla del problema decrece una unidad porque `fin` se decrementa en una unidad; por tanto, como antes de efectuarse la llamada  $\text{fin} \geq 0$ , tras decrementarse puede llegar a alcanzar el valor -1;
- en cualquiera de los dos casos establecidos, el valor del parámetro formal siempre cumple la precondición: `fin` toma valores en el rango  $[-1..talla-1]$  porque su valor en la llamada inicial, `mostrarArea(talla-1)`, se decrementa siempre en uno hasta llegar en el caso base a -1.

### 11.5.2 Esquemas recursivos de búsqueda

Para obtener un esquema recursivo de búsqueda se puede plantear el siguiente análisis de casos: sea  $a[\text{inicio}..\text{fin}]$ ,  $0 \leq \text{inicio} \leq a.length$  y  $-1 \leq \text{fin} < a.length$ , el subarray de talla  $t = \text{fin}-\text{inicio}+1$  donde se busca la posición de un elemento que cumpla una cierta propiedad; si  $t = 0$ , es decir, el subarray está vacío, obviamente ningún elemento cumple la propiedad y el resultado de `buscar` en su caso base es -1; si por el contrario  $t > 0$ , esto es, el subarray contiene al menos una componente, en base a una descomposición ascendente de éste, bien  $a[\text{inicio}]$  cumple la propiedad y el resultado de `buscar` es `inicio` o bien  $a[\text{inicio}]$  no la cumple y se deberá continuar buscando en el subarray  $a[\text{inicio}+1..\text{fin}]$ . En base a este análisis, se propone el siguiente *esquema recursivo de búsqueda ascendente* de la posición de un elemento que cumpla una cierta propiedad en un array `a` desde una posición `izq` hasta una posición `der`,  $0 \leq izq \leq der < a.length$ :

```
/** 0<=inicio<=der+1 y fin=der */
public static int buscar(tipoBase[] a, int inicio, int fin) {
 int resMetodo = -1;
 if (inicio<=fin)
 if (propiedad(a[inicio])) resMetodo = inicio;
 else resMetodo = buscar(a,inicio+1,fin);
 return resMetodo;
}
```

donde `propiedad(a[inicio])` comprueba si el elemento que ocupa la posición `inicio` del array cumple la propiedad enunciada.

Nótese cómo el análisis de casos realizado se refleja en su diseño: por motivos puramente sintácticos del lenguaje Java `resMetodo` se inicializa en cada llamada a

-1, el resultado de la búsqueda para el caso base; dicho valor sólo se modifica si el caso que expresan los parámetros de la llamada es distinto del base, es decir, si se busca un elemento que cumpla la propiedad en un subarray donde `inicio≤fin`. Además, al tratarse de una búsqueda también en el caso general se puede obtener el resultado del método sin efectuar llamada: como ya se ha indicado, si `a[inicio]` cumple la propiedad el resultado es `inicio`; sino, se debe seguir recorriendo `a` en busca de algún elemento que la cumpla –recuérdese que en el peor de los casos, cuando ningún elemento cumple la propiedad, cualquier búsqueda se transforma en recorrido–.

A partir de este esquema se pueden obtener fácilmente los esquemas recursivos de búsqueda ascendente simplificada (en la que se obvia el parámetro `fin`), descendente (basada en la descomposición descendente del array `a`) y descendente simplificada; basta con seguir los pasos y premisas del recorrido recursivo.

Para concluir con la presentación de los esquemas recursivos y con el fin de subrayar una vez más que la elección de un tipo de esquema ascendente o descendente no es en general arbitraria, se aborda a continuación el diseño de un método recursivo que obtenga la posición de la última aparición de un dato dado en un array.

**Ejemplo 11.3.** Se quiere añadir un método recursivo a la clase `SecuenciaDeCírculos` que obtenga la posición de la última aparición de un círculo de color `col` de la secuencia.

Tras una breve reflexión sobre el problema enunciado es fácil concluir que éste admite, al menos, los dos tipos de soluciones siguientes:

- en base a una descomposición recursiva *ascendente* de `elArray` no cabe más que plantear la solución requerida como un *recorrido* del subarray `elArray[inicio..fin]`, donde para cada elemento visitado se debe comprobar si es el último círculo del subarray cuyo color coincide con el dado;
- en base a una descomposición recursiva *descendente* de `elArray` la solución requerida es una *búsqueda* de la primera aparición de un círculo de color `col` en el subarray `elArray[inicio..fin]`.

Aunque ambos tipos de soluciones son correctas, como se verá en el capítulo 12, el criterio de eficiencia es el que lleva a elegir una u otra y, por tanto, el tipo de descomposición recursiva del array sobre el que se efectuará el diseño. Así, la solución óptima es la de búsqueda que, en su peor caso, tiene que visitar todos los elementos del array como si de un recorrido se tratara y, en su mejor caso, en la llamada inicial, inmediatamente obtiene la solución.

Una vez elegido el tipo de solución, se refinará hasta convertirse en un método ejecutable Java siguiendo las siguientes etapas:

1. Enunciado del problema. Se replantea el problema como sigue: “obtener la posición de la última aparición de un círculo de color `col` en el array `elArray[0..talla-1]`”. Siguiendo el esquema general recursivo de búsqueda descendente, se puede establecer como cabecera del método la siguiente:

```
/** inicio=0 y -1<=fin<talla */
private int ultimaP(String col, int inicio, int fin) {
 int resMetodo;
 ...
 return resMetodo;
}
```

Su método guía, público, sería el siguiente:

```
public int ultimaP(String col) {
 return ultimaP(col,0,talla-1);
}
```

2. Análisis de casos.

- Caso base. Si `elArray` es un array vacío obviamente no hay ningún círculo de color `col`. Por tanto, el caso base de `ultimaP` es aquel en que `fin<inicio` y tiene como resultado `resMetodo = -1`;
- Caso general. Cuando `fin≥inicio`, en base a una descomposición descendente de `elArray`, bien `elArray[fin]` es la primera aparición de un círculo de color `col` en `elArray` –por lo que el resultado de `ultimaP` es `fin`– o bien no lo es y se deberá continuar con su búsqueda en `elArray[inicio..fin-1]`. Equivalentemente,

```
if (elArray[fin].getColor().equals(col))
 resMetodo = fin;
else resMetodo = ultimaP(col,inicio,fin-1);
```

3. Transcripción del algoritmo a un método en Java, en el que no se utilizará el parámetro `inicio` pues queda claro tras el análisis de casos que no es necesario:

```
private int ultimaP(String col, int fin) {
 int resMetodo = -1;
 if (fin>-1)
 if (elArray[fin].getColor().equals(col))
 resMetodo = fin;
 else resMetodo = ultimaP(col,fin-1);
 return resMetodo;
}
```

Su método guía público sería:

```
public int ultimaP(String col) {
 return ultimaP(col,talla-1);
}
```

4. Validación del diseño. Al examinar el código se observa que:

- en el caso general, en cada llamada la talla del problema decrece una unidad porque se decrementa `fin`; por tanto, como antes de producirse la llamada `fin>-1`, tras decrementarse en uno puede llegar a alcanzar el valor -1;
- en cualquiera de los dos casos establecidos, el valor del parámetro formal siempre cumple la precondición: `fin` toma valores en el rango `[-1..talla-1]` porque su valor en la llamada inicial, `ultimaP(col,talla-1)`, se decrementa siempre en uno hasta llegar en el caso base al valor -1.

## 11.6 Recursión versus iteración

Al comparar recursión con iteración, se observa que presentan algunas similitudes:

- Tanto recursión como iteración hacen uso de estructuras de control: la recursión usa como instrucción principal una instrucción de selección (condicional) y la iteración usa como instrucción principal una instrucción de repetición (bucle).
- Ambas requieren una condición de terminación: la condición del caso base en la recursión y la condición de la guarda del bucle en la iteración.
- Ambas se aproximan gradualmente a la terminación: la iteración conforme se acerca al cumplimiento de una condición y la recursión conforme se divide el problema en otros más pequeños.
- Ambas pueden tener (por error) una ejecución potencialmente infinita.

Se puede demostrar que la solución algorítmica de cualquier problema algorítmicamente resoluble, se puede expresar recursivamente y también iterativamente. En este sentido, se dice que recursión e iteración son equivalentes, y por ello, alternativos.

No es posible afirmar en general qué es lo más conveniente o sencillo. Es frecuente encontrar problemas para los que la solución iterativa es más sencilla de estructurar, y de expresar en el lenguaje de programación, que un equivalente recursivo.

Además, la recursión supone, en general, más carga computacional (espacio en memoria) que la iteración y, a veces, conduce con más facilidad a soluciones redundantes (algunas soluciones recursivas resuelven repetidamente un (sub)problema) que la iteración. Pero, en otras ocasiones, la recursión es la alternativa más sencilla para resolver ciertos problemas cuya formulación iterativa resulta complicada. En estos casos, la versión recursiva refleja de manera más natural, concisa y elegante la solución al problema, lo que hace que sea más fácil de depurar y entender. Además, el planteamiento recursivo de muchos problemas es sólo un primer paso para poder desarrollar una solución iterativa posterior de mayor eficiencia mediante el uso de técnicas algorítmicas avanzadas (como la *programación dinámica*). Cabe también señalar que hay modelos de cómputo y lenguajes de programación cuyo único mecanismo de repetición es la recursión.

Se puede concluir, por lo tanto, que recursión e iteración, además de alternativos, son complementarios.

## 11.7 Problemas propuestos

1. Mostrar la evolución de la pila de llamadas para el siguiente método, suponiendo que la llamada inicial es `fact(5)`. Tomar como referencia las líneas con `*****`

```
/** n>=0 */
public static int fact(int n) {
 int p;

 if (n==0) p = 1;
 else p = n * fact(n-1);

 return p;
}
```

2. Escribir un método de clase recursivo que muestre en pantalla los números naturales del 1 al  $n$ , donde  $n > 0$  es el valor pasado como parámetro en la llamada inicial.
3. Escribir un método de clase recursivo que muestre en pantalla los números naturales del  $n$  al 1, donde  $n > 0$  es el valor pasado como parámetro en la llamada inicial.
4. Escribir un método de clase recursivo que, dados dos números naturales  $a \geq 0$  y  $b > 0$ , calcule el cociente de su división entera, basándose en el hecho de que dicha operación se puede realizar como una serie de restas sucesivas (de forma similar al problema del cálculo del resto de la división entera de la sección 11.4), siguiendo la recurrencia:
  - $a/b = 0$ , si  $a < b$ ,
  - $a/b = (a - b)/b + 1$ , si  $a \geq b$ .
5. Dados dos números enteros  $a$  y  $b$ , siendo  $b \geq 0$ , se puede definir recursivamente su producto  $a \cdot b$  del modo siguiente:
  - $a \cdot b = 0$ , si  $b = 0$ ,
  - $a \cdot b = a \cdot (b - 1) + a$ , si  $b > 0$ .

Nótese que, para esta definición, la multiplicación se reduce a una secuencia de sumas.

Considerando que tan sólo es posible utilizar operaciones aditivas, escribir un método de clase recursivo para realizar la operación pedida, siguiendo la definición anterior.

6. Dados dos números enteros  $a$  y  $b$ , siendo  $b \geq 0$ , otra forma de definir recursivamente su producto  $a \cdot b$  es la siguiente:

- $a \cdot b = 0$ , si  $b = 0$ ,
- $a \cdot b = (a \cdot 2) \cdot (b/2)$ , si  $b > 0$  y  $b$  es par, y
- $a \cdot b = (a \cdot 2) \cdot (b/2) + a$ , si  $b > 0$  y  $b$  es impar.

Este tipo de multiplicación se conoce como *multiplicación a la rusa*, siendo utilizada antiguamente por los comerciantes de dicho país, que no conocían la tabla de multiplicar, para efectuar el producto de dos números positivos cualesquiera utilizando solamente sumas, productos y divisiones por 2.

Considerando que tan sólo se pueden utilizar productos y divisiones por 2, así como sumas, escribir un método de clase recursivo para realizar la operación pedida, siguiendo la definición anterior.

7. Dados dos números enteros  $a \neq 0$  y  $b \geq 0$ , se puede definir recursivamente la potencia  $a^b$  del modo siguiente:

- $a^b = 1$ , si  $b = 0$ ,
- $a^b = a$ , si  $b = 1$ ,
- $a^b = (a^{b/2}) \cdot (a^{b/2})$ , si  $b > 1$  y  $b$  es par, y
- $a^b = (a^{b/2}) \cdot (a^{b/2}) \cdot a$ , si  $b > 1$  y  $b$  es impar.

Considerando que tan sólo se pueden utilizar divisiones por 2 y productos, escribir un método de clase recursivo para realizar la operación pedida, siguiendo la definición anterior.

8. Escribir un método de clase recursivo que devuelva la suma de los dígitos de un número natural  $n \geq 0$  pasado como parámetro.
9. Escribir un método de clase recursivo que devuelva el número de dígitos de un número natural  $n \geq 0$  pasado como parámetro.
10. Escribir un método de clase recursivo que muestre en orden inverso los dígitos que componen un número natural  $n \geq 0$  dado.
11. Escribir un método de clase recursivo que devuelva el valor binario (representado como un entero) de un número natural  $n \geq 0$  dado. Por ejemplo, si  $n=5$  el método devuelve 101, pero si  $n=31$  el método devuelve 11111.

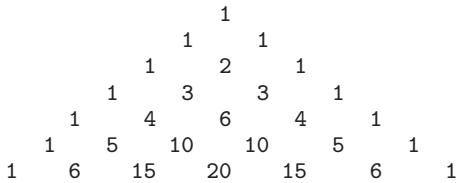
12. Escribir qué se muestra por pantalla al ejecutar este programa:

```
class Raro {
 /** n>=0 */
 public static void escribeRaro(int n) {
 if (n>0) {
 System.out.print(n);
 escribeRaro(n-1);
 System.out.print(n);
 }
 else System.out.print(0);
 }

 public static void main(String[] args) {
 escribeRaro(5);
 }
}
```

13. Escribir un método de clase recursivo que, dados dos **String** *s1* y *s2* y sin hacer uso de los métodos definidos en la clase **String** que resuelven el mismo problema, determine:
- a) si *s2* es prefijo de *s1*.
  - b) si *s2* es sufijo de *s1*.
  - c) si *s2* es una subcadena de *s1*.
14. Escribir un método de clase recursivo que, dados un **String** *s* y su longitud *l*, muestre en orden inverso los caracteres de *s*.
15. Escribir un método de clase recursivo que dé el mismo resultado que el ejercicio previo pero recibiendo sólo la cadena (**String**). Ayuda: usar el método **substring**.
16. Escribir un método de clase recursivo que compruebe si un **String** *s* dado es palíndromo. Ayuda: usar el método **substring** para reducir la longitud de *s* en cada llamada recursiva.

17. En un triángulo de Pascal, como se puede observar en el ejemplo de la siguiente figura, cada elemento es la suma de los dos elementos situados sobre él, excepto el primero y último de cada fila que valen 1.



Escribir un método de clase recursivo que devuelva el  $i$ -ésimo elemento de la fila  $f$  de un triángulo de Pascal; su cabecera será entonces del tipo `public static int trianguloPascal(int f, int i)`.

Además, escribir una clase Java que lea de teclado un número de filas  $y$ , usando el método diseñado, imprima (no necesariamente formando un triángulo) todos los elementos del correspondiente triángulo de Pascal.

18. Escribir qué se muestra por pantalla al ejecutar este programa:

```
class Suma{
 public static int sumav(int[] v, int i, int x) {
 int suma = 0;
 if (i<0) return suma;
 if (v[i]==x) return suma;
 for (int j=0; j<=i; j++)
 suma = suma + v[j];
 System.out.println("Suma parcial: " + suma);
 return suma + sumav(v,i-1,x);
 }

 public static void main(String[] args) {
 int[] v = {1, 2, 3, 4, 5};
 System.out.println("Suma total: " + sumav(v,4,2));
 }
}
```

19. Dado un array de `String`  $v$ , escribir un método de clase recursivo que:
- determine si es capicúa, esto es, si la primera y última palabra del array son la misma, la segunda y la penúltima palabras también lo son, y así sucesivamente. El método retornará `true` si el array es capicúa o `false` en caso contrario.
  - dada una palabra  $pal$ , determine la existencia de dicha palabra en el array entre dos posiciones dadas  $ini$  y  $fin$  que cumplen inicialmente:  $0 \leq ini \leq fin < v.length$ . Caso de existir la palabra, el método devolverá

la primera posición donde se encuentre la misma, y de no existir el método devolverá el valor -1.

20. Indicar qué se muestra por pantalla al ejecutar este código:

```
class Traza{
 public static void main(String[] args) {
 int v[] = {0, 11, 2, 13, 4, 5, 6, 17, 8};
 f(v, 0);
 }

 public static void f(int[] v, int i) {
 if (i>=v.length)
 System.out.println("-----");
 else if (i==v[i]) {
 System.out.printf("v[%d]==%d\n", i, v[i]);
 f(v, i+1);
 }
 else {
 f(v, i+1);
 System.out.printf("v[%d] !=%d\n", i, i);
 }
 }
}
```

21. Escribir la versión recursiva del método de la búsqueda binaria propuesto en el capítulo 10.
22. Dado un array de enteros  $v$ , escribir un método de clase recursivo que:
- Obtenga la suma de todos los elementos del array.
  - Obtenga la posición del máximo (mínimo) del array.
  - Dado un entero  $x$ , cuente cuántas veces aparece en el array.
  - Compruebe si el array está ordenado ascendentemente.
  - Determine la posición del primer (último) elemento no nulo del array.
  - Determine cuántos ceros consecutivos hay al final del array.
  - Dadas dos posiciones,  $izq$  y  $der$ , del array,  $0 \leq izq \leq der \leq v.length-1$ , invierta todos los elementos del array situados entre dichas posiciones, esto es, al finalizar la ejecución del método el array contendrá en su posición  $izq$  el elemento que inicialmente ocupaba la posición  $der$ , en su posición  $izq+1$  el elemento que inicialmente ocupaba la posición  $der-1$  y así sucesivamente.
  - Dadas dos posiciones,  $izq$  y  $der$ , del array,  $0 \leq izq \leq der \leq v.length-1$ , duplique el valor de los elementos del array situados entre dichas posiciones.

- i) Dado un entero  $b > 0$ , determine si  $b$  es igual a la suma de todas las componentes de  $v$ .
- j) Dado un entero  $x$ , determine la cantidad de elementos del array que son menores que  $x$ .
- k) Determine la cantidad de elementos impares que ocupan posiciones pares del array.
- l) Determine la posición, si existe, de la primera subsecuencia del array que comprenda, al menos tres números enteros consecutivos en posiciones consecutivas del array.

## Más información

- [Han99] B. Hansen. *Programming for everyone in Java*. Springer, 1999. Capítulo 10.
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010. Capítulo 11.
- [Sch07] H. Schildt. *Fundamentos de Java*. McGraw-Hill, 2007. Capítulo 6 (6.6).
- [Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulo 7.

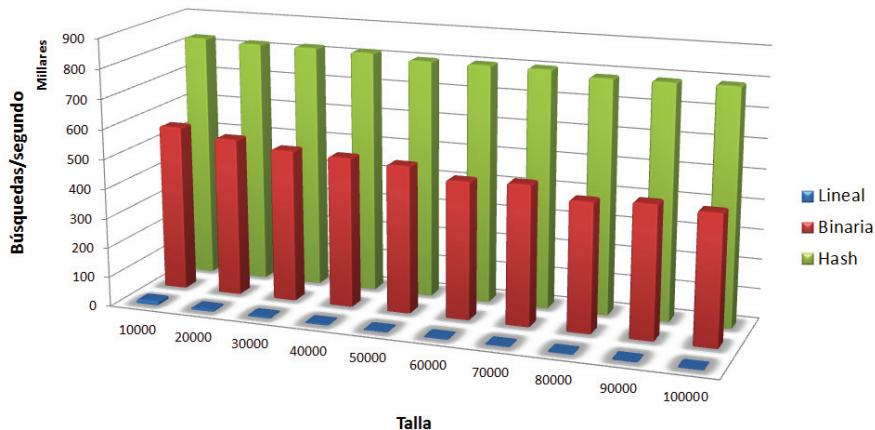
## Capítulo 12

# Análisis de algoritmos

El principal objetivo de los capítulos previos ha sido presentar conceptos y técnicas básicas de programación. En este sentido se han presentado los conceptos algorítmicos y de representación de datos que permiten resolver problemas sencillos; esto es tipos de datos, operadores, arrays y clases o instrucciones para controlar el flujo de un programa, como condicionales, bucles o métodos. Sin embargo, para poder plantear la resolución de problemas de mayor complejidad es necesario introducir la propiedad de la eficiencia. El código desarrollado no sólo debe ser correcto sino también eficiente, entendiendo por eficiente que use la menor cantidad de recursos posibles, recursos que, en este caso, son principalmente tiempo de cómputo y memoria utilizada. La forma de representar los datos y las estrategias algorítmicas utilizadas son las que permiten soluciones más o menos eficientes a un mismo problema.

Por ejemplo, la búsqueda de una o más palabras en un texto o en una colección de documentos puede abordarse de distintas formas, según como se represente el texto (o colección) sobre el que se efectúa la búsqueda. En la figura 12.1 se muestran el número de búsquedas por segundo que se pueden hacer de una palabra en función del tamaño del texto y del método de búsqueda. Se observa que en un texto de 20000 palabras se pueden buscar 6000 palabras por segundo si se realiza una búsqueda secuencial, más de medio millón si la estrategia de búsqueda es binaria y más de 800000 si se busca en una Tabla Hash. Obsérvese que la aproximación más eficiente tiene, en este caso, también la ventaja de que el tiempo de búsqueda permanece prácticamente constante aunque aumente el tamaño del texto en el que se efectúa la búsqueda. Por ejemplo, para una talla de 100000 palabras con una búsqueda lineal sólo se podrían efectuar mil búsquedas mientras que en la aproximación más eficiente se podrían seguir buscando casi 800000. En nuestros días, tan acostumbrados a un uso extensivo de los buscadores, es fácil imaginar lo que esto significa. Otro ejemplo paradigmático es el de la ordenación. Ordenar

un array de datos puede costar entre 20 o 3000 milisegundos<sup>1</sup> según el algoritmo empleado para la ordenación.



**Figura 12.1:** Velocidad de búsqueda según la estrategia utilizada.

En resumen, son muchas las aplicaciones en las que resulta imprescindible la utilización de algoritmos eficientes, tratamiento de imágenes, sistemas de recuperación y extracción de información, sistemas de reconocimiento y síntesis de voz, traductores, cálculo matricial y numérico para la resolución de problemas en diversas ingenierías, etc. Por ello es tan importante conocer los conceptos y técnicas relacionadas con el análisis de la complejidad de los algoritmos, propósito de este capítulo. De este modo se podrán comparar las diferentes soluciones algorítmicas a un problema con el fin de utilizar la más adecuada en cada caso e incluso utilizar la información del coste como una guía en el diseño de nuevas soluciones.

## 12.1 Análisis de algoritmos

La *eficiencia* de un determinado algoritmo o programa viene determinada por la cantidad de recursos que consume al ejecutarse; estos recursos son *tiempo* y *memoria*. El *coste* (*o complejidad*) *temporal* de un algoritmo es una medida del tiempo empleado por éste para ejecutarse. El *coste* (*o complejidad*) *espacial* de un algoritmo es una medida del espacio que ocupa en memoria a lo largo de su ejecución (suma de los tamaños de todas las variables que implícita o explícitamente se utilizan).

<sup>1</sup>Por ejemplo, aproximadamente,  $10^5$  enteros con un ordenador de sobremesa actual (core 2 duo, linux).

Dado un determinado problema y un algoritmo o programa concreto que lo resuelve, los costes empleados van a depender de:

- Factores propios del algoritmo utilizado, como son la estrategia de resolución y las estructuras de datos empleadas.
- Factores que son dependientes del entorno de programación, como son el tipo de computador, el lenguaje de programación utilizado, la carga del sistema, el ancho de banda de los elementos de entrada y salida empleados, etc.

Para estudiar el coste de un determinado algoritmo o programa se puede seguir alguna de estas dos aproximaciones, que en ningún caso son excluyentes, sino que se complementan:

- Análisis teórico o *a priori*. En este análisis se trata de definir el coste de un algoritmo como una función dependiente de ciertos parámetros que influyen en su comportamiento, siendo el parámetro más importante el tamaño del conjunto de datos de entrada. Lo que más adelante se denominará *talla del problema*.

Gracias a este tipo de análisis se puede predecir el coste de los algoritmos y comparar distintas soluciones posibles a un mismo problema.

- Análisis experimental o *a posteriori*. En este tipo de análisis se realizan mediciones concretas del tiempo de ejecución y de la ocupación de memoria. Por lo general se prepara una batería de pruebas con diferentes tamaños de los datos de entrada y, para un mismo tamaño, diferentes combinaciones de dichos datos; esto permite comprobar si la implementación del algoritmo es sensible o no a cómo se disponen los datos de entrada. Habitualmente, la batería de pruebas se realiza sobre un sistema concreto, utilizando una versión del algoritmo implementada en un lenguaje de programación determinado. Aunque, obviamente, se podría obtener un estudio más exhaustivo repitiendo la misma batería de pruebas en distintos sistemas o incluso con implementaciones del algoritmo en estudio en distintos lenguajes.

En cualquier caso, el objetivo final consiste en obtener medidas concretas del tiempo de ejecución para así comprobar si el comportamiento observado coincide con el previsto por el análisis teórico.

## 12.2 El coste temporal y espacial de los programas

En el estudio de un algoritmo concreto, el objetivo final será poder expresar su eficiencia como consecuencia de lo que el algoritmo hace, independizándolo del lugar concreto en el que lo hace. De esta manera será posible, por ejemplo, poder

comparar dos algoritmos que resuelvan un mismo problema, centrando el estudio exclusivamente en lo qué estos dos algoritmos realizan intrínsecamente para resolver el problema.

Para concretar la forma de actuación que se va a presentar para poder independizar este estudio, la discusión y ejemplos que siguen se centrarán principalmente en el coste temporal de los algoritmos. Aun así, mucho de lo que se pueda inferir de esta discusión, será también de aplicación al estudio del coste espacial que se abordará con algunos ejemplos específicos más adelante.

A continuación, se inicia la discusión caracterizando el tiempo de ejecución de algunos algoritmos como función del tiempo de ejecución de las operaciones elementales que estos efectúen. De este estudio preliminar se podrán extraer las conclusiones necesarias para independizar el cálculo del tiempo de ejecución del entorno específico en que este se efectúe.

### 12.2.1 El coste temporal medido en función de los tiempos de las operaciones elementales

Supóngase que se dispone de los siguientes algoritmos para calcular  $n^2$  guardando el resultado en la variable  $m$ :

- *Algoritmo A1:*

```
m = n*n;
```

- *Algoritmo A2:*

```
m = 0;
for (int i=0; i<n; i++) m+=n;
```

- *Algoritmo A3:*

```
m = 0;
for (int i=0; i<n; i++)
 for (int j=0; j<n; j++) m++;
```

Si se define el coste temporal de un algoritmo como la suma de los costes de las operaciones elementales que implica, entonces se tienen los siguientes costes:

- *Coste del algoritmo A1:*  $T_{A1} = t_a + t_{op}$

donde  $t_a$  es el coste de la operación de asignación y  $t_{op}$  es el coste de una operación aritmética, en concreto de una multiplicación.

- *Coste del algoritmo A2:*  $T_{A2} = t_a + t_a + (n + 1) * t_c + n * (t_a + 2 * t_{op})$

donde  $t_c$  es el coste de la comparación y  $t_{op}$  es, en este caso, el coste de la suma, que para simplificar se considera igual al de la multiplicación de A1. Nótese como la comparación de la guarda del bucle  $i < n$  se ejecuta una vez más que el resto de instrucciones del cuerpo del bucle, incluido el autoincremento.

- *Coste del algoritmo A3:*  $T_{A3} = t_a + t_a + (n + 1) * t_c + n * (t_{op} + t_a + (n + 1) * t_c + n * 2 * t_{op})$

Si se quiere comparar los costes de estos algoritmos, resulta complicado hacerlo con las expresiones obtenidas. Pero es que, además, los tiempos de las operaciones individuales pueden variar significativamente en función del entorno de programación utilizado. Nótese que la mayoría de las constantes que aparecen dependen de las características del entorno en el que se está realizando la ejecución (y de las que, por los motivos ya expuestos, se desea independizar en lo posible el estudio).

Por ello, no es necesario distinguir con tanto detalle el coste de cada operación elemental.

Una primera simplificación consiste en independizar las funciones de coste de los tiempos de ejecución de las operaciones elementales, asumiendo que el coste de una o más operaciones elementales consecutivas es constante (representado como una constante  $k$ ). De esta manera, las funciones de coste anteriores pueden simplificarse como sigue:

- *Coste del algoritmo A1:*  $T_{A1} = k_1$   
con esta expresión se indica que el coste del algoritmo A1 es *constante*, no depende de  $n$ .
- *Coste del algoritmo A2:*  $T_{A2} = k_2 * n + k_3$   
en este caso el coste del algoritmo A2 es proporcional a  $n$ , es decir, tiene una dependencia *lineal*.
- *Coste del algoritmo A3:*  $T_{A3} = k_4 * n^2 + k_5 * n + k_6$   
en este caso se obtiene una función de coste que es proporcional al cuadrado de  $n$ , por tanto, se dice que tiene una dependencia *cuadrática*.

Como ya se ha dicho, los valores de las constantes  $k_i$  dependen de la implementación que se haga del algoritmo, del lenguaje de programación y del ordenador sobre el que se ejecute, incluso del sistema operativo utilizado; por ello, devienen irrelevantes para el análisis teórico, de forma que, tal y como se verá a continuación, serán ignoradas.

### 12.2.2 El coste como una función del tamaño del problema. Talla del problema

En los ejemplos anteriores puede verse que los tiempos de ejecución de los algoritmos  $T_{A2}$  y  $T_{A3}$  dependen del valor,  $n$ , que se desea elevar al cuadrado.

En general, es evidente que el tiempo necesario para ejecutar un algoritmo depende casi siempre de la cantidad de datos que tiene que procesar; así, es de esperar que, por ejemplo, ordenar 10000 elementos requiera más tiempo que ordenar 10, del mismo modo que la resolución de un sistema de ecuaciones tardará más a medida que el número de ecuaciones del sistema sea mayor.

En este sentido, se denominará *tamaño* o *talla* de un problema al parámetro o conjunto de parámetros en función de los cuales se expresará el coste necesario para resolverlo. La talla de un problema es, en cierto modo, una medida proporcional al esfuerzo que costará resolverlo, ya que se espera que cuando la misma crezca, también lo haga el tiempo de ejecución y/o la memoria empleados en la resolución.

En el análisis preliminar realizado de los algoritmos  $A1$ ,  $A2$  y  $A3$  para calcular  $n^2$ , la talla del problema era precisamente  $n$ , el valor de cuyo número se quiere obtener su cuadrado. Otros ejemplos evidentes son el número de elementos de un array que se desea ordenar o el valor del número del que quiere calcularse su factorial, etc.

Efectuada la definición de la talla de un problema es posible precisar ahora lo que se entiende por coste temporal y espacial de un algoritmo:

El *coste temporal (espacial)* de un algoritmo se define como una función positiva, discreta y no decreciente que representa la cantidad de tiempo (memoria) que el algoritmo necesita para resolver el problema como función de la talla del problema.

Nótese que, según esta última definición, puede ocurrir que el tiempo de ejecución de un algoritmo dado no aumente aun cuando se incremente la talla (este es el caso del algoritmo anterior  $A1$ <sup>2</sup>) ya que lo único que se exige a la función de coste es que esta no sea decreciente. En esta situación se dice que el algoritmo tiene un coste constante. Los algoritmos de coste constante son significativos ya que representan un límite o cota inferior al coste de cualquier algoritmo; se puede decir que son, desde el punto de vista de su eficiencia, *los mejores*.

---

<sup>2</sup>ya que se asume que, en un ordenador actual, el coste de hacer un producto no depende del valor del número implicado cuando este tiene una representación de tamaño fija como, por ejemplo, 4 bytes para un entero.

### 12.2.3 Paso de programa. El coste temporal definido por conteo de pasos

Una forma de independizar el estudio del coste temporal de un algoritmo de las características del sistema en que este se ejecute, consiste en definir como equivalentes cualquier operación o secuencia de operaciones cuyo tiempo de ejecución no dependa de la talla, esto es, que se efectúe en tiempo constante.

De esta forma, por ejemplo, es posible afirmar que son equivalentes, desde un punto de vista del coste temporal, dos algoritmos que resuelven lo mismo cuando uno de ellos necesita, por ejemplo, el triple de operaciones que el otro. En realidad, lo que se está diciendo aquí, es que puede ocurrir que al trasladar de un sistema a otro distinto la comparación de ambos algoritmos, el tiempo relativo de las operaciones básicas cambia lo suficiente para hacer irrelevante esa diferencia del, por ejemplo, triple de operaciones.

Por ello, cuando el objetivo es independizar el estudio temporal del sistema específico, *se menospreciarán* todas las constantes multiplicativas que puedan diferenciar entre sí a los algoritmos, ya que se entiende que los cambios en el ámbito de ejecución podrían modificarlas y alterar la relación entre las mismas.

Para aplicar lo anterior, se introduce una nueva unidad temporal de ejecución denominada *paso de programa*. Un paso de programa se define como cualquier operación o secuencia de operaciones cuyo coste temporal sea constante (esto es, independiente de la talla). Por ejemplo, considérese el siguiente segmento de código:

```
double a = 10; double b;
b = (a * (a-1))/3;
b = b - a + 1;
```

Según la definición dada, el segmento anterior puede considerarse, por ejemplo, bien como 8 pasos de programa (uno por cada operación individual y uno por cada asignación); bien como 3 o 4 pasos de programa (uno por cada instrucción individual); o incluso como un único paso de programa (ya que toda la secuencia de operaciones es independiente de la talla y se efectúa en tiempo constante).

A tenor del ejemplo, puede parecer que la definición de paso de programa es ambigua ya que permite obtener valores distintos, en unidades temporales de ejecución, para un mismo segmento de código; sin embargo esto no es así ya que, por la definición hecha, el paso de programa hace irrelevantes a las constantes y de ahí que a dicho efecto, sean similares entre sí los valores 8, 3, 1, etc.

Por supuesto, cabe deducir también que, para facilitar en lo posible los cálculos, se considerará que segmentos de código como el anterior tardan en ejecutarse 1 paso de programa.

Considérese ahora el ejemplo:

```
int s = 0;
for (int i=0; i<a.length; i++) s = s + a[i];
double m = s/a.length;
```

Suponiendo que la longitud, `a.length`, del array `a`, sea cierto valor  $n$  (talla del problema) y aplicando una vez más la definición de paso de programa, se pueden hacer las consideraciones siguientes:

- la primera instrucción, de asignación a la variable acumulador `s`, cuesta 1 o 2 pasos de programa,
- el coste de cada una de las  $n$  iteraciones individuales del bucle puede considerarse de distinta forma, por ejemplo, se puede suponer que cuesta 6 pasos (uno por cada operación individual), o también puede pensarse que cuesta 2 pasos (1 por la guarda y otro por la asignación interna), o incluso puede estimarse que supone sólo 1 paso,
- finalmente, utilizando un razonamiento similar, se puede considerar que la última asignación de la secuencia, que se efectúa con posterioridad al bucle, puede costar bien 1, 2 o 3 pasos de programa.

Con todo ello, es posible llegar como expresión total del coste, a un grupo de expresiones, entre las que se encuentran las tres siguientes (nótese que otras expresiones similares, también serían igualmente válidas):

- $6n + 4$  pasos de programa,
- $2n + 2$  pasos de programa,
- $n + 1$  pasos de programa.

Al igual que antes, por razones de sencillez en los cálculos, será la última de las expresiones la que se utilizará habitualmente como expresión del coste temporal.

Gracias a la unidad paso de programa, se puede enunciar ahora el *coste temporal* independizándolo del tiempo de ejecución de las operaciones elementales. Así, se define el *coste temporal* de un algoritmo como el número de pasos de programa que este realiza, en función de la talla del problema.

Aplicando esta definición y teniendo en cuenta consideraciones similares a las visitas, los costes de los algoritmos presentados en el apartado anterior se pueden expresar como:

- *Coste del algoritmo A1:*  $T_{A1}(n) = 1$  paso,
- *Coste del algoritmo A2:*  $T_{A2}(n) = n$  pasos,
- *Coste del algoritmo A3:*  $T_{A3}(n) = n^2 + n$  pasos.

Como ya se ha repetido, la decisión sobre qué se puede considerar un paso de programa en un determinado algoritmo no es única. Aunque tal y como se ha visto, generalmente se elegirán aquellas expresiones que permitan facilitar los cálculos.

En cualquier caso, en el apartado siguiente se verá cómo estas diferencias en las expresiones del coste temporal expresadas en pasos de programa, no tendrán relevancia en el estudio que se va a realizar.

### 12.3 Complejidad asintótica

Uno de los principales objetivos del análisis de algoritmos es comparar las soluciones disponibles para un mismo problema o tipo de problemas. Para decidir con seguridad qué solución es la mejor, o cuáles son las mejores, o si todas las disponibles son equivalentes, es de gran utilidad disponer de un método que permita comparar los algoritmos de la manera más objetiva posible. Lo primero que se puede pensar es en representar en una gráfica las funciones de coste temporal de todos los algoritmos que son solución a un mismo problema. La representación será con respecto a la talla del problema, de manera que se podrá visualizar de una manera clara qué algoritmos son más eficientes y cuáles menos. Y también qué algoritmos serán mejores para tallas del problema pequeñas y cuáles para tallas del problema mayores.

Sin embargo, resulta evidente que cuando la talla del problema es pequeña, aunque un algoritmo tenga un coste muy superior en apariencia a otro, los tiempos de ejecución no mostrarán diferencias significativas. Pero no ocurrirá lo mismo cuando la talla del problema crezca considerablemente, es decir, cuando los algoritmos se ejecuten para conjuntos de datos de entrada de tallas grandes o muy grandes. En general, se ignoran las diferencias entre funciones de coste temporal para tallas pequeñas, dado que a efectos prácticos son generalmente irrelevantes; no se notarán grandes diferencias, por ejemplo, entre un algoritmo *A* que tarda 1 milisegundo y otro *B* que tarda 1.2 milisegundos para valores de  $n = 100$ ; pero sí importará que *A* tarde dos horas frente a los tres minutos de *B* cuando  $n = 10^6$ .

Si fruto del análisis teórico o *a priori* se obtiene una función de coste para cada algoritmo, lo que interesa para comparar los mismos es hacerlo para tallas suficientemente grandes o, lo que es lo mismo, determinar el comportamiento asintótico de estas funciones cuando la talla del problema tiende a infinito (o en el límite).

Como se verá a continuación, a efectos asintóticos, las funciones de coste quedarán caracterizadas por su término de mayor orden, pudiéndose incluso ignorar el resto de términos. Con este razonamiento se refleja el hecho de que a medida que la talla crece, se tiene que el término de mayor orden crece más rápidamente que cualquiera de los de orden menor; llegando los últimos a ser despreciables frente al primero para tallas suficientemente grandes. En el límite, por lo tanto, tan solo se tendrá que tener en cuenta el de orden mayor.

Para el caso particular de los polinomios, lo anterior significa que estarán caracterizados por su monomio de mayor grado. Así, por ejemplo, el coste del algoritmo A3 de la sección anterior,  $T_{A3}(n) = n^2 + n$ , queda caracterizado por el término  $n^2$ . Desde el punto de vista de la complejidad asintótica se dirá que el coste del algoritmo A3 es del orden de  $n^2$ .

### 12.3.1 Comparación de los costes de los algoritmos

Como se ha mencionado ya, el coste de un algoritmo es una expresión  $T(n)$ , positiva y no decreciente, función de la talla del problema. Por lo tanto, comparar costes es comparar funciones no decrecientes de las que interesa sólo su tasa de crecimiento asintótico. Cuando  $T(n)$  es un polinomio, su monomio de mayor grado es el que caracteriza el aspecto de la curva de crecimiento. Puede que los monomios menores determinen el aspecto de la curva para valores pequeños de  $n$  si la constante que los multiplica es relativamente grande en valor absoluto, pero su influencia quedará minimizada para valores grandes de  $n$ . Desde un punto de vista matemático los monomios de menor grado son irrelevantes en el límite, es decir, cuando  $n$  tiende a infinito.

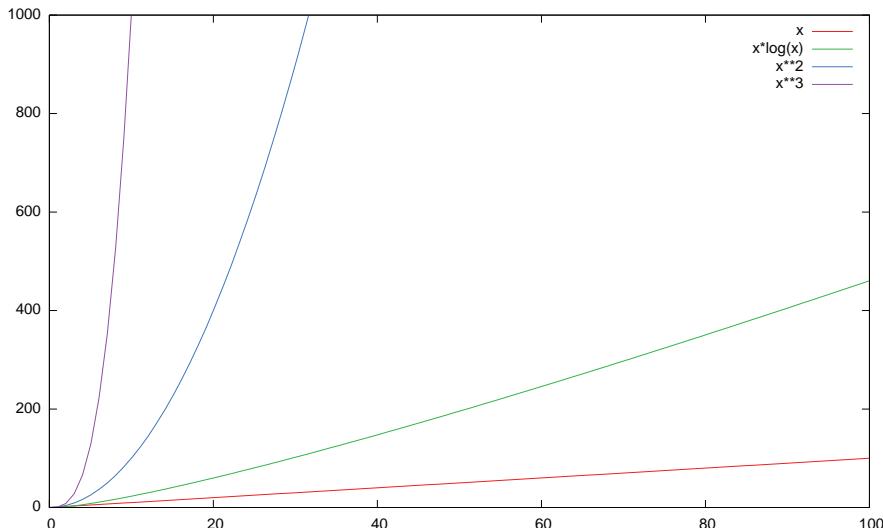
A continuación se presenta un ejemplo sencillo para entender cómo la influencia de los monomios de menor grado disminuye a medida que aumenta la talla del problema:

Supóngase que se está descargando un fichero de Internet, de forma que hay un retraso inicial de 2 seg. para establecer la conexión y que después la descarga se realiza a razón de 1.6 KB/seg. Si el tamaño del fichero es de  $N$  KBytes, el tiempo de descarga viene descrito por la expresión lineal:

$$T(N) = \frac{N}{1,6} + 2$$

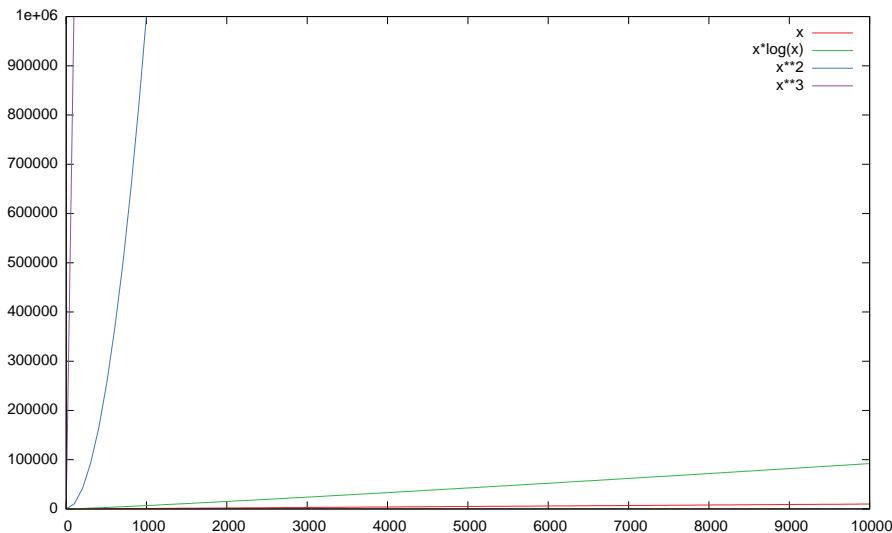
La descarga de un fichero de 80 KB requiere 52 seg., mientras que la descarga de un fichero el doble de grande requiere del orden de 102 seg., es decir casi el doble. Si el fichero es de 8 MB entonces previsiblemente tardará 5002 seg. Esta propiedad, por la cual el tiempo de ejecución es esencialmente proporcional al tamaño de la entrada, caracteriza a un algoritmo lineal. Además, puede observarse como el término independiente o monomio de grado 0 (los 2 segundos iniciales) es prácticamente despreciable cuando la talla del fichero alcanza un tamaño considerable.

En las figuras 12.2 y 12.3 se presentan cuatro funciones típicas en el análisis de algoritmos: lineal,  $n$ -logarítmica ( $n \log n$ ), cuadrática y cúbica para distintos tamaños del problema, respectivamente, desde 1 hasta 100 y desde 1 hasta 10000. Nótese como, a pesar del cambio de escala de un eje con respecto al otro, la cuadrática y la cúbica crecen mucho más rápido que la lineal.



**Figura 12.2:** Comportamiento asintótico de cuatro funciones típicas para valores de  $x$  hasta 100.

En las gráficas citadas se aprecia como las curvas no lineales conducen a tiempos de ejecución mayores. Esto indica que el coste lineal es, de entre todos los que se muestran en estas gráficas, el más favorable. Una función cúbica es una función cuyo término dominante es del tipo  $k'n^3$ . Análogamente, una función cuadrática tiene como término dominante uno del tipo  $k''n^2$  y una función lineal tiene como término dominante uno del tipo  $k'''n$ . En los tres casos  $k', k'', k'''$  representan constantes multiplicativas mayores que 0. Aunque no se representa en estas gráficas, la función logaritmo crece muy lentamente, más lentamente que cualquier raíz que, a su vez, lo hace más lentamente que cualquier función lineal, mientras que la



**Figura 12.3:** Comportamiento asintótico de cuatro funciones típicas para valores de  $x$  hasta 10000.

función  $n \log n$  crece más rápidamente que las lineales, pero más lentamente que las cuadráticas.

En las dos primeras columnas de la tabla 12.1 se muestran, en orden creciente de tasa de crecimiento, distintas funciones que describen comúnmente el tiempo de ejecución de los algoritmos. Son las que se denominan *funciones típicas*. El último paso en el análisis de los algoritmos consistirá en decir a qué función típica se asemeja  $T(n)$  cuando  $n \rightarrow \infty$ , usando para ello la notación asintótica que se describe en la siguiente subsección. En la última columna de la tabla 12.1 aparecen descritas las funciones típicas en términos de dicha notación.

| Función    | Nombre           | Notación Asintótica |
|------------|------------------|---------------------|
| $c$        | constante        | $\Theta(1)$         |
| $\log n$   | logarítmica      | $\Theta(\log n)$    |
| $\sqrt{n}$ | raíz             | $\Theta(\sqrt{n})$  |
| $n$        | lineal           | $\Theta(n)$         |
| $n \log n$ | $n$ -logarítmica | $\Theta(n \log n)$  |
| $n^2$      | cuadrática       | $\Theta(n^2)$       |
| $n^3$      | cúbica           | $\Theta(n^3)$       |
| $2^n$      | exponencial      | $\Theta(2^n)$       |

**Tabla 12.1:** Algunas funciones típicas de coste.

### 12.3.2 Introducción a la notación asintótica

En esta subsección y en la siguiente se explican algunos conceptos matemáticos útiles para comprender el enfoque del resto del capítulo; en concreto las tres variantes de la notación asintótica que se utilizan para expresar el comportamiento en el límite de las funciones de coste. Su utilidad se verá en la siguiente sección, cuando se introduzca el análisis por casos.

Para estudiar cuándo un algoritmo es más eficiente que otro se puede comparar los ratios de crecimiento de sus funciones de coste, esto es, básicamente, estudiar la pendiente de dichas funciones. Así, tendrá mayor crecimiento aquel cuya tasa de crecimiento sea mayor. En la práctica, esto equivale a comparar en el límite las funciones de coste entre sí o, lo que es lo mismo, asintóticamente. En otras palabras, observar cuál es su comportamiento para tamaños del problema suficientemente grandes, ya que las posibles diferencias para valores pequeños no aporta información relevante para poder compararlos.

Antes de continuar, se debe tener presente que habitualmente las funciones de coste temporal son funciones discretas<sup>3</sup> definidas en el dominio de los números naturales. Por tanto, la talla del problema siempre será un número natural.

Las definiciones relativas al uso de la notación asintótica hacen referencia a conjuntos de funciones, positivas y no decrecientes, que cumplen determinadas propiedades.

La primera de ellas es la siguiente:

Dada una función  $g(n)$ , donde  $n$  representa la talla del problema y, por tanto, es un número natural, se define  $\Theta(g(n))$  como el conjunto de funciones:

$$\Theta(g(n)) = \{ f(n) : \exists c_1 > 0, c_2 > 0, \text{ y } n_0 > 0 \mid 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0 \}$$

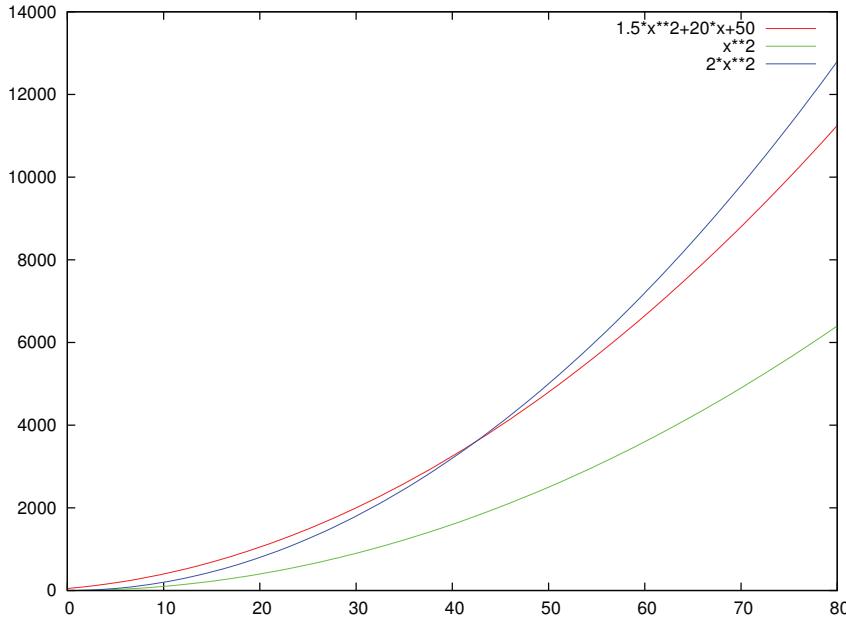
cuyo significado es que  $\Theta(g(n))$  representa el conjunto de funciones  $f(n)$  que pueden ser acotadas, tanto inferior como superiormente, por la misma función típica, en este caso representada por  $g(n)$ . Por ejemplo, decir que  $T(n) \in \Theta(n^2)$  es afirmar que, a partir de cierto valor  $n_0$  de la talla del problema, existen dos constantes positivas  $c_1$  y  $c_2$  de manera que el valor de  $T(n)$  nunca será inferior a  $c_1 \cdot n^2$  ni superior a  $c_2 \cdot n^2$ .

En la figura 12.4 puede observarse como una función cuadrática, el polinomio  $\frac{3}{2}x^2 + 20x + 50$ , está acotada inferiormente por  $x^2$  y superiormente por  $2x^2$  a

---

<sup>3</sup>Sin embargo, a efectos prácticos, suele ser conveniente proceder con ellas como si fueran funciones continuas y, gracias a ello, derivables.

partir de un valor de  $x$  ligeramente superior a 40. En este caso, se puede utilizar  $c_1 = 1$ ,  $c_2 = 2$  y  $n_0 = 50$ , siendo  $f(n) = \frac{3}{2}n^2 + 20n + 50$  y  $g(n) = n^2$ .



**Figura 12.4:** Ejemplo de como la función  $\frac{3}{2}x^2 + 20x + 50$  queda acotada inferiormente por  $x^2$  y superiormente por  $2x^2$ .

De manera análoga a la definición anterior, se definen otros dos conjuntos funcionales mediante los que se representan, respectivamente, las cotas inferiores ( $\Omega(g(n))$ ) y superiores ( $O(g(n))$ ) de cierta función  $g(n)$ :

$$\begin{aligned}\Omega(g(n)) &= \{ f(n) : \exists c > 0 \text{ y } n_0 > 0 \quad | \quad 0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0 \} \\ O(g(n)) &= \{ f(n) : \exists c > 0 \text{ y } n_0 > 0 \quad | \quad 0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0 \}\end{aligned}$$

Entonces, decir que  $t(n) \in \Omega(n)$  es afirmar que, a partir de cierto valor  $n_0$  de la talla del problema, el valor de la función de coste temporal será mayor o igual a  $c \cdot n$ , siendo  $c$  una constante positiva. En otras palabras, es como decir que  $T(n)$  tendrá un comportamiento que como mínimo será lineal, pero que puede ser mayor.  $\Omega(g(n))$  representa el conjunto de funciones que toman valores *mayores* a  $g(n)$  cuando  $n$  tiende a infinito.

Análogamente, decir que  $T(n) \in O(n^3)$  es afirmar que, a partir de cierto valor  $n_0$  de la talla del problema, el valor de la función de coste temporal será menor o igual

a  $c \cdot n^3$ , siendo  $c$  una constante positiva. Por tanto, es como decir que  $T(n)$  tendrá un comportamiento que como máximo será cúbico.  $O(g(n))$  representa el conjunto de funciones que toman valores *menores* que  $g(n)$  cuando  $n$  tiende a infinito.

### 12.3.3 Algunas propiedades de los conjuntos $\Theta$ , $O$ y $\Omega$

Puede observarse que las definiciones de los conjuntos  $\Theta$ ,  $O$  y  $\Omega$  están fuertemente relacionadas con la noción clásica de *límite en el infinito*. Utilizando la definición de este último, es posible enunciar la siguientes tres equivalencias que, además de clarificar las definiciones vistas, suelen mostrarse de gran utilidad:

- $f(n) \in \Theta(g(n)) \iff \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = k, k \in R^+$
- $f(n) \in O(g(n)) \iff (\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = k, k \in R^+) \text{ o } (\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = +\infty)$
- $f(n) \in \Omega(g(n)) \iff (\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = k, k \in R^+) \text{ o } (\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0)$

Tomando de nuevo como ejemplo las funciones que aparecen en la figura 12.4,  $f(n) = \frac{3}{2}n^2 + 20n + 50$  y  $g(n) = n^2$ , entonces, por aplicación inmediata de las equivalencias anteriores, es fácil decir ahora que se cumple tanto que  $f(n) \in O(g(n))$ , como que  $f(n) \in \Omega(g(n))$  y que  $f(n) \in \Theta(g(n))$  puesto que se cumple que  $\lim_{n \rightarrow \infty} \frac{n^2}{\frac{3}{2}n^2 + 20n + 50} = \frac{2}{3}$ .

Obviamente, por el mismo motivo es posible afirmar también que:

$g(n) \in O(f(n))$ , como que  $g(n) \in \Omega(f(n))$  y que  $g(n) \in \Theta(f(n))$  ya que, al igual que antes, se tiene que:  $\lim_{n \rightarrow \infty} \frac{\frac{3}{2}n^2 + 20n + 50}{n^2} = \frac{3}{2}$ .

Otros ejemplos inmediatos son los siguientes:

- $3n^2 + 2n \in O(n^2)$  así como  $3n^2 + 2n \in O(n^3)$  y  $3n^2 + 2n \in O(n^4)$ ,
- $3n^2 + 2n \in \Omega(n^2)$  así como  $3n^2 + 2n \in \Omega(n)$  y  $3n^2 + 2n \in \Omega(\log(n))$ ,
- pero  $3n^2 + 2n \notin \Omega(n^3)$  y  $3n^2 + 2n \notin O(\log(n))$ .

Se anima al lector a calcular los límites de los cocientes de las funciones implicadas para comprobar los resultados mostrados.

Utilizando las definiciones y equivalencias presentadas hasta el momento, es posible demostrar de forma sencilla numerosas propiedades relacionadas con la notación

asintótica. Muchas de ellas son relevantes por su posible aplicación en la determinación de las cotas de complejidad de los algoritmos. De entre las mismas se destacan las tres siguientes:

- $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$
- $f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ y } f(n) \in \Omega(g(n))$
- $f(n) \in O(g(n)) \iff f(n) + g(n) \in O(g(n))$

Mediante la primera propiedad se señala que cualquier función,  $f(n)$ , está acotada superiormente por otra,  $g(n)$ , si y solo si la segunda tiene a la primera como cota inferior.

La segunda propiedad señala que una función,  $f(n)$ , tiene el mismo crecimiento que otra,  $g(n)$ , si y solo si la segunda es cota superior e inferior, a la vez, de la primera.

La tercera propiedad es de aplicación, por ejemplo, en el caso en que hayan varias secuencias de código acotadas, cada una de ellas, por cierta función de coste que formen, a su vez, parte de una secuencia de código mayor. Entonces el coste temporal de la secuencia completa de código estará acotado por el de la función de mayor orden. Así, por ejemplo, el coste de un programa formado por una secuencia de ejecución de varios subprogramas es el del subprograma de mayor coste.

También es de aplicación en la determinación del coste asintótico de una función expresada como una suma de términos tal y como puede ser, por ejemplo, cualquier polinomio; entonces, el coste de toda la función viene dado por el del término de mayor orden (el monomio de mayor grado, cuando se trata de un polinomio).

#### 12.3.4 La jerarquía de complejidades

De las definiciones y relaciones anteriores se pueden deducir las siguientes dos relaciones de inclusión, equivalentes entre sí:

$$\begin{aligned}O(1) &\subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^3) \dots \subset O(2^n) \subset O(n!) \\ \Omega(n!) &\subset \Omega(2^n) \subset \dots \subset \Omega(n^3) \subset \Omega(n^2) \subset \Omega(n\log n) \subset \Omega(n) \subset \Omega(\log n) \subset \Omega(1)\end{aligned}$$

Nótese que el hecho de que un conjunto esté estrictamente incluido en otro, implica que las funciones del primero están incluidas en el segundo, no siendo cierto lo contrario. Así, por ejemplo, la inclusión  $O(n) \subset O(n\log n)$  (o su equivalente  $\Omega(n\log n) \subset \Omega(n)$ ) puede entenderse como que las funciones con un crecimiento con cota superior lineal,  $O(n)$ , están a su vez acotadas superiormente por las n-logarítmicas,  $O(n\log n)$ , aunque, naturalmente, no es cierto el caso contrario, esto

es, no todas las funciones con cota superior n-logarítmica están acotadas superiormente de forma lineal.

Por ello, las relaciones de inclusión que aparecen en cualquiera de las dos expresiones anteriores muestran una jerarquía de posibles cotas (superiores o inferiores según la expresión) que permiten clasificar las funciones por su tasa de crecimiento. Así, las funciones constantes tienen una cota superior en las logarítmicas que, a su vez, la tienen en las lineales, n-logarítmicas, cuadráticas, y así sucesivamente.

Esta sucesión de relaciones estrictas de inclusión recibe el nombre de *jerarquía de complejidades*. La misma permite establecer, a su vez, una jerarquía de los algoritmos como función de su eficiencia asintótica.

Hay que notar que existen otros muchos tipos de función que, por simplicidad, no se han reflejado en las jerarquías anteriores. Por ejemplo, para cada pareja de inclusión entre dos conjuntos de funciones polinómicas, existen conjuntos intermedios de funciones polinómico-logarítmicas como los de la secuencia:  $O(n^2) \subset O(n^2\log n) \subset O(n^2\log^2 n) \subset \dots \subset O(n^2\log^k n) \subset \dots \subset O(n^3)$ .

### 12.3.5 Uso de la notación asintótica

Cuando se analiza el coste de un algoritmo, el objetivo es conocer a qué función típica se aproxima el comportamiento asintótico de dicho algoritmo. También es necesario poder comparar estas curvas de costes con el fin de poder decidir si un determinado algoritmo es mejor, peor o equivalente a otro. Es por esto que lo que interesa es calcular la tasa de crecimiento de las funciones de coste, expresándolo en notación asintótica. Como ya se ha dicho y se resume ahora, tres razones apoyan esta decisión:

1. Para valores de  $n$  suficientemente grandes el valor de la función está completamente determinado por el término dominante: por el monomio de mayor grado en un polinomio, por la exponencial de mayor base en el caso de una función que combine varias exponenciales, etc.

Por ejemplo, si se tiene  $T(n) = n^3 + n^2 + n$ , para  $n > 100$  las diferencias con respecto a  $n^3$  ya comienzan a no ser significativas. En concreto  $n^2 + n$  representa apenas el 1%. Como ejercicio, el lector puede estudiar como decrece la función  $T(n)/n^3$ .

2. El valor exacto del coeficiente del término dominante no se conserva al cambiar de entorno de programación.

En efecto, si tras el análisis teórico se obtiene como función de coste temporal de un algoritmo  $T(n) = 4n^2 + 2n + 10$ , los valores de las constantes 4, 2 y 10 serán distintos según lo que se haya considerado como paso de programa. Si se intenta determinar su valor concreto tras un análisis experimental, se verá

como varían de un ordenador a otro, incluso en un mismo ordenador con el mismo sistema operativo varían si se utilizan lenguajes de programación diferentes. Por tanto, si el resultado de  $T(n)$  son pasos de programa, según lo que se haya considerado paso de programa, cambia el valor concreto de estas constantes y, por supuesto, su equivalencia en segundos cambia en función del entorno de programación.

3. El uso de la notación asintótica permite, como ya se ha dicho, establecer un orden relativo entre las funciones de coste, siguiendo para ello el establecido en la jerarquía de complejidades:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset \dots \subset O(2^n) \subset (n!)$$

De esta manera, en cuanto se obtiene la función de coste temporal tras el análisis teórico, y según lo comentado en los puntos anteriores, ya se puede afirmar qué algoritmo es (asintóticamente) más eficiente, o si son (asintóticamente) equivalentes, observando el término dominante e ignorando su constante multiplicativa.

## 12.4 Análisis por casos

### 12.4.1 Caso mejor, caso peor y coste promedio

Como ya se ha comentado, el coste de un algoritmo es una función no decreciente de la talla del problema. En esta sección se estudia el hecho de que, para un mismo valor de la talla, el coste del algoritmo puede depender de la configuración de la entrada del problema, es decir, puede depender de la *instancia del problema*. Si se piensa en la ordenación de un conjunto de valores enteros, se tiene que  $\{4, -1, 7, 3, -2, 6, 0\}$  y  $\{1, 2, 3, 4, 5, 6, 7\}$  son dos instancias diferentes del mismo problema, es decir, son dos entradas de la misma talla pero de diferente configuración.

Una *instancia de un problema* representa un subconjunto de todas las posibles configuraciones equivalentes de la entrada para una misma talla. Siguiendo con la ordenación, todas las entradas de siete enteros ya ordenados pertenecen a la misma instancia del problema. La función de coste temporal del algoritmo, es decir, su comportamiento, no varía para todas las muestras (o configuraciones) pertenecientes a una misma instancia.

Algunos algoritmos se ven afectados por la manera en que se le presentan los datos de entrada, por ejemplo, algunos algoritmos de ordenación acaban antes su ejecución si los datos que deben ordenar ya lo están que si no lo están. Es decir, distintas instancias del problema afectan de manera distinta a un mismo algoritmo. En este caso se dice que existen *instancias significativas*, pues el coste

del algoritmo no es el mismo para todas las configuraciones posibles de los datos de entrada.

Si en el análisis de los costes de un algoritmo se detectan instancias significativas, entonces se tipifican los siguientes casos:

- *Coste del algoritmo en el caso peor*: es la complejidad del mismo para la instancia del problema que presente el peor coste. Se denota por  $T^p(n)$ .
- *Coste del algoritmo en el caso mejor*: es la complejidad del mismo para la instancia del problema que presente el coste menor. Se denota por  $T^m(n)$ .
- *Coste promedio del algoritmo*: es la media de los costes de todas las instancias del problema. Se denota por  $T^\mu(n)$ .

Los estudios más útiles son el coste del algoritmo en el peor caso y el coste promedio. En general, el estudio del coste promedio de un algoritmo es difícil de realizar, tanto analíticamente como experimentalmente, la principal dificultad reside en conocer la distribución de probabilidad sobre las instancias del problema.

#### 12.4.2 Ejemplos: algoritmos de recorrido y búsqueda

El problema del recorrido de un array de  $n$  elementos presenta una única instancia ya que, independientemente de cuáles sean los datos contenidos en el array, hay que tratarlos todos; por tanto, no se puede distinguir entre casos mejor y peor. Sin embargo, el problema de la búsqueda secuencial presenta varias instancias:

- que el elemento se encuentre en la posición 0 del array,
- que se encuentre en la posición 1,
- ...
- que se encuentre en la última posición ( $n - 1$ ),
- que no se encuentre.

En total, se tienen  $n + 1$  posibilidades, las  $n$  primeras corresponden a casos de búsqueda con éxito, y la última al caso de la búsqueda sin éxito. Si el algoritmo implementado realiza una búsqueda secuencial ascendente, el caso mejor, el más favorable, corresponderá a la primera instancia (el elemento a buscar se encuentra en la primera posición). El caso peor vendrá dado para la última instancia (búsqueda sin éxito).

Por ello, el algoritmo de búsqueda, para un mismo tamaño del array en el que encontrar un valor, se comportará de manera diferente según el valor exista dentro

del array o no, o según la posición en que se encuentre en el caso de que exista. Por lo tanto, la distribución de los valores dentro del array afecta al comportamiento del algoritmo, aunque el tamaño del array sea siempre el mismo.

En general, el análisis por casos deberá efectuarse siempre que se observe que un algoritmo presente diferentes comportamientos según se distribuyan, para un mismo tamaño del problema, los datos de entrada. La determinación de las instancias significativas y la particularización a los casos peor y mejor permitirán establecer las cotas temporales de ejecución del algoritmo. He ahí la importancia de efectuar este análisis.

## 12.5 Análisis del coste de los algoritmos

A continuación se señalan los pasos a seguir para establecer el coste de un algoritmo de forma independiente a que el mismo se haya expresado iterativa o recursivamente. Más adelante, en las secciones siguientes, se detallarán algunos de estos pasos, dependiendo del tipo, iterativo o recursivo, del algoritmo en estudio. Conviene ser conscientes, sin embargo, de que es posible utilizar las técnicas de cálculo dirigidas a una tipología determinada (por ejemplo, iterativa o recursiva), para obtener el coste de un algoritmo expresado con la otra (recursiva o iterativa). Los pasos a seguir son los siguientes:

1. Determinar la talla del problema; esto es, estudiar de qué parámetro o parámetros va a depender la función de coste.
2. Elegir la unidad de medida en que se vaya a expresar el coste del algoritmo. Esta unidad de medida será diferente según el cálculo se corresponda con el del coste espacial, por ejemplo, *bytes* de memoria empleados, o temporal, como por ejemplo, los ya vistos *pasos de programa* o algún otro tipo de unidad, tales como las *instrucciones críticas*, que serán introducidas un poco más adelante.
3. Analizar si para una misma talla del problema existen instancias significativas para el coste.  
Para esto se puede intentar responder la siguiente pregunta: ¿para una misma talla del problema, se tendrá un coste mayor o menor según la configuración de los datos de entrada?
4. Obtener la función de coste (temporal o espacial). Este paso consiste en obtener una expresión que representa el esfuerzo de cómputo o el del uso de la memoria que empleará el algoritmo, en función de la talla. Si existen instancias significativas, el estudio de costes se particularizará para el caso peor y el caso mejor. El estudio de costes en el caso peor proporcionará la cota superior del coste del algoritmo y en el caso mejor la cota inferior.

5. Expresar mediante notación asintótica el comportamiento del algoritmo.

Si el algoritmo presenta instancias significativas, se utilizará la notación  $O(f(n))$  para expresar la cota superior y  $\Omega(f(n))$  para la cota inferior. En caso de no distinguirse diferentes casos, se utilizará la notación  $\Theta(f(n))$ , esto es, la misma función típica es cota superior e inferior.

## 12.6 Análisis del coste de los algoritmos iterativos

### 12.6.1 Otra unidad de medida temporal: la instrucción crítica

Si el objetivo final del estudio temporal de un algoritmo es obtener su coste asintótico, entonces una forma simplificada de hacerlo consiste en contar el número de veces que se repite una operación o instrucción, elegida por nosotros, que debe tener la propiedad de que se repita por lo menos tanto como cualquier otra. Este tipo de instrucción que, a fin de cuentas, está entre las más repetidas (asintóticamente hablando) se denomina *instrucción crítica* o *instrucción barómetro*.

Una instrucción crítica es, por lo tanto, una operación elemental cuyo tiempo de ejecución es constante, no dependiendo de la talla del problema, para la que, además, el orden de su número total de repeticiones es igual o mayor que el de cualquier otra. Es decir, *se ejecuta por lo menos con tanta frecuencia como cualquier otra*. En realidad, no hay problema si hay alguna instrucción que aparece más veces en el código del algoritmo siempre que, asintóticamente hablando, no presente un orden temporal superior que la instrucción elegida.

Es importante resaltar que cuando se utiliza como unidad de medida temporal bien la instrucción crítica, bien el paso de programa, se puede decidir que cualquiera de ellas esté compuesta por una secuencia de más de una operación básica siempre que el coste temporal de dicha secuencia sea independiente de la talla.

### 12.6.2 Eficiencia de los algoritmos de recorrido

Dado un array **a** definido de tamaño  $n$ , el siguiente algoritmo realiza un recorrido de dicho array para ejecutar una operación sobre cada elemento. La operación se representa por **tratar()** y  $n$  es la talla del problema.

```
for (int i=0; i<n; i++) tratar(a[i]);
```

Si se asume que **tratar(a[i])** es una operación de coste constante que se realiza sobre el elemento  $i$ -ésimo del array, es decir, su tiempo de ejecución es siempre el mismo y no depende de  $n$  ni del valor del elemento a tratar, entonces el coste del algoritmo será una función que dependerá del número de elementos del array,  $T(n)$ .

El algoritmo no presenta instancias significativas porque siempre se deben tratar todos los elementos del array. De esta manera, la función de coste  $T(n)$  se puede aproximar contando el número de veces que se repite la instrucción `tratar(a[i])`, que puede ser una instrucción crítica para este segmento de código. También se puede tomar como instrucción crítica el incremento de la variable de control del bucle, la expresión `i++`, o la evaluación de la guarda `i < n`. Si se considera como instrucción crítica `tratar(a[i])` o `i++`, se tiene el problema de que si el array está vacío el conteo da como resultado cero. Cabe decir que en Java se pueden definir arrays de tamaño cero, lo que puede provocar alguna incoherencia en un análisis preliminar del algoritmo. Así que, aunque las tres operaciones comentadas son válidas como instrucciones críticas, resulta más cómodo considerar la guarda del bucle `i < n` como instrucción crítica, gracias a que se ejecuta, al menos, una vez más que el resto. Así el coste del algoritmo será:  $T(n) = n + 1$  instrucciones críticas  $\in \Theta(n)$ .

Naturalmente, si en lugar de contar instrucciones críticas en el cálculo del coste del bucle anterior se hubiesen contabilizado pasos de programa, se podría tener como expresión válida del coste, suponiendo que se hacen  $n + 1$  evaluaciones de la guarda y que cada una de ellas se corresponde con un paso de programa, que se suma al paso contabilizado como inicialización del bucle:  $T(n) = 1 + n + 1$  pasos  $\in \Theta(n)$ . Obviamente, no hay diferencia entre ambos cálculos a efectos asintóticos.

### 12.6.3 Eficiencia de los algoritmos de búsqueda secuencial

Dado un array `a` definido de  $n$  elementos, el siguiente algoritmo realiza una búsqueda secuencial ascendente para determinar si algún elemento cumple una propiedad dada. Devuelve la posición del primer elemento que cumple dicha propiedad, o devuelve el valor  $-1$  en caso de que ningún elemento del array la cumpla. Al igual que en el ejemplo de recorrido visto antes,  $n$  representa la talla del problema.

```
int i = 0;
while (i < n && !propiedad(a[i])) i++;
if (i < n) return i;
else return -1;
```

Supóngase que `propiedad(a[i])`, la comprobación de la condición de búsqueda, es una operación de coste constante. Entonces el coste del algoritmo será una función dependiente del número de elementos del array en el que se efectúa la búsqueda,  $T(n)$ . Sin embargo, en este caso, el coste del algoritmo sí que va a depender de la instancia del problema; para una misma talla (arrays del mismo tamaño) el algoritmo puede acabar en cualquier posición antes de llegar a la última. En particular este algoritmo presenta  $n + 1$  instancias significativas.

**Análisis del caso peor.** El caso peor se da cuando el bucle se ejecuta el mayor número posible de veces:  $n$ ; esto ocurre cuando no hay ningún elemento en el

mismo que satisfaga la propiedad enunciada. La función de coste  $T^p(n)$  se puede aproximar contando el número de veces que se repite la guarda del bucle `i<n && !propiedad(a[i])`, considerando ésta como la instrucción crítica. También se podría haber tomado como instrucción crítica la operación de autoincremento de la variable de control del bucle `i++`. Pero por lo comentado en el ejemplo anterior resulta más cómodo la guarda del bucle. Además, en este caso, se puede ver que se considera como un paso de programa una secuencia de tres operaciones elementales: la comparación `i<n`, la evaluación de la propiedad `propiedad()` que también es una operación elemental en este ejemplo, y el *and* lógico que combina el resultado de las dos anteriores para obtener un único resultado de tipo `boolean`. Así se obtiene como resultado:

$$T^p(n) = n + 1 \in \Theta(n) \quad \text{Lineal}$$

**Análisis del caso mejor.** El caso mejor se da cuando el bucle se ejecuta el menor número posible de veces: 1; esto ocurre cuando el primer elemento satisface la propiedad enunciada. En este caso ni tan siquiera se llega a ejecutar una sola vez el autoincremento. Por tanto, considerando la guarda del bucle como instrucción crítica, al igual que en el caso peor, se tiene que la función de coste  $T^m(n)$  se puede aproximar contando el número de veces que se repite la instrucción crítica, que es una vez:

$$T^m(n) = 1 \in \Theta(1) \quad \text{Constante}$$

**Cotas para el coste del algoritmo.** Se usa como cota superior la función de coste del algoritmo en el caso peor y como cota inferior la función de coste del algoritmo en el caso mejor. Para expresar estas cotas también se utiliza notación asintótica,  $O$  para expresar la cota superior y  $\Omega$  para expresar la cota inferior. Así, a partir de las  $T^p(n)$  y  $T^m(n)$  obtenidas, se puede decir que la función de coste temporal del algoritmo en general,  $T(n)$ , pertenece a dos conjuntos:

$$T(n) \in O(n) \quad T(n) \in \Omega(1)$$

uno que representa la cota inferior  $\Omega(1)$  y otro la superior  $O(n)$ , o bien:

$$T(n) \in \Omega(1) \cap O(n)$$

Esto significa que  $T(n)$ , la función de coste temporal del algoritmo de búsqueda secuencial, nunca tendrá un comportamiento peor (coste mayor) que el que puede representarse por una función lineal, y que tampoco tendrá nunca un comportamiento mejor (coste menor) que el que puede representarse por una constante.

### 12.6.4 Estudio del coste promedio del algoritmo de búsqueda secuencial

Para estudiar el coste promedio de un algoritmo es necesario conocer la distribución de probabilidad sobre las diferentes instancias. Se analizarán dos supuestos:

1. La búsqueda siempre tiene éxito y la probabilidad de que el elemento buscado se encuentre en cualquiera de las posiciones del array es la misma.
2. Es equiprobable que el elemento buscado esté o no en el array; en el caso de encontrarse, todas las posiciones son equiprobables.

**Primer supuesto.** Hay  $n$  instancias posibles, cada una con probabilidad  $\frac{1}{n}$ . El coste de cada instancia será el número de veces que se evalúa la guarda del bucle para alcanzar la posición  $i$ , que será  $i + 1$ . Luego

$$T^\mu(n) = \sum_{i=0}^{n-1} \frac{1}{n}(i+1) = \frac{1}{n} \sum_{i=1}^n i = \frac{n(n+1)}{2n} = \frac{(n+1)}{2} \in \Theta(n)$$

**Segundo supuesto.** Hay  $n + 1$  instancias posibles, que el elemento no esté (con probabilidad  $\frac{1}{2}$ ), o que esté en una cierta posición  $i$ , con  $i = 0..n - 1$  (cada una de estas posibilidades con probabilidad  $\frac{1}{2n}$ ). El coste de la primera situación sería  $n + 1$  y el del resto de situaciones sería  $i + 1$ . Luego

$$T^\mu(n) = \frac{n+1}{2} + \frac{1}{2n} \sum_{i=1}^n i = \frac{n+1}{2} + \frac{n+1}{4} = \frac{3}{4}(n+1) \in \Theta(n)$$

## 12.7 Análisis del coste de los algoritmos recursivos

Si la función de coste de un algoritmo iterativo se formula como el número de veces que se realiza la instrucción crítica, en un algoritmo recursivo la forma natural de establecer la función de coste es también recurrente.

En los algoritmos iterativos de la sección anterior, al menos, aparece un bucle y, por tanto, su análisis se centra en obtener una función de coste temporal que refleje el número de veces que se repite el bucle. En esta sección se estudian algoritmos recursivos que incluyen, al menos, una invocación a sí mismo. Analizar un algoritmo recursivo se centrará en obtener una función de coste temporal que refleje el número de veces que el método se invoca así mismo antes del caso trivial, caso en el que ya no vuelve a invocarse a sí mismo. La manera más natural de obtener la función de coste temporal de un método que se invoca a sí mismo será igualmente con una función recurrente.

### 12.7.1 Planteamiento de la función de coste. Ecuaciones de recurrencia

Un algoritmo recursivo es aquel que se define en función de él mismo. En la complejidad temporal de un algoritmo recursivo influyen:

- El número de llamadas recursivas que genera cada llamada al método. Si la recursión es lineal, cada llamada provoca otra a su vez. Si no, cada llamada provoca 2 o más.
- La forma en la que se reduce el tamaño del problema ( $n$ ) en cada llamada. Normalmente, se reduce en una constante  $c$  tal que la reducción es de la forma  $(n - c)$  con  $c \geq 1$  o  $\frac{n}{c}$  con  $c > 1$ .
- El coste del resto de operaciones que realiza el algoritmo excluida(s) la(s) llamada(s) recursiva(s).

La función de coste de un algoritmo recursivo lineal será:

$$T(n) = \begin{cases} T(n') + f(n) & \text{si } n > n_0 \\ T(n_0) & \end{cases}$$

donde  $n_0$  es el valor de la talla del problema para el caso base y  $T(n_0)$  el coste del algoritmo para dicho caso,  $f(n)$  es el coste del resto de operaciones que realiza el método excluida la llamada y  $n'$  es el nuevo tamaño del problema (inferior a  $n$ ).

La función de coste de un algoritmo recursivo doble será:

$$T(n) = \begin{cases} 2 \cdot T(n') + f(n) & \text{si } n > n_0 \\ T(n_0) & \end{cases}$$

si se supone que en las dos llamadas el tamaño del problema se ha reducido de igual forma.

Estas fórmulas, que son la expresión del coste en forma recursiva, se denominan *ecuaciones de recurrencia*. El planteamiento de estas ecuaciones cuando el número de llamadas es mayor que dos es directo.

La resolución de estas ecuaciones de recurrencia da la función de coste del algoritmo bajo estudio.

**Ejemplo 12.1.** El siguiente método calcula el factorial de un número natural:

```
/** n>=0 */
static int factorial(int n) {
 if (n==0) return 1;
 else return n * factorial(n-1);
}
```

La talla del problema es precisamente  $n$  y no se observan instancias significativas para el coste por lo que se planteará una única función de coste. El método `factorial(int)` presenta recursión lineal y su coste se puede plantear a través de la ecuación de recurrencia siguiente:

$$T(n) = \begin{cases} T(n - 1) + 1 & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

considerando, para simplificar, que en el caso trivial se realiza un paso de programa (devolver el valor 1) y que en el caso general también se realiza un paso de programa tras la llamada al método recursivo (la multiplicación por  $n$  y la devolución del resultado).

No debe llevar a confusión que se utilice  $n$  como argumento del método `factorial()` y que, a su vez, represente la talla del problema que, en este caso, es justamente el valor del cual se quiere obtener el factorial.

**Ejemplo 12.2.** El siguiente método recursivo devuelve el término  $n$ -ésimo de la sucesión de Fibonacci:

```
/** n>=0 */
public static int fibonacci(int n) {
 if (n<=1) return n;
 else return fibonacci(n-1) + fibonacci(n-2);
}
```

La talla del problema es precisamente  $n$  y no se observan instancias significativas para el coste por lo que, también en este caso se planteará una única función de coste. El método `fibonacci(int)` presenta recursión múltiple con dos llamadas y su coste se puede plantear a través de la ecuación de recurrencia siguiente:

$$T(n) = \begin{cases} T(n - 1) + T(n - 2) + 1 & \text{si } n > 1 \\ 1 & \text{si } 0 <= n <= 1 \end{cases}$$

considerando, para simplificar, que en el caso trivial se realiza un paso de programa (devolver el valor  $n$ ) y que en el caso general también se realiza un paso

de programa tras la llamada al método recursivo (la suma y la devolución del resultado).

**Ejemplo 12.3.** El método `ultimaP(String)` (ejemplo 11.3) de la clase `SecuenciaDeCírculos` devuelve la posición de la última aparición de un círculo de color `col` de la secuencia. Consta de una única instrucción que es una llamada al método recursivo homónimo `ultimaP(String,int)` que resuelve el problema planteado de forma recursiva lineal.

```

public int ultimaP(String col) {
 return ultimaP(col,talla-1);
}

/** -1<=fin<elArray.length */
private int ultimaP(String col, int fin) {
 int resMetodo = -1;
 if (fin>-1)
 if (elArray[fin].getColor().equals(col))
 resMetodo = fin;
 else resMetodo = ultimaP(col,fin-1);
 return resMetodo;
}

```

Se trata de un problema de búsqueda en un array desde la posición 0 hasta `fin`; por eso la talla del problema será precisamente el tamaño de la parte del array en el que se efectúa la búsqueda,  $n = \text{fin}$ . Del análisis del algoritmo empleado se desprende la existencia de instancias significativas para el coste.

- El caso mejor se da cuando en la posición de la primera llamada (`fin=talla-1`) se encuentra un círculo del color buscado (`elArray[fin].getColor().equals(col)`). Corresponde a un caso base de la recursión en el que la llamada se resuelve devolviendo el valor del parámetro `fin`. La función de coste es constante:

$$T^m(n) = 1 \in \Theta(1)$$

- El caso peor se da cuando no hay ningún círculo del color buscado (búsqueda sin éxito). Las ecuaciones de recurrencia que expresan la función de coste son:

$$T^p(n) = \begin{cases} T^p(n-1) + 1 & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

### 12.7.2 Resolución de las ecuaciones de recurrencia. Teoremas

Aunque no es objeto de este libro el estudio de métodos de resolución de ecuaciones de recurrencia, se plantea su resolución mediante la aplicación del *método directo o de sustitución* en los casos de ecuaciones de recurrencia de primer orden y el *método de la ecuación característica* en el caso de ecuaciones de recurrencia de segundo orden. También se pueden utilizar cambios de variables y acotaciones para facilitar la resolución de la ecuación de recurrencia.

**Ejemplo 12.4.** La ecuación de recurrencia que expresa el coste del método `factorial(int)` es la siguiente, donde  $n$  es el valor del parámetro:

$$T(n) = \begin{cases} T(n - 1) + 1 & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Se trata de una recurrencia de primer orden de coeficiente constante ( $a_n = a_{n-1} + 1$ ) que se puede resolver por el *método directo o de sustitución*, que consiste en ir desarrollando la función, tomando valores decrecientes de  $n$ , hasta llegar al caso en el que cesa la recurrencia (caso base o trivial). Así:

$$\begin{aligned} T(n) &= T(n - 1) + 1 \\ &= T(n - 2) + 2 \\ &= T(n - 3) + 3 \\ &\dots \\ &= T(n - i) + i \\ &\dots \\ &= T(n - (n - 1)) + n - 1 \\ &= T(n - n) + n \\ &= 1 + n \in \Theta(n) \end{aligned}$$

**Ejemplo 12.5.** La ecuación de recurrencia que expresa el coste del método `fibonacci(int)` es la siguiente, donde  $n$  es el valor del parámetro:

$$T(n) = \begin{cases} T(n - 1) + T(n - 2) + 1 & \text{si } n > 1 \\ 1 & \text{si } 0 \leq n \leq 1 \end{cases}$$

Se trata de una recurrencia de segundo orden con coeficientes constantes ( $t_n = t_{n-1} + t_{n-2}$ ) que se puede resolver por el *método de la ecuación característica*, que consiste en plantear la ecuación característica ( $x^2 - x - 1 = 0$ ) a partir de la recurrencia y resolverla.

Las soluciones de esta ecuación son  $r_1 = \frac{1+\sqrt{5}}{2}$  y  $r_2 = \frac{1-\sqrt{5}}{2}$  y la solución de la ecuación de recurrencia tiene la forma:

$$T(n) = c_1 \cdot \frac{1 + \sqrt{5}^n}{2} + c_2 \cdot \frac{1 - \sqrt{5}^n}{2}$$

donde las constantes  $c_1$  y  $c_2$  se podrían calcular a partir de condiciones iniciales. No es necesario para constatar que el comportamiento asintótico de la función de coste del método `fibonacci(int)` es exponencial con el valor del parámetro.

Para obtener el orden de las relaciones de recurrencia más habituales se pueden aplicar los siguientes teoremas.

Sean  $a \geq 1, b, c, d, n \in \mathbb{N}$ ,

**Teorema 1:** La solución a la ecuación  $f(n) = a \cdot f(n - c) + b$  es:

- Si  $a = 1$ ,  $f(n) \in \Theta(n)$
- Si  $a > 1$ ,  $f(n) \in \Theta(a^{\frac{n}{c}})$

**Teorema 2:** La solución a la ecuación  $f(n) = a \cdot f(n - c) + b \cdot n + d$  es:

- Si  $a = 1$ ,  $f(n) \in \Theta(n^2)$
- Si  $a > 1$ ,  $f(n) \in \Theta(a^{\frac{n}{c}})$

**Teorema 3:** La solución a la ecuación  $f(n) = a \cdot f(\frac{n}{c}) + b$  es:

- Si  $a = 1$ ,  $f(n) \in \Theta(\log_c n)$
- Si  $a > 1$ ,  $f(n) \in \Theta(a^{\log_c n})$

**Teorema 4:** La solución a la ecuación  $f(n) = a \cdot f(\frac{n}{c}) + b \cdot n + d$  es:

- Si  $a < c$ ,  $f(n) \in \Theta(n)$
- Si  $a = c$ ,  $f(n) \in \Theta(n \cdot \log_c n)$
- Si  $a > c$ ,  $f(n) \in \Theta(n \cdot a^{\log_c n})$

**Ejemplo 12.6.** Las ecuaciones de recurrencia que expresan la función de coste para el caso peor de `ultimpaP(String)` son:

$$T^p(n) = \begin{cases} T^p(n - 1) + 1 & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Se pueden resolver aplicando el Teorema 1 con  $a = c = 1$ , así  $T^p(n) \in \Theta(n)$ . El coste de `ultimpaP(String)` es por tanto:  $T(n) \in \Omega(1) \cap T(n) \in O(n)$ .

**Ejemplo 12.7.** El problema de las *Torres de Hanoi* se plantea de la manera siguiente: se dispone de tres torres numeradas de 1 a 3 y de  $n$  discos de diferentes tamaños. Inicialmente, todos los discos se encuentran en una de las torres (torre 1) apilados en forma decreciente según su diámetro. Se deben desplazar a la torre número 3, utilizando como auxiliar la torre número 2. Se imponen las siguientes restricciones:

- Sólo se puede desplazar un disco en cada movimiento.
- No puede haber un disco de mayor diámetro situado sobre uno más pequeño.

Una solución recursiva a este problema la da el siguiente algoritmo:

```
/** n>=1 */
static void hanoi(int n, Torre orig, Torre dest, Torre aux) {
 if (n==1) mueveDisco(orig,dest);
 else {
 hanoi(n-1,orig,aux,dest);
 mueveDisco(orig,dest);
 hanoi(n-1,aux,dest,orig);
 }
}
```

Si se supone que la operación `mueveDisco()` tiene coste constante (considerando un paso de programa, su función de coste temporal  $\in \Theta(1)$ ), entonces la siguiente ecuación de recurrencia expresa el coste de este algoritmo:

$$T(n) = \begin{cases} 2 \cdot T(n - 1) + 1 & \text{si } n > 1 \\ 1 & \text{si } n = 1 \end{cases}$$

Aplicando el *método de sustitución* se obtiene el coste exponencial característico de este problema:

$$\begin{aligned}
 T(n) &= 2^1 \cdot T(n-1) + 1 \\
 &= 2^2 \cdot T(n-2) + 3 \\
 &= 2^3 \cdot T(n-3) + 7 \\
 &= 2^4 \cdot T(n-4) + 15 \\
 &\dots \\
 &= 2^i \cdot T(n-i) + 2^i - 1 \\
 &\dots \\
 &= 2^{n-1} + (2^{n-1} - 1) \\
 &= 2^n - 1 \in \Theta(2^n)
 \end{aligned}$$

### 12.7.3 Coste espacial de la recursión

Se puede definir el *coste espacial* de un método como el número de registros de activación que, como mucho, llegan a existir simultáneamente en memoria a lo largo de la ejecución de la llamada, multiplicado por el tamaño del registro de activación.

Como se ha visto en la sección 11.3 del capítulo 11, a diferencia de los métodos iterativos, los métodos recursivos hacen un uso intensivo de la pila de registros de activación. En un método recursivo, el número máximo de registros de activación que coexisten simultáneamente en la pila coincide con el número máximo de llamadas realizadas al método hasta llegar a la del caso base. Un método iterativo equivalente hace uso únicamente de un registro de activación, el de la propia llamada al método. Así, el coste espacial de un método recursivo muchas veces será mayor que el del método iterativo equivalente.

Por ejemplo, para el método recursivo **factorial** visto anteriormente, la llamada al método con  $n = k$ , **factorial(k)**, provoca  $k$  llamadas sucesivas, con  $n = k-1$ ,  $n = k-2$ , ...  $n = 0$ , coexistiendo en memoria  $k+1$  registros de activación. Cada registro de activación contendrá, al menos, la variable de tipo **int** asociada al parámetro del método. Se puede decir, por tanto, que la complejidad espacial de este algoritmo es lineal con el tamaño del problema; en notación asintótica se escribirá  $\Theta(k)$ . Mientras que la complejidad espacial de un algoritmo iterativo equivalente será constante; en notación asintótica se escribirá  $\Theta(1)$ .

## 12.8 Complejidad de algunos algoritmos numéricos recursivos

### 12.8.1 La multiplicación de números naturales

Una solución recursiva al problema de la multiplicación de dos números naturales podría ser la siguiente:

```
/** a>=0 y b>=0 */
static int multNat1(int a, int b) {
 if (a==0) return 0;
 else return multNat1(a-1,b) + b;
}
```

El número de llamadas recursivas que se producen en total es  $a$ . Una primera mejora de este algoritmo consiste en hacer decrecer el menor de  $a$  y  $b$ , lo que se puede lograr simplemente invocando a `multNat1(int,int)` como sigue:

```
/** a>=0 y b>=0 */
static int multNat2(int a, int b) {
 if (a<b) return multNat1(a,b);
 else return multNat1(b,a);
}
```

Volviendo a `multNat1(int,int)`, su complejidad va a depender del valor del primer argumento:  $a$ . La recursión es lineal, pues el tamaño del problema decrece en una unidad a cada nueva llamada, y el coste del resto de operaciones es constante. Por esto, las ecuaciones de recurrencia que expresan el coste de este algoritmo son las siguientes:

$$T(a) = \begin{cases} T(a - 1) + 1 & \text{si } a > 0 \\ 1 & \text{si } a = 0 \end{cases}$$

Si estas ecuaciones se resuelven por sustitución o aplicando el Teorema 1, se tiene que el coste del algoritmo es lineal con el valor de  $a$ ; esto es  $T(a) \in \Theta(a)$ . Por otra parte, el coste espacial del algoritmo es también lineal con  $a$ . Sin embargo, es preferible la versión iterativa que se reproduce a continuación, pues aunque los costes temporales de ambas versiones son equivalentes, el coste espacial de la versión recursiva es lineal y el de la versión iterativa es constante, pues sólo se invoca al método una vez, necesitándose un único registro de activación.

```
/** a>=0 y b>=0 */
static int multNatIterativa(int a, int b) {
 int p = 0;
 while (--a>=0) p+=b;
 return p;
}
```

Una mejora del coste temporal de la multiplicación se obtiene cambiando la estrategia utilizada por otra en la que la reducción del tamaño del problema sea dividirlo por 2, en lugar de restarle 1. El algoritmo es muy parecido al que se emplea en la implementación hardware de la multiplicación y se le suele denominar **multiplicación a la rusa**. Las únicas operaciones necesarias para realizar la multiplicación son la suma y la división por 2. En la tabla 12.2 se ilustra el proceso de multiplicación según este método. En las dos primeras columnas se ubican el multiplicando y el multiplicador y se procede dividiendo el primero por 2 y duplicando el segundo. El algoritmo termina cuando en la primera columna se obtiene el valor 1. El resultado de la multiplicación es la suma de todas las filas en las que el número de la izquierda sea impar.

|     |        |         |
|-----|--------|---------|
| 981 | 1234   | 1234    |
| 490 | 2468   |         |
| 245 | 4936   | 4936    |
| 122 | 9872   |         |
| 61  | 19744  | 19744   |
| 30  | 39488  |         |
| 15  | 78976  | 78976   |
| 7   | 157952 | 157952  |
| 3   | 315904 | 315904  |
| 1   | 631808 | 631808  |
|     |        | 1210554 |

**Tabla 12.2:** Multiplicación a la rusa [BB97].

La transcripción a Java de la versión recursiva del algoritmo de multiplicación a la rusa es la siguiente:

```
/** a>=0 y b>=0 */
static int multNat3(int a, int b) {
 if (a==0) return 0;
 else if (a%2==0) return multNat3(a/2,b*2);
 else return multNat3(a/2,b*2) + b;
}
```

Para plantear las ecuaciones de recurrencia del mismo, obsérvese que la recursión es lineal, el tamaño del problema se divide por la mitad en cada llamada y el

coste del resto de operaciones continúa siendo constante. Luego la ecuación de recurrencia que describe la función de coste temporal es:

$$T(a) = \begin{cases} T(a/2) + 1 & \text{si } a > 0 \\ 1 & \text{si } a = 0 \end{cases}$$

Resolviendo esta ecuación por sustitución se tiene:

$$\begin{aligned} T(a) &= T(a/2) + 1 \\ &= T(a/2^2) + 2 \\ &= T(a/2^3) + 3 \\ &\dots \\ &= T(a/2^i) + i \\ &\dots \\ &= T(a/2^{\lfloor \log_2(a) \rfloor}) + \lfloor \log_2(a) \rfloor \\ &= T(1) + \lfloor \log_2(a) \rfloor \\ &= T(0) + 1 + \lfloor \log_2(a) \rfloor \\ &= 2 + \lfloor \log_2(a) \rfloor \in \Theta(\log a) \end{aligned}$$

Nótese que  $\lfloor \alpha \rfloor$  es el mayor entero que es menor o igual a  $\alpha$ .

En este caso, se tiene que el coste del algoritmo es logarítmico con respecto al valor de  $a$ ,  $T(a) \in \Theta(\log a)$ . Resultado que se obtiene también de aplicar el Teorema 3 con  $a = 1$  y  $c = 2$ .

Además, el coste espacial del algoritmo es también logarítmico con respecto a  $a$ , pues si en total llega a haber  $\log_2(a)$  llamadas recursivas a la vez, el número de registros de activación necesarios es  $\log_2(a)$  y el consumo de memoria en la pila es proporcional a este valor.

En la práctica, se suele implementar el algoritmo usando desplazamientos binarios en lugar de las operaciones de división y multiplicación por 2 que aparecen en el mismo. En la versión que se muestra a continuación se han sustituido los operadores multiplicativos por los de desplazamiento binario existentes en Java:

```
/** a>=0 y b>=0 */
static int multNat3(int a, int b) {
 if (a==0) return 0;
 else if (a%2==0) return multNat3(a>>1,b<<1);
 else return multNat3(a>>1,b<<1) + b;
}
```

Para comparar la diferencia real temporal existente entre los algoritmos de multiplicación que se están considerando y la propia multiplicación de enteros que

utiliza el sistema (máquina virtual Java), en la tabla 12.3 se presenta el resultado de la ejecución de un programa en el que, para diferentes tallas (valores del multiplicando), se ha medido el tiempo de ejecución (en  $\mu$ segundos) de las siguientes versiones de la multiplicación:

- Multiplicación del sistema.
- Multiplicación rápida (a la rusa implementada iterativamente).
- Multiplicación rápida (a la rusa implementada recursivamente).
- Multiplicación lenta (mediante sumas, implementada iterativamente).

Como puede verse, la talla se incrementa multiplicándola por 2 en cada ocasión y tomando el valor inmediatamente anterior, lo que maximiza el número de sumas en las versiones del producto a la rusa ya que el multiplicando será siempre impar; esto es, cada uno de los valores de la talla considerados son del tipo  $2^i - 1$  con  $i$  variando desde 8 hasta 30. En lugar de utilizar en los algoritmos valores `int`, se han utilizado valores `long` para poder operar con números mas grandes.

Las conclusiones son evidentes y pueden resumirse en que:

- La multiplicación del sistema se efectúa en tiempo constante y es del orden de 1 nanosegundo.
- Las multiplicaciones rápidas (a la rusa) presentan un crecimiento logarítmico (obsérvese que cuando la talla se eleva al cuadrado, por ejemplo al pasar de 32767 a 1073741823, el tiempo de ejecución de la versión iterativa se duplica al pasar de 0.0796  $\mu$ segundos a 0.1564  $\mu$ segundos, mientras que la recursiva pasa de 0.0973  $\mu$ segundos a 0.1876  $\mu$ segundos).
- El tiempo de ejecución de la versión recursiva del producto rápido sólo es ligeramente peor que el correspondiente a la versión iterativa.
- El producto realizado mediante sumas es muchísimo más lento (efectuar uno de ellos puede llegar a costar varios segundos), presentando, como puede verse, un crecimiento claramente peor (es lineal) que el resto de algoritmos.

| Talla      | Sistema | Rusa (iter) | Rusa (rec) | Lenta        |
|------------|---------|-------------|------------|--------------|
| 255        | 0.0014  | 0.0443      | 0.0547     | 1.1594       |
| 511        | 0.0013  | 0.0489      | 0.0601     | 1.4387       |
| 1023       | 0.0013  | 0.0541      | 0.0709     | 2.7168       |
| 2047       | 0.0013  | 0.0591      | 0.0729     | 5.2870       |
| 4095       | 0.0013  | 0.0645      | 0.0785     | 10.3854      |
| 8191       | 0.0013  | 0.0693      | 0.0904     | 20.5962      |
| 16383      | 0.0013  | 0.0749      | 0.0913     | 39.8933      |
| 32767      | 0.0013  | 0.0796      | 0.0973     | 83.6699      |
| 65535      | 0.0013  | 0.0845      | 0.1084     | 159.0635     |
| 131071     | 0.0013  | 0.0901      | 0.1092     | 317.9454     |
| 262143     | 0.0013  | 0.0951      | 0.1148     | 682.0420     |
| 524287     | 0.0013  | 0.0998      | 0.1268     | 1353.5102    |
| 1048575    | 0.0013  | 0.1044      | 0.1286     | 2713.1311    |
| 2097151    | 0.0013  | 0.1103      | 0.1330     | 5361.2963    |
| 4194303    | 0.0013  | 0.1155      | 0.1456     | 10750.7790   |
| 8388607    | 0.0013  | 0.1201      | 0.1467     | 21531.9124   |
| 16777215   | 0.0013  | 0.1249      | 0.1513     | 42943.7404   |
| 33554431   | 0.0013  | 0.1305      | 0.1635     | 85950.4766   |
| 67108863   | 0.0013  | 0.1354      | 0.1645     | 171817.9069  |
| 134217727  | 0.0013  | 0.1402      | 0.1702     | 343536.5468  |
| 268435455  | 0.0013  | 0.1459      | 0.1809     | 686936.4959  |
| 536870911  | 0.0013  | 0.1508      | 0.1832     | 1374979.2793 |
| 1073741823 | 0.0013  | 0.1564      | 0.1876     | 2748884.3447 |

Tabla 12.3: Tiempo ( $\mu$ segs) según tipo de multiplicación.

### 12.8.2 Exponenciación modular

Se trata de calcular  $X^N \bmod P$ . Este cálculo se puede hacer de forma trivial multiplicando repetidamente por  $X$  y aplicando el operador `%` cada vez. Esta forma es mejor que efectuar una única operación módulo tras calcular  $X^N$ , gracias a que los resultados parciales se mantienen más pequeños. Y es correcto por la propiedad:

$$X * Y \bmod P = (X \bmod P) * (Y \bmod P) \bmod P$$

Si  $N$  es un número de, por ejemplo, 100 dígitos, la multiplicación es impracticable. Un algoritmo más rápido se basa en el siguiente análisis por casos:

- Si  $N$  es par, entonces  $X^N = (X * X)^{\lfloor N/2 \rfloor}$ .
- Si  $N$  es impar, entonces  $X^N = X * X^{N-1} = X * (X * X)^{\lfloor N/2 \rfloor}$ .

Recuérdese que  $\lfloor \alpha \rfloor$  es el mayor entero que es menor o igual a  $\alpha$ .

Una implementación directa de esta estrategia, tomando como caso base  $X^0 = 1$  y considerando la propiedad anterior, da lugar al siguiente algoritmo recursivo:

```
/** x>0, n>=0 y p>1 */
static long potencia(long x, long n, long p) {
 if (n==0) return 1;
 else {
 long aux = potencia(x*x%p, n/2, p);
 if (n%2!=0) aux = (aux*x)%p;
 return aux;
 }
}
```

El número total de llamadas que se producen depende del valor de  $n$  que, en este caso, es la talla del problema. Para plantear las ecuaciones de recurrencia, obsérvese que la recursión es lineal, el tamaño del problema se divide por la mitad en cada llamada y el coste del resto de operaciones continúa siendo constante. Se tiene:

$$T(n) = \begin{cases} T(n/2) + 1 & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Si estas ecuaciones se resuelven por sustitución, como en el último caso del problema anterior, entonces se tiene que el coste del algoritmo es logarítmico con respecto al valor de  $n$ ; esto es  $T(n) \in \Theta(\log n)$ . Por tratarse de un algoritmo recursivo que necesita apilar registros de activación, el coste espacial del algoritmo es también logarítmico con respecto a  $n$ .

## 12.9 Problemas propuestos

1. Considerar dos algoritmos  $A$  y  $B$  que resuelven el mismo problema y cuyos costes respectivos son  $1000n$  y  $n^2$  (en unidades de tiempo), dónde  $n$  es el tamaño del problema. ¿Para qué valores de  $n$  es preferible cada uno de los algoritmos anteriores?
2. Los algoritmos con coste exponencial son impracticables, y este hecho es independiente de los avances tecnológicos. Supóngase que en un determinado sistema, para un algoritmo de coste  $\Theta(k^n)$  los tiempos de ejecución son aceptables hasta una cierta talla  $m$ . Demuéstrese que si la velocidad del procesador mejora en un factor  $c$ , en el mismo tiempo que antes se resolvía el problema para talla  $m$ , se resuelve ahora para una talla  $m + \log_k(c)$ , y que la mejora es, por tanto, despreciable. ¿Cuál sería la mejora para un algoritmo de coste cuadrático? ¿Y cúbico?
3. El algoritmo siguiente encuentra la posición del elemento más pequeño en un array de  $n$  enteros ( $n \geq 1$ ):

```
/** n>=1 */
static int posMinimo(int[] v) {
 int pos, min;
 pos = 0; min = v[pos];
 for (int i=1; i<n; i++)
 if (v[i]<min) { pos = i; min = v[pos]; }
 return pos;
}
```

- a) ¿De qué va a depender el coste temporal del método `posMinimo`? Y, por lo tanto, ¿cuál es el tamaño del problema?
- b) ¿Existen instancias significativas?
- c) ¿Cuál podría ser la instrucción crítica en este algoritmo? ¿Cuántas veces se repite?
- d) ¿Cuál es la complejidad temporal de `posMinimo` expresada en términos del número de veces que se repite la instrucción crítica? ¿Cuál es su complejidad asintótica?

4. En la tabla siguiente se muestran los tiempos de ejecución, en segundos, de cierto algoritmo para los valores de las tallas que se muestran en la primera columna:

| #      | Talla     | Tiempo (seg.) |
|--------|-----------|---------------|
| #----- |           |               |
| 10000  | 0.0497889 |               |
| 20000  | 0.2023334 |               |
| 30000  | 0.4544204 |               |
| 40000  | 0.8040325 |               |
| 50000  | 1.2702825 |               |
| 60000  | 1.8414770 |               |
| 70000  | 2.5063077 |               |
| 80000  | 3.2536286 |               |
| 90000  | 4.1410572 |               |
| 100000 | 5.2779916 |               |

- a) Desde un punto de vista asintótico, ¿cuál es la función de coste temporal que aproxima de forma más precisa los valores de la columna Tiempo como función de los de la columna Talla? ¿Por qué?
- b) De forma aproximada, ¿cuánto tiempo tardará el algoritmo anterior en ejecutarse para una talla de 500000 elementos?
5. Estudiar la complejidad asintótica de los algoritmos iterativos obtenidos de la implementación directa (sin mejoras especiales) de las definiciones de las siguientes operaciones:
- a) Producto escalar de dos arrays de dimensión  $n$ .
- b) Suma de dos matrices cuadradas de dimensión  $n \times n$ .
- c) Producto de matrices cuadradas de dimensión  $n \times n$ .
6. El siguiente algoritmo calcula la suma de las componentes de un array de  $n$  naturales ( $n \geq 1$ ):

```
/** n>=1 */
static int suma(int[] v) {
 int s = 0;
 for (int i=0; i<n; i++)
 for (int j=0; j<v[i]; j++)
 s++;
 return s;
}
```

- a) ¿Se puede considerar como instrucción crítica de este método la instrucción más interna de los dos bucles `s++`? ¿Por qué?
- b) ¿Cuál es la instrucción crítica para este algoritmo?
- c) ¿Cuál es el coste de este algoritmo en términos del número de veces que se repite la instrucción crítica? ¿Cuál es su complejidad asintótica?

7. Considerar los dos algoritmos siguientes:

**Algoritmo 1:**  
for (int i=0; i<n; i++) {  
 w[i] = 0;  
 for (int j=0; j<i+1; j++)  
 w[i]+=v[j];  
}

**Algoritmo 2:**  
w[0] = v[0];  
for (int i=1; i<n; i++) {  
 w[i] = w[i-1] + v[i];  
}

- a) ¿Qué hacen estos dos algoritmos?
  - b) Calcular la complejidad asintótica de estos dos algoritmos.
  - c) ¿A qué se debe que las complejidades obtenidas no coincidan?
8. Considerar el siguiente segmento de código perteneciente a un método numérico:

```
// a, b y m son matrices de números reales, cuadradas, con
// la misma dimensión
int dimension = m.length;
for (int i=0; i<dimension; i++)
 for (int j=0; j<dimension; j++) {
 m[i,j] = 0;
 for (int k=0; k<dimension; k++)
 m[i,j] = m[i,j]+a[i,k]*b[k,j];
 // m contiene el producto de las matrices a * b
```

- a) Estudiar cuál es la talla del problema.
- b) Estudiar el coste temporal del algoritmo utilizando para ello como unidad de coste temporal el paso de programa.
- c) Estudiar el coste temporal del algoritmo utilizando para ello como unidad de coste temporal la instrucción crítica. ¿Qué instrucción crítica se ha elegido?
- d) Expressar ambos costes asintóticamente. ¿Hay alguna diferencia asintótica entre las expresiones?

9. Dado el siguiente método:

```
static int cifras(int[] v, int m) {
 int i = 0, q = m;
 while (q>0) {
 i++;
 v[i] = q%10;
 q = q/10;
 }
 return i;
}
```

- a) Estimar su coste temporal si se toma como talla el valor de  $m$ .
  - b) Estimar su coste temporal si se decide tomar como talla el número de cifras de  $m$ .
  - c) ¿Hay alguna contradicción entre los resultados de los dos apartados anteriores? Razonar la respuesta.
10. Escribir un método iterativo que encuentre el  $k$ -ésimo elemento más pequeño de un array de  $n$  enteros. Calcular la complejidad temporal del algoritmo diseñado.
11. Sobre un array de  $n$  enteros,  $n \geq 1$ , ordenado de forma creciente, se definen los siguientes métodos que no están implementados:

- **frec(x, v, i, j)**: devuelve el número de veces que aparece el valor  $x$  en el subarray  $v[i..j]$ .
- **moda(v, i, j)**: que devuelve el valor que aparece más veces en el subarray  $v[i..j]$ , es decir, aquel  $x$  cuya **frec(x, v, i, j)** sea máxima. Si existen varios valores con frecuencia máxima, cualquiera de ellos es la moda.

Se debe diseñar un método que dado  $v$ , escriba por pantalla la moda de este array y su frecuencia. Para ello, se recorrerá secuencialmente el array, usando las variables  $i$ ,  $x$ ,  $f$ , y  $fu$ , tales que:

- $i$  marca el elemento que se está revisando, y  $fu$  es su frecuencia.
- $x$  servirá para recordar la moda de los elementos ya revisados y  $f$  será la frecuencia de  $x$  en ese subarray.

En el algoritmo sólo se podrá acceder una vez a cada elemento de  $v$ . Analizar la complejidad del algoritmo propuesto.

12. Estudiar la complejidad temporal del siguiente algoritmo recursivo:

```
/** i>=0 */
static int maximo(int[] v, int i) {
 if (i==0) return v[i];
 else { int m = maximo(v,i-1);
 if (m>v[i]) return m;
 else return v[i];
 }
}
```

Supóngase que *v* es un array creado con *n* enteros y que la llamada inicial al método se realiza como `máximo(v,n-1)`.

13. El siguiente método recursivo calcula la suma de los elementos del array *v* comprendidos entre las posiciones *ini* y *fin*:

```
/** 0<=ini<=fin */
static int suma(int[] v, int ini, int fin) {
 if (ini==fin) return v[ini];
 else { int m = (ini+fin)/2;
 return suma(v,ini,m) + suma(v,m+1,fin);
 }
}
```

Calcular el coste temporal del algoritmo, para lo que se tendrá que:

- a) Definir la talla del problema.
- b) Determinar si hay diferentes instancias y, caso de haberlas, describir cuáles son las más significativas.
- c) Plantear las ecuaciones de recurrencia con el fin de estudiar el coste temporal en las distintas instancias significativas, si las hubiere,
- d) Resolverlas por el método de sustitución.
- e) Expresar el coste obtenido utilizando, para ello, notación asintótica.

Estudiar si es posible modificar el algoritmo para obtener una complejidad logarítmica para el mismo. Justificar la respuesta.

14. Dado el siguiente método recursivo:

```
/** 0<a y 0<=b */
static int potencia(int a, int b) {
 if (b==0) return 1;
 else if (b==1) return a;
 else if (b%2==0)
 return potencia(a,b/2) * potencia(a,b/2);
 else
 return potencia(a,b/2) * potencia(a,b/2) * a;
}
```

- a) Estudiar la complejidad temporal de este método.
- b) La función admite una mejora obvia. ¿En qué afecta al coste esta posible modificación?
15. Considerar el siguiente segmento de código:
- ```
// n contiene un valor entero no negativo
int alea = (int)(Math.random()*n);
int acum = 0;
for (int i=0; i<alea; i++) acum++;
for (int j=alea; j<n; j++) acum--;
```
- a) Estudiar cuál es la talla del problema.
- b) Estudiar el coste temporal del algoritmo utilizando para ello como unidad de coste temporal el paso de programa.
- c) Si se desea estudiar el coste utilizando como unidad la instrucción crítica, es necesario seleccionar una en primer lugar. ¿Existe una instrucción crítica para este código?
16. Estudiar el coste espacial del método recursivo para calcular el n -ésimo número de Fibonacci.
17. Considerar el método de multiplicación a la rusa visto al final de este capítulo.
- a) Estudiar su coste espacial.
- b) Escribir el mismo algoritmo de multiplicación, pero resolviéndolo iterativamente en lugar de recursivamente.
- c) ¿Cuál es el coste espacial de la versión iterativa?
- d) ¿Se puede utilizar el mismo razonamiento que se ha seguido para calcular el coste temporal de la versión recursiva si se desea calcular el de la iterativa?, esto es, ¿se pueden plantear ecuaciones de recurrencia para el cálculo del coste temporal del algoritmo iterativo?

Más información

- [BB97] G. Brassard and P. Bratley. *Fundamentos de Algoritmia*. Prentice Hall, 1997. Capítulos 2, 3 y 4.
- [FAM98] F.J. Ferri, J.V. Albert, and G. Martín. *Introducció a l'anàlisi i disseny d'algorismes*. Universitat de València, 1998. Capítulos 2 y 3.
- [Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulo 5.

Capítulo 13

Ordenación y otros algoritmos sobre arrays

Dada una secuencia de n elementos, $\{a_0, a_1, \dots, a_{n-1}\}$, el problema de la ordenación se define como la obtención de una permutación $\{a'_0, a'_1, \dots, a'_{n-1}\}$ de dicha secuencia que cumpla que $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$, donde \leq es una relación de orden definida entre los elementos de la misma.

Existen diversas estrategias para resolver este problema pero, del mismo modo, los tiempos que invierten para ello pueden llegar a ser muy diferentes entre sí, dependiendo en todos los casos del número de elementos a ordenar y, en algunos, además, de su disposición en la secuencia inicial.

Los algoritmos de ordenación, atendiendo a distintos criterios, pueden clasificarse en:

- Algoritmos de ordenación *internos* o *externos* si los datos a ordenar se encuentren en la memoria interna o en la externa, respectivamente.
- Algoritmos de ordenación *directos* o *rápidos* si el coste de ordenar n elementos es cuadrático o $n \log n$, respectivamente.
- Algoritmos de ordenación *estables* o *inestables* si el resultado de la ordenación mantiene o no los elementos iguales en el mismo orden que en la secuencia original.
- Algoritmos de ordenación *naturales* o *no naturales* si su coste es menor cuando la secuencia de entrada está ya (casi) ordenada o cuando la secuencia está ordenada en sentido inverso, respectivamente.

En este capítulo se presentan sólo algoritmos de ordenación internos y otros algoritmos sobre arrays y se estudia su eficiencia. En concreto, los algoritmos de ordenación directos *selección*, *inserción* e *intercambio*, el algoritmo de ordenación rápido *mergesort*, el algoritmo de *mezcla natural* y el de *búsqueda binaria*. Resulta muy importante estudiar la eficiencia de los algoritmos de ordenación citados porque, por ejemplo, si en 1 segundo, mergesort puede ordenar X elementos, el algoritmo de inserción directa necesitaría Y horas.

Para el estudio de los distintos algoritmos de ordenación, se asume que los elementos a ordenar se encuentran almacenados en un array y que sobre ellos se define una relación de orden natural que en el caso de los tipos básicos será \leq y en los tipos referencia se implementará como un método de comparación definido en la clase (por ejemplo, el método dinámico `compareTo` de la clase `String` o de las clases envolventes) y que la ordenación se desea realizar sin utilizar ninguna otra estructura de datos auxiliar.

13.1 Selección directa

El algoritmo de *selección directa* consiste básicamente en lo siguiente:

1. Seleccionar el elemento mínimo del array y situarlo en la posición 0.
2. Seleccionar el siguiente elemento más pequeño y colocarlo a continuación.
3. Repetir este proceso hasta que sólo quede 1 elemento, que necesariamente tiene que ser el elemento máximo del array, bien situado al final de éste.

El algoritmo de selección directa se puede describir de manera más precisa utilizando un índice *i*, indicando que en cada iteración del bucle se selecciona el *i*-ésimo elemento más pequeño del array. Dicha variable tomará valores sucesivos entre 0 y *n*-2 (ambos inclusive), siendo *n* el tamaño del array, *v.length*. Esta variable establece una partición en el array de forma que:

- Todos los elementos del array desde la posición 0 hasta la *i-1* están ordenados entre sí y además tienen valores inferiores a los que se encuentran en el subarray comprendido entre *i* y *n-1*.
- El subarray comprendido entre *i* y *n-1* comprende la zona problema en la cuál hay que buscar el mínimo para situarlo en la posición *i*.
- El incremento de *i* nos aproxima a la solución completa del problema, ya que establece que en cada iteración la zona ordenada es más grande y la zona problema más pequeña. En concreto, cuando *i==n-1*, el algoritmo debe terminar.

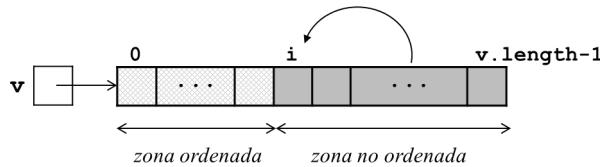


Figura 13.1: Selección del i -ésimo mínimo del array.

La estrategia del algoritmo de selección directa, ilustrada en la figura 13.1, se puede describir de la siguiente manera:

Para i tomando valores desde 0 hasta $n-2$ hacer:

1. Encontrar el mínimo en el subarray $v[i..n-1]$
2. Intercambiarlo con el elemento $v[i]$
3. Incrementar en 1 el valor de i

La operación de encontrar el mínimo en el subarray se puede plantear como un problema de recorrido. Nótese que es necesario saber la posición que ocupa dicho valor para situarlo posteriormente en la posición i del array en el paso 2:

```
int pMin = i; // inicialmente el mínimo está en la posición i
for (int j=i+1; j<v.length; j++)
    if (v[j]<v[pMin]) pMin = j;
```

El método selDirecta

Finalmente, el algoritmo de selección directa se suele implementar en Java como un método estático que tiene como parámetro el array a ordenar y que será público o privado según donde se ubique y el uso que se le dé:

```
static void selDirecta(int[] v) {
    for (int i=0; i<v.length-1; i++) {
        // calcular la posición del mínimo de v[i..v.length-1]
        int pMin = i;
        for (int j=i+1; j<v.length; j++)
            if (v[j]<v[pMin]) pMin = j;
        // en pMin está la posición del mínimo de v[i..v.length-1]

        // intercambiar v[i] con v[pMin]
        int aux = v[pMin];
        v[pMin] = v[i];
        v[i] = aux;
        // desde 0 hasta i están ordenados
    }
    // array ordenado desde 0 hasta v.length-1
}
```

Si, por ejemplo, se quisiera ordenar un array de `String`, el código del método sólo variaría en la comparación de los elementos del array, pues habría que utilizar `compareTo` en lugar de `<`, como sigue:

```
static void selDirecta(String[] v) {
    for (int i=0; i<v.length-1; i++) {
        // calcular la posición del mínimo de v[i..v.length-1]
        int pMin = i;
        for (int j=i+1; j<v.length; j++)
            if (v[j].compareTo(v[pMin])<0) pMin = j;
        // en pMin está la posición
        // del mínimo de v[i..v.length-1]

        // intercambiar v[i] con v[pMin]
        int aux = v[pMin];
        v[pMin] = v[i];
        v[i] = aux;
        // desde 0 hasta i están ordenados
    }
    // array ordenado desde 0 hasta v.length-1
}
```

Nótese que el método de selección directa se puede definir para que sea válido para ordenar arrays de cualquier `tipoBase` siempre que dicho `tipoBase` implemente el método `compareTo`. Esto daría lugar a la definición del método usando la *genericidad* en Java que queda fuera de los objetivos de este libro.

El coste del algoritmo

La talla del problema es el número de elementos a ordenar, $n = v.length$. La estructura del algoritmo consta de dos recorridos anidados y el coste no va a presentar variación con respecto a las posibles instancias del problema. Como instrucción crítica se puede tomar la evaluación de la condición de la instrucción condicional ($v[j] < v[pMin]$). Calculando el número de veces que se repite esta instrucción, el coste del algoritmo será:

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - i - 1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

13.2 Inserción directa

El algoritmo de *inserción directa* consiste básicamente en lo siguiente:

1. Dividir el array en dos partes A y B, de tamaños 1 y $v.length - 1$, respectivamente.
2. Tomar el primer elemento del subarray B y situarlo en el subarray A, de manera que todos los elementos de A respeten la relación de orden \leq .
3. Repetir el proceso hasta que todos los elementos estén en el subarray A.

El algoritmo de inserción directa se puede describir de manera más precisa utilizando un índice i , indicando que en cada iteración del bucle se selecciona el elemento i -ésimo del array a insertar. Dicha variable tomará valores comprendidos entre 1 y $n-1$ (ambos inclusive), siendo n el tamaño del array, $v.length$. Esta variable establece una partición en el array de forma que:

- Todos los elementos del array desde la posición 0 hasta la $i-1$ están ordenados entre sí.
- El subarray comprendido entre i y $n-1$ comprende la zona problema, de la que se toma el primer elemento (el de la posición i) y se inserta en el lugar que le corresponda según la relación \leq entre los elementos de su izquierda.
- El incremento de i nos aproxima a la solución completa del problema, ya que establece que en cada iteración la zona ordenada es más grande, y la zona problema más pequeña. En concreto, cuando $i==n$, el algoritmo debe terminar.

La estrategia del algoritmo de inserción directa se puede describir de la siguiente manera:

Para i tomando valores desde 1 hasta $n-1$ hacer:

1. Insertar el elemento $v[i]$ de manera ordenada en el subarray $v[0..i-1]$.
2. Incrementar en 1 el valor de i .

La inserción ordenada del elemento $v[i]$ en el subarray $v[0..i-1]$ se puede desglosar, a su vez, en las siguientes operaciones:

- a) Realizar una búsqueda secuencial descendente en el intervalo de búsqueda hasta encontrar un elemento menor o igual a $v[i]$; sea j el índice de dicho elemento.
- b) Desplazar todos los elementos entre las posiciones $j+1$ y $i-1$ una posición a la derecha.
- c) Almacenar el elemento a insertar en la posición $v[j]$.

Nótese que los pasos a) y b) pueden hacerse a la vez e implementarse como sigue:

```
int x = v[i];    // elemento a insertar
int j = i-1;    // posición final de la parte ordenada
while (j>=0 && v[j]>x) {
    v[j+1] = v[j];
    j--;
}
```

En la figura 13.2 se representa gráficamente la estrategia de la inserción directa.

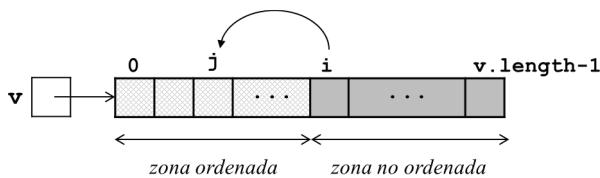


Figura 13.2: Inserción del elemento i en el subarray ordenado.

El método insDirecta

Finalmente, el algoritmo de inserción directa se suele implementar en Java como un método estático que tiene como parámetro el array a ordenar y que será público o privado según donde se ubique y el uso que se le dé:

```
static void insDirecta(int[] v) {
    for (int i=1; i<v.length; i++) {
        int x = v[i]; // elemento a insertar
        int j = i-1; // posición final de la parte ordenada

        // buscar en la parte ordenada,
        // desplazando a la derecha los elementos mayores que x
        while (j>=0 && v[j]>x) {
            v[j+1] = v[j];
            j--;
        }

        v[j+1] = x; // almacenar x en la parte ordenada
    }
}
```

El coste del algoritmo

La talla del problema es el número de elementos a ordenar, $n = v.length$. El algoritmo se estructura como una búsqueda anidada dentro de un recorrido. El bucle de búsqueda se ejecuta un número de veces que puede variar según la entrada del problema (nótese que en la guarda se evalúa la condición $v[j] > x$). Es por esto que el coste presenta las siguientes instancias significativas:

Caso mejor: El número de iteraciones del bucle de búsqueda `while` es 0. Esto ocurre cuando $v[j-1] \leq v[j]$ para todos los valores posibles de j , es decir, $j = 1..n-1$. Esta condición define un array ya ordenado.

Caso peor: El número de iteraciones del bucle de búsqueda es el máximo, i . Esto ocurre si el elemento a insertar debe ir a la posición 0 del array en todas las iteraciones del bucle externo. Esta condición define un array ordenado de forma decreciente.

Si se toma la guarda del bucle de búsqueda como instrucción crítica, entonces:

Caso mejor: $T^m(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$.

Caso peor: $T^p(n) = \sum_{i=1}^{n-1} \sum_{j=-1}^{i-1} 1 = \sum_{i=1}^{n-1} (i + 1) = \frac{n(n+1)}{2} - 1 \in \Theta(n^2)$.

Nótese que, a la vista del coste obtenido en cada caso, inserción directa es un método de ordenación natural. A partir de las funciones $T^p(n)$ y $T^m(n)$ obtenidas, se puede decir que la función de coste temporal del algoritmo de inserción directa $T(n)$ está acotada como sigue:

$$T(n) \in \Omega(n) \cap O(n^2)$$

13.3 Intercambio directo o algoritmo de la burbuja

El algoritmo de *intercambio directo* (o algoritmo de la *burbuja*) consiste básicamente en lo siguiente:

1. Aplicación reiterada de un algoritmo iterativo de recorrido descendente.
2. El elemento que ocupa la posición j del array:
 - a) Se compara con el elemento de su izquierda (posición $j - 1$).
 - b) Si no están correctamente ordenados entre sí, se intercambian.
3. Repetir el proceso $n-1$ veces.

El algoritmo de intercambio directo se puede describir de forma más precisa utilizando una variable *i* como contador del número de recorridos realizados. Cada recorrido garantiza que, a su término, el elemento *i*-ésimo más pequeño del array se encuentra correctamente situado en la posición *i*-ésima del mismo. Dicha variable tomará valores comprendidos entre 0 y *n*-2 (ambos inclusive), siendo *n* el tamaño del array, *v.length*. Esta variable establece una partición en el array de manera similar a cómo lo hace el algoritmo de selección directa:

- Todos los elementos del array desde la posición 0 hasta la *i*-1 están ordenados entre sí y además tienen valores inferiores a los que se encuentran en el subarray comprendido entre *i* y *n*-1.
- El subarray comprendido entre *i* y *n*-1 comprende la zona problema en la que, a base de intercambios, el mínimo se traslada a la posición *i*.
- El incremento de *i* nos aproxima a la solución completa del problema, ya que establece que en cada iteración la zona ordenada es más grande, y la zona problema más pequeña. En concreto, cuando *i==n-1*, el algoritmo debe terminar.

Cada recorrido sobre la zona problema va desde la posición *n*-1 hasta la *i*+1 (cada una de esas *j* posiciones se compara con la que está a su izquierda *j*-1).

El método burbuja

El algoritmo de intercambio directo también se puede implementar en Java como un método estático que tiene como parámetro el array a ordenar y que será público o privado según donde se ubique y el uso que se le dé:

```
static void burbuja(int[] v) {  
    for (int i=0; i<v.length-1; i++)  
        // situar en la posición i el mínimo de v[i...v.length-1]  
        for (int j=v.length-1; j>i; j--)  
            // comprobar si todo par de elementos  
            // consecutivos está ordenado  
            if (v[j-1]>v[j]) {  
                // si no están en orden, intercambiar  
                int x = v[j];  
                v[j] = v[j-1];  
                v[j-1] = x;  
            }  
        // desde 0 hasta i-1 están ordenados  
    // array ordenado desde 0 hasta v.length-1  
}
```

En la figura 13.3 se representa gráficamente la situación del array en una iteración intermedia del bucle interno.

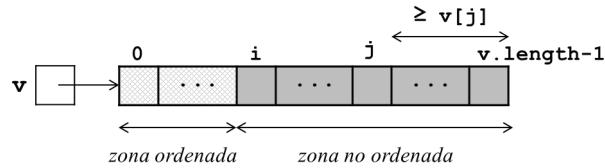


Figura 13.3: Intercambio del elemento j del array.

El coste del algoritmo

La talla del problema es el número de elementos a ordenar, $n = v.length$. La estructura del algoritmo consta de dos recorridos anidados y no presenta instancias significativas para el coste. Como instrucción crítica se puede tomar la evaluación de la condición de la instrucción condicional ($v[j-1] > v[j]$). Si se calcula el número de veces que se repite esta instrucción, el coste temporal del algoritmo será:

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - i - 1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

13.4 Ordenación por mezcla o mergesort

El algoritmo de *ordenación por mezcla* o *mergesort* es un algoritmo recursivo consistente en lo siguiente:

1. Dividir el array por la mitad, dando lugar a dos subarrays del mismo tamaño.
2. Ordenar cada subarray por separado mediante una llamada recursiva.
3. Fusionar ambas partes (ya ordenadas) respetando la relación de orden.

El caso base de este algoritmo se plantea para un array con un solo elemento. En tal caso, no se hace nada ya que dicho array, por definición, ya está ordenado.

Para identificar el subarray asociado a cada una de las llamadas recursivas, se usan dos índices `inicio` y `fin` como parámetros adicionales de este algoritmo, que indicarán así las posiciones extremas que definen cada subarray problema. La llamada inicial, encargada de ordenar todo el array, se instancia, por tanto, con los siguientes valores: `inicio = 0` y `fin = v.length-1`.

Para el paso 3 del algoritmo, se hace uso del algoritmo de *mezcla natural*, descrito y analizado a continuación en la sección 13.5.1 de este mismo capítulo, cuyo coste, lineal con el tamaño del array, no presenta instancias significativas.

El método mergesort

Finalmente, el algoritmo recursivo mergesort se puede implementar en Java como un método estático lanzadera que llama a uno privado que tiene como parámetros el array *v* y los índices *inicio* y *fin* que definen el (sub)array a ordenar:

```
static void mergesort(int[] v) { mergesort(v,0,v.length-1); }

private static void mergesort(int[] v, int inicio, int fin) {
    if (inicio<fin) {
        int mitad = (fin+inicio)/2;

        mergesort(v, inicio, mitad);
        mergesort(v, mitad+1, fin);

        // versión de mezcla natural especializada:
        // mezcla en v[inicio..fin] de
        // v[inicio..mitad] y v[mitad+1..fin]
        mezclaNatural2(v, inicio, mitad, fin);
    }
}
```

En la figura 13.4 se ilustra la estrategia del método *mergesort*.

El coste del algoritmo

La talla del problema es el número de elementos a ordenar, $n = \text{fin}-\text{inicio}+1$, por consiguiente, la talla del problema en la llamada inicial es $n = \text{v.length}$. El coste de mergesort es independiente de las posibles instancias del problema. Para plantear las ecuaciones de recurrencia, obsérvese que la recursión es múltiple y, para un problema de talla n , el algoritmo genera 2 nuevas llamadas de talla $\frac{n}{2}$, y 1 llamada de talla n al algoritmo de mezcla natural, cuyo coste es $2n$. Luego la ecuación de recurrencia para obtener la función de coste temporal es:

$$T(n) = \begin{cases} 2T(n/2) + 2n & \text{si } n > 1 \\ 1 & \text{si } 0 \leq n \leq 1 \end{cases}$$

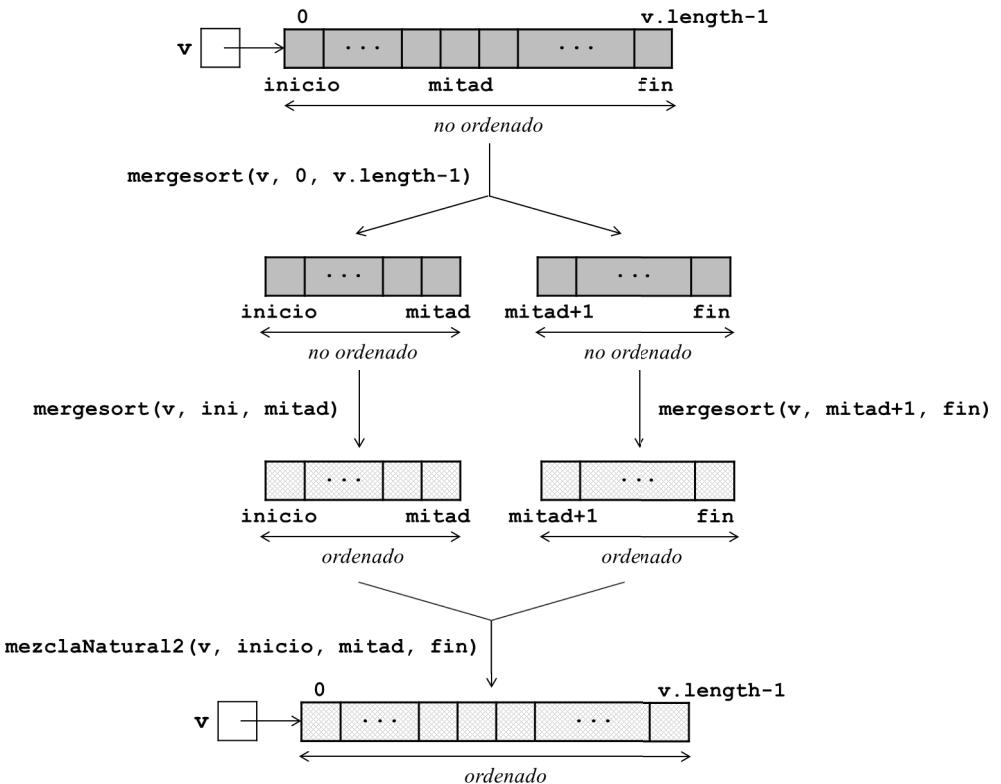


Figura 13.4: Ordenación por mergesort.

Resolviendo esta ecuación por sustitución se tiene:

$$\begin{aligned}
 T(n) &= 2T(n/2) + 2n \\
 &= 4T(n/2^2) + 4n \\
 &= 8T(n/2^3) + 6n \\
 &\dots \\
 &= 2^i T(n/2^i) + i2n \\
 &\dots \\
 &= nT(n/2^{\lfloor \log_2(n) \rfloor}) + \lfloor \log_2(n) \rfloor 2n \\
 &= nT(1) + 2n\lfloor \log_2(n) \rfloor = n + 2n\lfloor \log_2(n) \rfloor \in \Theta(n \log n)
 \end{aligned}$$

13.5 Otros algoritmos sobre arrays

13.5.1 El algoritmo de mezcla natural

El algoritmo de *fusión* o *mezcla natural* resuelve el siguiente problema: dados dos arrays **a** y **b**, previamente ordenados según una relación de orden \leq , se desea construir un nuevo array, **c**, que contenga los elementos de **a** y de **b**, respetando a su vez la relación de orden \leq .

La estrategia para abordar la solución a este problema es la siguiente:

1. Bucle de recorrido ascendente sobre los dos arrays que compara los elementos de los dos arrays y los copia de forma ordenada en el array destino. Cuando **a** o **b** se ha procesado completamente, el bucle acaba.
2. Bucle que copia, ya sin comparar, el resto de elementos del array a medio recorrer en el array destino.

El método mezclaNatural

Finalmente, el algoritmo de mezcla natural se puede implementar en Java como un método que recibe los tres arrays, **a**, **b** y **c**, como parámetros:

```
/** a y b están ordenados
 * c.length es a.length + b.length
 */
static void mezclaNatural(int[] a, int[] b, int[] c) {
    int i = 0; // índice para recorrer a
    int j = 0; // índice para recorrer b
    int k = 0; // índice para recorrer c

    while (i<a.length && j<b.length) {
        if (a[i]<b[j])
            { c[k] = a[i]; i++; }
        else { c[k] = b[j]; j++; }
        k++;
    }

    for (int r=i; r<a.length; r++, k++) c[k] = a[r];
    for (int r=j; r<b.length; r++, k++) c[k] = b[r];
}
```

En la figura 13.5 se representa gráficamente una traza del método mezclaNatural.

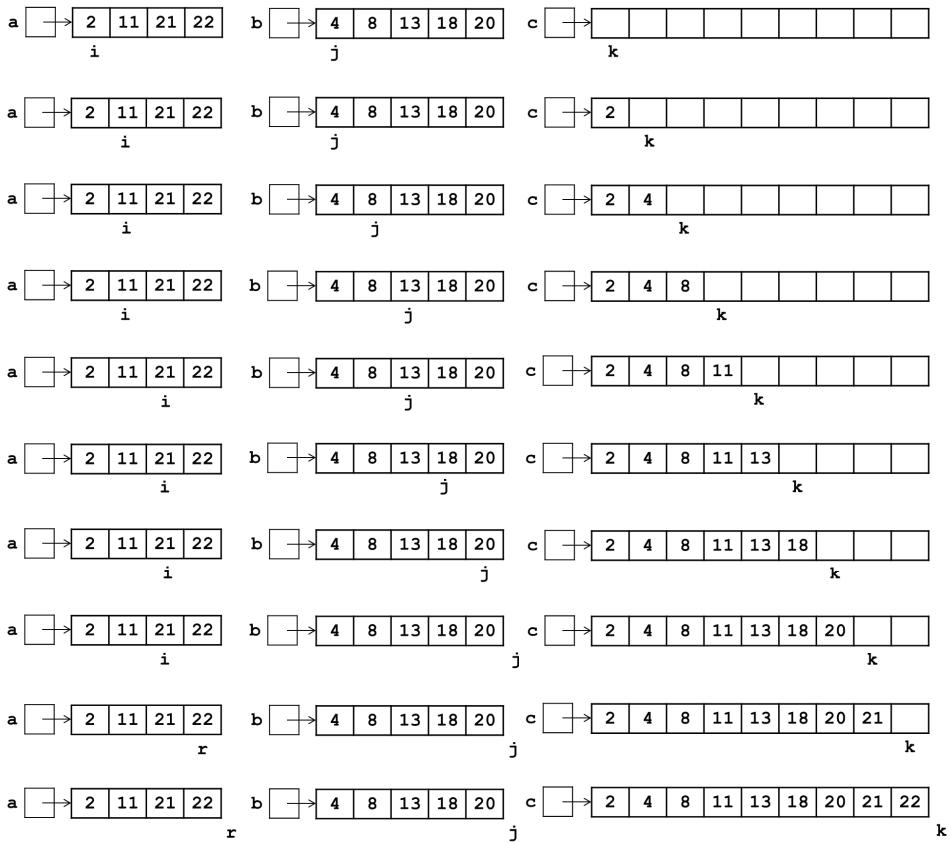


Figura 13.5: Traza del método `mezclaNatural`.

El coste del algoritmo

La talla del problema es la suma de los tamaños de los arrays de entrada, $n = a.length + b.length$, lo que coincide con el tamaño del array resultado. Para estudiar el coste temporal de este algoritmo, se puede tomar como instrucción crítica el incremento de la variable ($k++$) para el recorrido del array c . Contando el número de veces que se ejecuta esta instrucción, el coste temporal del algoritmo será:

$$T(n) = n \in \Theta(n)$$

Implementación eficiente del algoritmo para su uso por mergesort

El algoritmo mergesort (véase la sección 13.4) utiliza el de mezcla natural para fusionar las dos mitades de un (sub)array (previamente ya ordenadas) de manera que el (sub)array quede completamente ordenado de principio a fin.

Por tanto, la diferencia con respecto al algoritmo presentado anteriormente es que, en este caso, todos los datos de entrada y de salida del problema se gestionan mediante un único contenedor de información, el array a ordenar. Por un lado, las secuencias de entrada conforman un (sub)array dentro de éste, que se almacenan en sus dos respectivas mitades, una a continuación de la otra. Por lo tanto, quedan identificadas mediante una tripleta de índices en el array v (`inicio`, `mitad`, `fin`) que definen la primera secuencia ya ordenada como $v[inicio..mitad]$, mientras que la segunda secuencia ya ordenada se encuentra en $v[mitad+1..fin]$. Por otro lado, el resultado de mezclar las dos mitades del (sub)array a ordenar debe quedar en el propio (sub)array, es decir, como una permutación del mismo.

Finalmente, el algoritmo de mezcla natural para mergesort se implementa en Java como un método parametrizado con el array v como contenedor de información y los 3 índices para las 2 particiones de datos ordenadas independientemente:

```
static void mezclaNatural2(int[] v, int inicio, int mitad, int fin) {
    int i = inicio; // índice para recorrer v[inicio..mitad]
    int j = mitad+1; // índice para recorrer v[mitad+1..fin]
    int[] aux = new int[fin-inicio+1];
    int k = 0; // índice para recorrer aux

    while (i<=mitad && j<=fin) {
        if (v[i]<v[j]) { aux[k] = v[i]; i++; }
        else { aux[k] = v[j]; j++; }
        k++;
    }

    for (int r=i; r<=mitad; r++, k++) aux[k] = v[r];
    for (int r=j; r<=fin; r++, k++) aux[k] = v[r];

    for (k=0, i=inicio; k<aux.length; k++, i++) v[i] = aux[k];
}
```

La talla sigue siendo el número de elementos a ordenar, en este caso $n = \text{fin}-\text{inicio}+1$, y el coste de esta implementación también permanece lineal respecto a n , $T(n) = 2n \in \Theta(n)$.

13.5.2 El algoritmo de búsqueda binaria

El algoritmo de *búsqueda binaria* (o *dicotómica*) es un algoritmo de búsqueda que requiere que el array esté ordenado según una relación de orden \leq . Así, se plantea el problema de buscar un elemento x en un array v , ordenado ascendenteamente, desde una posición `inicio` hasta una posición `fin`, $0 \leq \text{inicio} \leq \text{fin} < a.length$. En la sección 10.3.2 del capítulo 10 se presentó la versión iterativa del mismo. En esta sección se estudiará su coste y se presentará la versión recursiva del algoritmo.

El algoritmo - Versión iterativa

El algoritmo iterativo de búsqueda binaria es el siguiente:

```
/** 0<=inicio<=fin<v.length */
static int busBinaria(int[] v, int inicio, int fin, int x) {
    int izq = inicio, der = fin, mitad = 0;
    boolean encontrado = false;

    while (izq<=der && !encontrado) {
        mitad = (izq+der)/2;
        if (x==v[mitad]) encontrado = true;
        else if (x<v[mitad]) der = mitad-1;
        else izq = mitad+1;
    }

    if (encontrado) return mitad;
    else return -1;
}
```

La llamada inicial si se quiere realizar la búsqueda en todo el array será:
`int pos = busBinaria(a,0,a.length-1,x);`

El coste del algoritmo - Versión iterativa

La talla del problema viene dada por el número de elementos del array, $n = \text{fin}-\text{inicio}+1$. Si la búsqueda se realiza en todo el array, $n = v.length$. El tamaño de la zona de búsqueda queda inicialmente definido por la talla pero, en cada iteración, dicha zona se reduce a la mitad de su tamaño anterior, haciendo que el número máximo de iteraciones del bucle sea un logaritmo de n .

Sin embargo, al tratarse de una búsqueda, el bucle se ejecuta un número de veces que puede variar según la entrada del problema (nótese que en la guarda se evalúa la condición `!encontrado`). Por tanto, el coste presenta las siguientes instancias significativas:

Caso mejor: El bucle `while` termina tras ejecutar una única iteración. Esto pasa si el dato a buscar `x` coincide con el elemento que ocupa la posición central del array (`x==v[(inicio+fin)/2]` con `inicio=0` y `fin=v.length-1`).

Caso peor: El número de iteraciones del bucle `while` es el máximo ($\log_2(n) + 1$). Esto sucede cuando el dato a buscar `x` no se encuentra dentro del array.

Asumiendo que el cuerpo del bucle es una única instrucción crítica, entonces:

Caso mejor: $T^m(n) = 1 \in \Theta(1)$.

Caso peor: $T^p(n) = \log_2(n) + 1 \in \Theta(\log n)$.

A partir de las funciones $T^p(n)$ y $T^m(n)$ obtenidas, se puede decir que las cotas de la función de coste temporal del algoritmo iterativo de búsqueda binaria serán:

$$T(n) \in \Omega(1) \cap O(\log n)$$

El algoritmo - Versión recursiva

Para abordar el diseño recursivo del algoritmo de búsqueda binaria se siguen los pasos vistos en la sección 11.1.

1. Enunciado del problema. Se replantea el problema como sigue: “obtener la posición de `x` en el array `v[0..v.length-1]` sabiendo que sus elementos están ordenados ascendenteamente”. Siguiendo el esquema general recursivo de búsqueda ascendente, se puede establecer como cabecera del método la siguiente:

```
/** 0<=inicio<=v.length y -1<=fin<v.length */
static int busBinariaRec(int[] v, int inicio, int fin, int x)
```

Si se busca `x` en todo el array `v`, la llamada inicial debe ser:
`int pos = busBinariaRec(a,0,a.length-1,x);`

2. Análisis de casos.

- Caso base. Si v es un array vacío, obviamente x no está en el array. Por tanto, el caso base de `busBinariaRec` es aquel en que $\text{fin} < \text{inicio}$ y tiene como resultado -1.
- Caso general. Cuando $\text{fin} \geq \text{inicio}$, para el subarray $v[\text{inicio}..\text{fin}]$, habrá que:
 - Determinar la posición que ocupa el elemento central, $\text{mitad} = (\text{inicio}+\text{fin})/2$.
 - Acceder a dicho elemento, $v[\text{mitad}]$, comprobando que si:
 - $v[\text{mitad}] == x$, entonces la búsqueda acaba con éxito y devuelve mitad .
 - $v[\text{mitad}] > x$, entonces la búsqueda continúa en el subarray $v[\text{inicio}..\text{mitad}-1]$.
 - $v[\text{mitad}] < x$, entonces la búsqueda continúa en el subarray $v[\text{mitad}+1..\text{fin}]$.

3. Transcripción del algoritmo a un método en Java.

```
/** 0<=inicio<=v.length y -1<=fin<v.length */
static int busBinariaRec(int[] v, int inicio, int fin, int x) {
    if (inicio>fin) return -1;
    else { int mitad = (inicio+fin)/2;
        if (v[mitad]==x) return mitad;
        else if (v[mitad]>x)
            return busBinariaRec(v,inicio,mitad-1,x);
        else return busBinariaRec(v,mitad+1,fin,x);
    }
}
```

4. Validación del diseño. Al examinar el código se observa que:

- en el caso general, en cada llamada la talla del problema se divide por 2 (división entera); así, en algún momento $\text{inicio} > \text{fin}$, alcanzando el caso base y finalizando el algoritmo.
- en cualquiera de los dos casos establecidos, los valores de `inicio` y `fin` siempre cumplen la precondition ($0 \leq \text{inicio} \leq v.length$ y $-1 \leq \text{fin} < v.length$).

El coste del algoritmo - Versión recursiva

La talla del problema es el número de elementos del array en consideración, $n = \text{fin}-\text{inicio}+1$. Al tratarse de una búsqueda, igual que en la versión iterativa, se distinguen las siguientes instancias significativas:

Caso mejor: El elemento a buscar está donde se realiza la primera comparación ($v[(\text{inicio}+\text{fin})/2]==x$ con $\text{inicio}=0$ y $\text{fin}=v.\text{length}-1$).

Caso peor: El elemento no se encuentra en el array v y, por ello, se hacen el máximo número de llamadas recursivas.

Se plantea la relación de recurrencia para cada caso:

Caso mejor: En este caso, el coste es constante, $T^m(n) = 1 \in \Theta(1)$.

Caso peor: En cada llamada recursiva el tamaño del problema se divide por la mitad. El coste del algoritmo en este caso será:

$$T^p(n) = \begin{cases} T^p(n/2) + 1 & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Resolviendo esta ecuación por sustitución se tiene:

$$\begin{aligned} T^p(n) &= T^p(n/2) + 1 \\ &= T^p(n/2^2) + 2 \\ &= T^p(n/2^3) + 3 \\ &\dots \\ &= T^p(n/2^i) + i \\ &\dots \\ &= T^p(n/2^{\lfloor \log_2(n) \rfloor}) + \lfloor \log_2(n) \rfloor \\ &= T^p(0) + k = 1 + k \in \Theta(\log n) \end{aligned}$$

donde $k > \log_2(n)$, ya que el argumento de la función para el caso trivial debe ser 0, y esto se da cuando $2^k > n$ gracias a la división entera.

A partir de las funciones $T^p(n)$ y $T^m(n)$ obtenidas, se puede decir que tiene el mismo comportamiento que el iterativo y, por tanto, las cotas de la función de coste temporal del algoritmo recursivo de búsqueda binaria serán:

$$T(n) \in \Omega(1) \cap O(\log n)$$

13.6 Problemas propuestos

1. Dados los siguientes arrays:

- a) {14, 8, 16, 9, 11, 3, 14, 5}
- b) {14, 2, 17, 17, 3, 15, 9, 10}
- c) {13, 16, 10, 5, 14, 8, 15, 14}
- d) {7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3}
- e) {53, 32, 41, 11, -3, 37, 0, 15, 38, 5, 44, 10, 37, 19}

Escribir la traza para los algoritmos de:

- Selección directa.
- Inserción directa.
- Intercambio directo.
- Mergesort.

2. Escribir una versión del algoritmo de selección directa donde se busque el máximo en lugar del mínimo.
3. Una variante del algoritmo de selección consiste en ir tomando los dos menores elementos del array que queda por ordenar para situarlos en el lugar adecuado. Escribir un método con esa variante del algoritmo de selección para arrays de números reales.
4. Considérese el método de ordenación de la burbuja. ¿Cómo habría que modificar el algoritmo para ordenar sólo los m elementos más pequeños ($1 \leq m \leq n$)? ¿Cuál sería la complejidad en ese caso?
5. Dado un array v de n enteros, ordenado de forma creciente, diseñar un algoritmo iterativo de coste $O(\log n)$, para determinar si existe algún elemento que coincida con su índice; es decir, si existe algún índice i , con i entre 0 y $n-1$, tal que $v[i]=i$.
6. Modificar el algoritmo de inserción directa para que el array quede ordenado descendente.
7. ¿Cuál sería el coste del algoritmo de inserción directa para arrays con todos sus elementos iguales?
8. Modificar el algoritmo de inserción directa para que calcule la posición donde debe ir el elemento a insertar mediante una búsqueda binaria.
9. Implementar una versión del algoritmo de inserción directa para arrays de números enteros donde el subarray ordenado se sitúe *al final* del array original, de manera que el array va quedando ordenado por el final (es decir, se hace el proceso de inserción de manera descendente).

10. ¿Cuál sería el coste del algoritmo de selección directa para arrays con todos sus elementos iguales?
11. Escribir una versión del algoritmo de ordenación por intercambio para que devuelva una secuencia ordenada de mayor a menor.
12. Modificar el algoritmo de ordenación por intercambio para determinar la posición del último intercambio y así ahorrar iteraciones.
13. Escribir una versión del algoritmo de ordenación por intercambio para arrays de números enteros en la que se vayan situando los máximos en vez de los mínimos (es decir, que se hagan los recorridos de manera ascendente).
14. Alberto, en la primera clase del tema de los algoritmos de ordenación, propuso el siguiente algoritmo:

Se repite el siguiente proceso $n/2$ veces (siendo n la talla del array): buscar el elemento mínimo y el máximo entre aquellos que no se han ordenado ya (en el primer paso, se buscan entre todos los elementos del array, en la siguiente iteración se buscan entre todos menos el primero y el último...). Una vez encontrados, se intercambian con los elementos que se encuentran en las posiciones que deberían ocupar.

Implementar dicho algoritmo y calcular su complejidad temporal.

15. Un algoritmo de ordenación es *estable* si mantiene el orden original entre los elementos iguales. El algoritmo de inserción directa, tal y como se ha implementado en este capítulo, es un algoritmo de ordenación estable. Ahora imaginar que se modifica la condición del `while` del siguiente modo: `while (j>=0 && v[j]>=x)`. Con dicha modificación,
 - a) ¿el algoritmo seguiría ordenando correctamente?
 - b) ¿el algoritmo continuaría siendo estable?
 - c) El algoritmo de intercambio directo, tal y como se ha implementado en este capítulo, también es un algoritmo de ordenación estable. Ahora imaginar que se modifica la condición del `if` más interno del siguiente modo: `if (v[j-1]>=v[j])`. Con dicha modificación, indicar si el algoritmo seguiría ordenando correctamente, y en caso afirmativo, si el algoritmo seguiría siendo estable.
16. Añadir a la clase `Punto` del capítulo 5 un método para comparar puntos, de forma que el punto (x_1, y_1) es menor que el punto (x_2, y_2) si $x_1 < x_2$ o bien si ($x_1 = x_2$ y $y_1 < y_2$). Haciendo uso de dicho método, escribir un método para ordenar un array de `Punto` siguiendo cualquiera de los algoritmos presentados en este capítulo.

Más información

- [BB97] G. Brassard and P. Bratley. *Fundamentos de Algoritmia*. Prentice Hall, 1997. Capítulo 7 (7.3 y 7.4.1).
- [FAM98] F.J. Ferri, J.V. Albert, and G. Martín. *Introducció a l'anàlisi i disseny d'algorismes*. Universitat de València, 1998. Capítulo 5 (5.1, 5.2, 5.3 y 5.4.1).
- [Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulo 4 (4.3.1) y Capítulo 8 (8.1, 8.2, 8.3 y 8.5).

Capítulo 14

Extensión del comportamiento de una clase. Herencia

Como se ha mencionado en el capítulo 2, uno de los objetivos fundamentales de la POO es la de facilitar la *reutilización del código*. Ello permite volver a emplear elementos que al haber sido ya realizados son bien conocidos y están, posiblemente, exhaustivamente probados. En particular, en los lenguajes de programación orientados a objetos el mecanismo básico para el reuso del código es la *herencia*. Mediante ella es posible definir nuevas clases extendiendo o restringiendo las funcionalidades de otras clases ya existentes.

La *herencia* es un mecanismo que permite modelar relaciones jerárquicas entre elementos, del tipo *is a* (*es un(a)*), por ejemplo, esta es la relación que se da entre una máquina y un ordenador, en la que un ordenador *es una* máquina. En una relación así un elemento, el *heredero*, tiene las características de otro elemento pero, tal vez, refinándolas para definirlo como un caso especial del primero. Nótese que, para el ejemplo anterior, si un **Ordenador** *es una* **Maquina**, también un **PCCompatible** *es un* **Ordenador**, así como **miPC** es, a su vez, un **PCCompatible**. Naturalmente, desde la POO **Maquina**, **Ordenador** y **PCCompatible** son todos ellos clases, que forman una *jerarquía*, siendo una instancia de todos ellos (un objeto) **miPC**.

Desde el punto de vista del lenguaje Java hay dos puntos donde la herencia es particularmente relevante. Por una parte, la herencia se emplea exhaustivamente en el propio lenguaje a lo largo del conjunto de librerías de clases que posee. Por otra parte el lenguaje, como cabía prever, da soporte a la definición de nuevas clases *herederas* de las características de otras ya definidas. Cualquier clase, predefinida o definida por el programador, es una clase que hereda de **Object**, la clase base de la jerarquía.

14.1 Jerarquía de clases. Clases base y derivadas

La herencia es el mecanismo mediante el cual se utiliza la definición de una clase llamada *base* para definir una nueva clase llamada *derivada*, la cual hereda sus propiedades. La relación de herencia entre clases genera lo que se denomina *jerarquía*. A modo de ejemplo, se plantea definir dos clases: **Circulo** y **Rectangulo**. Supóngase que ambas clases definen un atributo para la posición y otro para el color; además, en **Circulo** se define el radio y en **Rectangulo** la base y la altura. La funcionalidad de las clases se define a través de los métodos *gets* y *sets* correspondientes y otros para desplazar la figura o para calcular su área. En la tabla 14.1 se muestra de forma resumida esta información.

Clase	Atributos y Métodos
Circulo	<pre>Point2D.Double posicion; Color color; double radio;</pre> <pre>Point2D.Double getPosition() Color getColor() double getRadio() void setPosition(double,double) void setColor(Color) void setRadio(double) void desplazar(double) double area()</pre>
Rectangulo	<pre>Point2D.Double posicion; Color color; double base, altura;</pre> <pre>Point2D.Double getPosition() Color getColor() double getBase() double getAltura() void setPosition(double,double) void setColor(Color) void setBase(double) void setAltura(double) void desplazar(double) double area()</pre>

Tabla 14.1: Las clases **Circulo** y **Rectangulo**.

Ambas clases comparten elementos comunes (los atributos *posicion*, *color* y los métodos *getPosition*, *getColor*, *setPosition*, *setColor* y *desplazar*) con los que podría definirse una nueva clase **Figura**, y tanto **Circulo** como **Rectangulo** se definirían como clases que derivan de ella. Esta jerarquía puede ampliarse. Por ejemplo, se puede definir una nueva clase **Cuadrado** como una derivada de **Rectangulo** con la base y la altura iguales. La relación que se establece entre las

clases es una relación de herencia ya que tanto **Círculo** como **Rectángulo** heredan de **Figura** parte de sus atributos y métodos. La herencia es una relación transitiva, es decir, **Cuadrado** hereda los atributos y operaciones de **Rectángulo** junto a las que éste hereda de **Figura**. La relación entre las cuatro clases se ilustra en la figura 14.1, donde las flechas indican que una clase es una derivada de una clase base: **Figura** es la clase base de la que derivan tanto **Círculo** como **Rectángulo** y **Rectángulo** es la clase base de la que deriva **Cuadrado**.

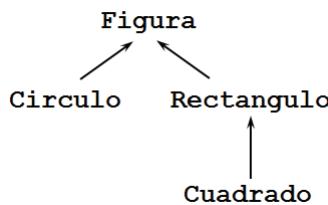


Figura 14.1: Ejemplo de jerarquía de clases.

La herencia establece una relación *es una* entre las clases, (*NomClaseDerivada es una NomClaseBase*). En la tabla 14.2 se muestra la relación de herencia y los atributos propios y heredados de cada clase.

Clase	Relación	Atributos propios y heredados
Figura	clase base	posicion , color
Círculo	<i>es una Figura</i>	posicion , color radio
Rectángulo	<i>es una Figura</i>	posicion , color base , altura
Cuadrado	<i>es un Rectángulo</i>	posicion , color base , altura

Tabla 14.2: Relación de herencia y atributos.

Como ya se ha comentado, la relación de herencia es intrínseca a la propia definición del lenguaje, dando lugar a una *jerarquía de clases*. La base de toda la jerarquía es la clase **Object**, de la que derivan todas las clases. En la figura 14.2 se ilustra parte de la jerarquía de algunas componentes para el diseño de interfaces gráficas. En esta jerarquía los objetos de tipo **JFrame** son visualizables en pantalla y pueden contener componentes gráficos distribuidos en contenedores como **JPanel**. Otros ejemplos que se verán en los siguientes capítulos son la jerarquía **Throwable** en el capítulo 15 y las jerarquías **InputStream** y **Writer** en el capítulo 16.

La jerarquía de clases es una estructura lógica basada en la relación de herencia. Existe otra jerarquía basada en la ubicación de los ficheros en los que se

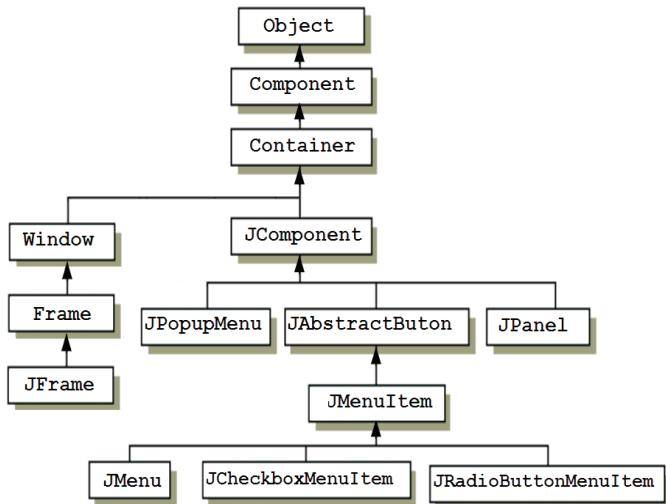


Figura 14.2: Ejemplo de jerarquía de las librerías de Java.

definen las clases. Estos ficheros se organizan en directorios dando lugar también a una estructura de árbol conocida como *jerarquía de librerías o paquetes* que se describirá con detalle en la sección 14.5. Los paquetes contienen clases y subpaquetes con funcionalidades fuertemente relacionadas entre sí. En un programa Java se accede a los paquetes separando sus nombres por puntos. Por ejemplo, `java.awt.geom.Point2D.Double` representa el camino para acceder a la clase `Point2D.Double` que se encuentra en la librería `geom` localizada en la librería `awt` del lenguaje Java. Todas las clases del paquete `java.awt.geom` están diseñadas para trabajar con figuras geométricas en el plano. Otro ejemplo son las clases del paquete `javax.swing` que definen componentes gráficos como `JFrame` y `JPanel` usadas para diseñar interfaces gráficas de usuario.

14.2 Diseño de las clases base y derivadas: `extends`, `protected` y `super`

En esta sección se abordan los mecanismos que aporta Java para definir la relación de herencia. Las clases derivadas se implementan usando la palabra reservada `extends`. La sintaxis para crear una clase derivada es:

```
class NomClaseDerivada extends NomClaseBase { ... }
```

Se dice que la clase `NomClaseDerivada` es una clase *derivada*, *descendiente* o que *es hija* o *subclase* de la clase `NomClaseBase` y que la clase `NomClaseBase` es la *clase*

base, la *clase padre* o *superclase* de la clase `NomClaseDerivada`; o simplemente que `NomClaseDerivada` *extiende* `NomClaseBase`. Desde un punto de vista conceptual, también se dice que `NomClaseDerivada` es una *especialización* de `NomClaseBase`.

En el ejemplo de las figuras, la clase `Figura` se puede definir como la Clase Tipo de Dato siguiente:

```
import java.awt.geom.*;
import java.awt.*;
public class Figura {
    // Atributos
    private Color color;
    private Point2D.Double posicion;

    // Constructor
    public Figura(Color color, double x, double y) {
        this.color = color;
        posicion = new Point2D.Double(x,y);
    }

    // Consultores
    public Color getColor() { return color; }
    public Point2D.Double getPosicion() { return posicion; }

    // Modificadores
    public void setColor(Color c) { color = c; }
    public void setPosicion(double x, double y) {
        posicion.setLocation(x,y);
    }
    public void desplazar(double d) {
        posicion.setLocation(posicion.getX()+d,posicion.getY()+d);
    }
}
```

En `Figura` se definen los atributos: `color`, de tipo predefinido `Color`, y `posicion`, la posición de la figura en el plano, de tipo también predefinido `Point2D.Double`. Estas dos clases están definidas en la librería `java.awt` y `java.awt.geom`, respectivamente. En `Color` se definen constantes públicas de clase para algunos colores estándar como `Color.red` (rojo), `Color.green` (verde), etc. La representación interna de cada color en la clase `Color` es una combinación de los tres colores básicos rojo, verde y azul (*rgb*) en cantidades de tipo `byte`. Por ejemplo: la combinación `[r=255,g=0,b=0]` representa el color rojo. La clase `Point2D.Double` define una coordenada (*x,y*) en el plano. Los métodos comunes a todas las figuras son los que se definen en la clase `Figura`, por ejemplo, los consultores `getColor` y `getPosicion`. Nótese que no existe ningún método para el cálculo del área de una figura. Esto se debe a que el área se calcula de forma distinta para cada figura y,

por tanto, se implementa con un método distinto en cada clase derivada. Usando el mecanismo de herencia, se puede extender la clase **Figura** con las peculiaridades de cada figura:

```
public class Circulo extends Figura { ... }
public class Rectangulo extends Figura { ... }
```

Modificador protected

Como se explicó en su momento, los atributos de una clase deben ser privados, es decir, visibles sólo en la clase en la que se definen, y accesibles sólo a través de métodos públicos *gets* y *sets*. Esto implementa el *principio de ocultación* que aporta la ventaja de que cualquier cambio en la estructura de datos repercute sólo en la clase en la que se realizan los cambios facilitando así el mantenimiento de la aplicación. Si se respeta esto, las clases derivadas no tienen acceso directo a los atributos privados de la clase base aunque debido a la relación existente, en ocasiones, resulte apropiado permitir el acceso. Se podría resolver este problema declarando los atributos públicos dejando la estructura de datos al descubierto; de esta forma, cualquier modificación en la estructura implica cambios en el resto del programa. No obstante, la solución adecuada es hacer que las componentes de la clase base sean visibles sólo por las clases derivadas, es decir, expandir el principio de ocultación a las clases derivadas. Para ello se usa la palabra reservada **protected**. Así, existen cuatro tipos de visibilidad para los elementos definidos en una clase: **public**, **protected**, **private** y **friendly**. Esta última es la visibilidad que se aplica cuando no se utiliza ninguna de las anteriores. La siguiente tabla muestra el alcance de la visibilidad de un elemento de una clase según el modificador usado en su definición, suponiendo que todas las clases están en el mismo paquete.

Visible en	private	protected	public	friendly
la propia clase	sí	sí	sí	sí
una clase derivada	no	sí	sí	sí
una clase no derivada	no	no	sí	sí

Creación de objetos y super

Si se aplica la visibilidad **private** en la definición de los atributos de la clase base, los constructores de las clases derivadas están obligados a invocar al constructor de la clase base. Para invocar al constructor de la clase base, se usa la palabra reservada **super** sin nombrar explícitamente la clase base. La sintaxis de esta invocación es:

```
super( arg1, arg2, ..., argn );
```

en donde **arg₁**, **arg₂**, ..., **arg_n** forman la lista de argumentos, siendo *n* el número de parámetros del constructor de la clase base (*n* ≥ 0).

El lenguaje impone una restricción: cuando se invoca al constructor de la clase base en el constructor de la clase derivada, ésta debe ser la primera instrucción del cuerpo del método. En este caso el código para definir la clase `Circulo` queda como sigue:

```
public class Circulo extends Figura {  
    private double radio;  
  
    public Circulo(double radio, Color color, double x, double y){  
        super(color,x,y);  
        this.radio = radio;  
    }  
    ...  
}
```

Conceptualmente, con esta definición se expresa que un objeto de la clase base es una *subparte* de la clase derivada y que esa *subestructura* de datos sólo es accesible con métodos *gets* y *sets*. El uso del modificador `protected` permite heredar los atributos como si los atributos de la clase base estuvieran definidos en la propia clase derivada formando parte de la misma estructura sin la necesidad de invocar a los métodos *gets* y *sets*.

En los casos en los que los atributos no son públicos, también es recomendable usar `super` desde el punto de vista del mantenimiento y de la reutilización de código. Además de invocar un código ya escrito, si se cambia la estructura de datos de la clase base, la clase derivada será más resistente a los cambios ya que estos sólo afectarán al constructor en la clase base.

En la figura 14.3 se muestra la implementación de todas las clases que conforman la jerarquía de figuras.

Sobrescritura de métodos heredados y super

Al igual que los atributos, las clases derivadas también heredan los métodos no privados de la clase base y por lo tanto también se pueden aplicar a objetos de la clase derivada. Por ejemplo, se puede obtener el color y la posición de un círculo sin haber definido los métodos `getColor` y `getPosicion` en la clase `Circulo` como ocurre en el siguiente ejemplo:

```
Circulo c = new Circulo(4.5,Color.blue,3.0,5.0);  
Color col = c.getColor();  
Point2D.Double p = c.getPosicion();
```

```

// Fichero: Figura.java
import java.awt.geom.*;
class Figura {
    protected String color;
    protected Point2D.Double posicion;
    public Figura(Color color, double x, double y) {
        this.color = color; posición = new Point2D.Double(x,y);
    }
    public Color getColor() { return color; }
    public Point2D.Double getPosition() { return posicion; }
    public void setColor(Color c) { color = c; }
    public void setPosition(double x, double y) {
        posicion.setLocation(x,y);
    }
    public void desplazar(double d) {
        posicion.setLocation(posicion.getX()+d,posicion.getY()+d);
    }
}

// Fichero: Circulo.java
import java.awt.geom.*; import java.awt.*;
class Circulo extends Figura {
    protected double radio;
    public Circulo(double radio, Color color, double x, double y) {
        super(color,x,y); this.radio = radio;
    }
    public double getRadio() { return radio; }
    public void setRadio(double r) { radio = r; }
    public double area() { return Math.PI*radio*radio; }
}

// Fichero: Rectangulo.java
import java.awt.geom.*; import java.awt.*;
class Rectangulo extends Figura {
    protected double base, altura;
    public Rectangulo(double base, double altura,
                      Color color, double x, double y) {
        super(color,x,y); this.base = base; this.altura = altura;
    }
    public double getBase() { return base; }
    public double getAltura() { return altura; }
    public void setBase(double b) { base = b; }
    public void setAltura(double a) { altura = a; }
    public double area() { return base*altura; }
}

// Fichero: Cuadrado.java
import java.awt.geom.*; import java.awt.*;
class Cuadrado extends Rectangulo {
    public Cuadrado(double lado, Color color, double x, double y) {
        super(color,x,y); this.base = lado; this.altura = lado;
    }
    public double getLado() { return base; }
    public void setLado(double l) { base = l; altura = l; }
}

```

Figura 14.3: Implementación de la jerarquía de figuras.

Existen ocasiones en las que se desea cambiar la implementación de algún método heredado para particularizar su comportamiento en la clase derivada. Cuando en la clase derivada se define un método con el mismo perfil que el de la clase base, se anula la herencia de dicho método. En este caso se dice que el método de la clase derivada *sobrescribe* al de la clase base. Si se desea invocar el método sobrescrito de la clase base, se usa la palabra `super`. A continuación, se presenta un ejemplo de sobrescritura en el que se utiliza `super` para llamar al método homónimo de la clase base.

Ejemplo 14.1. La clase `Corona` representa una corona circular formada por dos círculos concéntricos. `Corona` puede definirse como una derivada de la clase `Circulo` con un atributo `radioIn` para el radio del círculo interior ya que el centro y el color son los mismos que los del círculo exterior. En esta clase el constructor crea un círculo externo invocando al constructor de la clase padre con el mayor de los radios e inicializa el radio interno con el menor. El área de una corona se obtiene restando el área del círculo interno a la del círculo externo. Para obtener el área del círculo externo se invoca al método `area` de la clase base, y la del círculo interno se obtiene creando un nuevo círculo de radio `radioIn` y aplicando el método `area` de la clase `Circulo`. El atributo `posicion` es del tipo predefinido `Point2D.Double` que proporciona los métodos `getX` y `getY` para obtener el valor en la abscisa y en la ordenada, respectivamente.

```
import java.awt.*;
public class Corona extends Circulo {
    private double radioIn;

    public Corona(double radio1, double radio2,
                  Color color, double x, double y) {
        super(Math.max(radio1,radio2),color,x,y);
        this.radioIn = Math.min(radio1,radio2);
    }

    public double area() {
        Circulo c = new Circulo(radioIn,color,posicion.getX(),
                               posicion.getY());
        double areaIn = c.area();
        return super.area() - areaIn;
    }
}
```

La sobrescritura de métodos es muy habitual en Java. Como ya se vió en la sección 5.5.2, los métodos `toString()` y `equals(Object)`, definidos en la clase `Object`, se sobrescriben para adecuarlos a cada clase. En la sección 14.3.2 se muestra un ejemplo en el que se realiza la sobrescritura del método `toString()`. A continuación, se presenta un ejemplo de sobrescritura del método `paintComponent(Graphics)` definido en la clase `JComponent`.

Ejemplo 14.2. El método `paintComponent(Graphics)` se invoca automáticamente cada vez que se requiere dibujar una componente gráfica en pantalla usando un objeto de tipo `Graphics`, el cual contiene toda la información necesaria para ello. Esto ocurre cuando una componente gráfica aparece en primer plano de la pantalla o cambia de tamaño o forma. Este método se define en la clase `JComponent` y lo heredan todas sus derivadas, entre ellas la clase `JPanel` base de la clase `MiPanel` de la figura 14.4. Normalmente, este método dibuja la componente con un fondo en blanco. La nueva funcionalidad del método en esta última clase consiste en añadir al `JPanel` una imagen de fondo guardada en el fichero *fondo.jpg* y escalarla cuando el método se llame para redibujar.

La primera instrucción invoca a `paintComponent(Graphics)` de la clase base para dibujar la componente sin imagen de fondo. Después se obtienen las dimensiones de `MiPanel` y se escala la imagen a las nuevas dimensiones de la ventana. La clase `ImageIcon` se usa para crear un ícono con la imagen guardada en el fichero. El método `getImage()` devuelve la imagen del ícono de la que se obtiene la nueva imagen escalada pasando las nuevas dimensiones a los dos primeros parámetros del método `getScaledInstance(int,int,int)` de la clase `Image` del paquete `java.awt`. La nueva imagen se inserta en un ícono para dibujarla.

El método `main(String[])` de la clase `Ventana` crea una ventana (`JFrame`) a la que se le asigna unas dimensiones iniciales y, tras añadirle el objeto de la clase `MiPanel`, la dibuja en pantalla invocando al método `setVisible(boolean)`. En la figura 14.5 se aprecia el efecto del escalado del fondo al cambiar la forma de la ventana.

14.3 Uso de una jerarquía de clases. Polimorfismo

En la sección 3.6.3 se estudió el concepto de compatibilidad de tipos para los tipos básicos y los tipos referencia. En esta sección se extiende este concepto para el caso de los tipos referencia que forman parte de una jerarquía. En presencia de herencia, se dice que dos tipos son compatibles si pertenecen a la misma línea de la jerarquía en sentido descendente. La compatibilidad garantiza que toda variable de tipo referencia C pueda referenciar a un objeto de la clase C y sus derivadas, pero no a la inversa. De esta forma, la herencia proporciona un *polimorfismo de variables* al poder referenciar a una variedad de tipos de objetos. En Java el siguiente código es totalmente legal, siendo las variables `f1` y `f2` capaces de referenciar tanto a un `Circulo` como a un `Rectangulo`:

```
Figura f1 = new Circulo(4.5,Color.blue,3.0,5.0);
Figura f2 = new Rectangulo(4.5,3.2,Color.red,3.0,4.0);
```

```

// Fichero: MiPanel.java
import javax.swing.*;
import java.awt.*;
public class MiPanel extends JPanel {
    ImageIcon fondo;

    public MiPanel(String nomFich) {
        fondo = new ImageIcon(nomFich);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int alturaJPanel = this.getHeight(),
        anchoJPanel = this.getWidth();
        Image imagXEscalar = fondo.getImage();
        Image imagEscalada = imagXEscalar.getScaledInstance(anchoJPanel,
            alturaJPanel, Image.SCALE_FAST);
        ImageIcon nuevoFondo = new ImageIcon(imagEscalada);
        nuevoFondo.paintIcon(this,g,0,0);
    }
}

// Fichero: Ventana.java
import javax.swing.*;
public class Ventana {
    public static void main(String[] args) {
        JFrame v = new JFrame();
        v.add(new MiPanel("fondo.jpg"));
        v.setSize(100,100);
        v.setVisible(true);
    }
}

```

Figura 14.4: Sobrescritura del método `paintComponent`.

La herencia también proporciona un *polimorfismo de métodos* pudiendo aplicarse los métodos heredados de la clase base a objetos instancia de todas sus clases derivadas. Como ocurre en el siguiente código:

```

Color cf1 = f1.getColor();
Color cf2 = f2.getColor();

```

donde el mismo método `getColor`, heredado de la clase **Figura**, se aplica a dos objetos de distinto tipo.

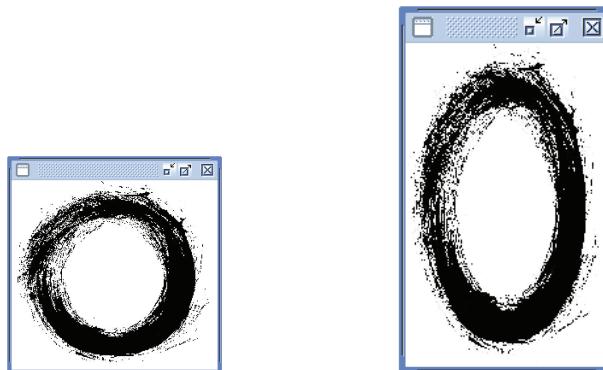


Figura 14.5: Ejemplo de escalado.

14.3.1 Tipos estáticos y dinámicos

Cuando en Java se define una variable referencia, se le asocia el tipo de los objetos a los que puede referenciar. Este tipo es conocido en tiempo de compilación y se conoce como *tipo estático*. En tiempo de compilación se comprueba si se producen violaciones de tipos estáticos. Los lenguajes con tipos estáticos pueden manejar tipos explícitos o tipos inferidos. El primer caso consiste en declarar el tipo de la variable sin importar que sea un atributo o una variable de clase. Por ejemplo:

```
// definición del tipo estático de un atributo
Figura a;
// definición del tipo estático de una variable de clase
static Figura b;
```

En el segundo caso, el compilador infiere los tipos de las expresiones y las declaraciones de acuerdo al contexto. Por ejemplo, ante la expresión

```
Double d = new Double(2.0) + 3;
```

el compilador deduce que el tipo del número 3 de tipo `int` debe ser `Double` y le aplica una conversión implícita de tipo.

Además de un tipo estático, las variables también poseen un tipo *dinámico* que se obtiene durante la ejecución. El tipo dinámico de una variable es el tipo del objeto al que referencia la variable. Por ejemplo, en la expresión

```
Object o = new Double(3.5);
```

el tipo estático de la variable `o` es `Object` y el tipo dinámico es `Double`.

Durante la ejecución, el tipo dinámico se maneja mediante un *enlace dinámico*. Cuando se aplica un método a un objeto, se determina la clase a la que pertenece el objeto, y se busca el método correspondiente en la clase. En el caso de no encontrarse, se busca en la clase base ascendiendo en la jerarquía. Este proceso se repite hasta encontrar la clase donde se define el método que se desea ejecutar. Si se llega al final de la jerarquía sin encontrarlo, se produce un error en la ejecución. Por ejemplo, considerando la jerarquía:

```
java.lang.Object
|
+--Figura
|
+--Circulo
```

y que la variable `f1` referencia a un círculo, la instrucción `f1.getColor()` ejecuta el método definido en la clase `Figura` ya que no está definido en la clase `Circulo` pero sí en la clase `Figura`. La instrucción `f1.toString()` ejecuta el método definido en la clase `Object` ya que no está definido en la clase `Circulo` ni en `Figura`.

El enlace dinámico permite al programador usar la sobreescritura como un regulador de la herencia: sabiendo que el enlace dinámico se aplicará igualmente, el programador decide sobre escribir o no un método heredado y, por tanto, modificar o mantener invariante un comportamiento definido en la clase base.

14.3.2 Ejemplo de uso del polimorfismo

En las secciones anteriores se ha visto que la herencia aporta polimorfismo a las variables y métodos dinámicos, y que junto al principio de ocultación, permite un mantenimiento más simple del software. En esta sección se plantea un ejemplo de su uso. Considerando la jerarquía de las figuras, se plantea el diseño de una clase `GrupoFiguras` para guardar varios tipos de figuras y poder gestionarlo añadiendo o eliminando una figura, calculando el área de todas ellas, mostrando su información en forma de cadena de caracteres, etc. El principal objetivo consiste en diseñar la clase `GrupoFiguras` de forma que el código quede inalterado ante una posible modificación del conjunto de tipos de figuras. Centrando el ejemplo en el diseño de un método para mostrar su información en forma de cadena de caracteres, se debe sobre escribir el método `toString()`, definido en la clase `Object`, en todas las clases de la jerarquía de `Figura`.

El método `toString()` en la clase `Object`

En la clase `Object`, este método devuelve la información básica de un objeto sobre una cadena de caracteres de la forma `NombreClase@direcciónMemoria`, donde

`NombreClase` es el tipo dinámico y `direcciónMemoria` es una representación en hexadecimal de la primera posición de memoria que ocupa el objeto en el montículo (heap). Por ejemplo, si no se sobrescribe el método `toString()` de la clase `Figura`, las siguientes instrucciones:

```
Figura f = new Figura();
System.out.println(f);
```

muestran por la salida estándar:

```
Figura@50618d26
```

Lo mismo ocurre al aplicar el método `toString()` a los objetos de las clases derivadas de `Figura` ya que, al no sobrescribir el método heredado, ejecutan el método de la clase `Object`.

El método `toString()` en la jerarquía `Figura`

La figura 14.6 muestra la sobreescritura del método `toString()` de `Object` en cada una de las clases de la jerarquía de figuras.

Cabe señalar que al concatenar una cadena con un objeto, se invoca automáticamente el método `toString()` de la clase del objeto sin necesidad de explicitarlo. Es decir, las expresiones `" " + posicion` y `" " + color` son equivalentes a escribir `" " + posicion.toString() y " " + color.toString()`, donde el método `toString()` es el definido en las clases `Point2D.Double` y `java.awt.Color`.

Los métodos de las clases `Circulo` y `Rectangulo` invocan al método de su clase base (`Figura`) usando `super`. Nótese que el método `toString()` de la clase `Cuadrado` no invoca al de su clase base para evitar que aparezca la palabra *Rectángulo*. A continuación se muestra el efecto de la sobreescritura en las clases `Cuadrado` y `Circulo`.

```
Figura f1 = new Circulo(4.5,Color.blue,3.0,5.0);
Figura f2 = new Cuadrado(2.0,Color.red,3.0,3.0);
System.out.println(f1 + "\n" + f2);
```

El resultado de la ejecución de las instrucciones anteriores en la salida estándar será el que sigue:

Entrada/Salida Estándar	
Círculo de radio 4.50, color: java.awt.Color[r=0,g=0,b=255] y posición: Point2D.Double[3.0, 5.0]	
Cuadrado de lado 2.00, color: java.awt.Color[r=255,g=0,b=0] y posición: Point2D.Double[3.0, 3.0]	

En la clase Figura:

```
public String toString() {
    return ", color: " + color + " y\n\tposición: " + posicion;
}
```

En la clase Circulo:

```
public String toString() {
    String res = "Círculo de radio ";
    String formato = String.format("%.2f", radio);
    res += formato + super.toString();
    return res;
}
```

En la clase Rectangulo:

```
public String toString() {
    String res = "Rectángulo de ";
    String formato = String.format("base %.2f, altura %.2f",
                                    base, altura);
    res += formato + super.toString();
    return res;
}
```

En la clase Cuadrado:

```
public String toString() {
    String res = "Cuadrado de lado ";
    String formato = String.format(new Locale("US"), "%.2f", base);
    res += formato + ", color: " + color;
    res += " y\n\tposición: " + posicion;
    return res;
}
```

Figura 14.6: Sobrescritura del método `toString()`.

El método `toString()` en la clase `GrupoFiguras`

A continuación se abordan tres aproximaciones para implementar la clase `GrupoFiguras` y su método `toString()`: usando varios arrays, usando el tipo `Object` y usando el tipo `Figura`. En cada una de ellas se avanza hacia un código más estable ante modificaciones del conjunto de figuras.

Primera aproximación: sin usar herencia

Una aproximación simple sin usar herencia consiste en definir un array de tamaño MAX por cada tipo de figura, una operación para insertar cada tipo de figura en el array correspondiente y, finalmente, sobrescribir el método `toString()` implementando un recorrido sobre cada uno de los arrays.

```
class GrupoFiguras {  
    private final int MAX = ...;  
    private Circulo[] circulos;  
    private Rectangulo[] rectangulos;  
    private Cuadrado[] cuadrados;  
    private int numCirculos, numRectangulos, numCuadrados;  
  
    public GrupoFiguras() {  
        circulos = new Circulo[MAX];  
        rectangulos = new Rectangulo[MAX];  
        cuadrados = new Cuadrado[MAX];  
        numCirculos= 0; numRectangulos = 0; numCuadrados = 0;  
    }  
  
    public void insertar(Circulo c) {  
        if (numCirculos<MAX) circulos[numCirculos++] = c;  
    }  
    public void insertar(Rectangulo r) {  
        if(numRectangulos<MAX) rectangulos[numRectangulos++] = r;  
    }  
    public void insertar(Cuadrado c) {  
        if (numCirculos<MAX) cuadrados[numCuadrados++] = c;  
    }  
  
    public String toString() {  
        String res = "";  
        for (int i=0; i<numCirculos; i++) res += circulos[i] + "\n";  
        for (int i=0; i<numRectangulos; i++) res += rectangulos[i]+"\n";  
        for (int i=0; i<numCuadrados; i++) res += cuadrados[i] + "\n";  
        return res;  
    }  
}
```

Con esta aproximación, por cada nuevo tipo de figura se debe añadir un nuevo array, un nuevo contador, un nuevo método `insertar` y un nuevo bucle en el método `toString()`. Nótese que no se mantiene información sobre el orden de inserción.

Segunda aproximación: usando el tipo Object

Una primera forma de evitar los cambios anteriores, consiste en usar el tipo polimórfico `Object` y definir un único array y un único método `insertar(Object)`.

```
class GrupoFiguras {
    private final int MAX = ...;
    private Object[] objetos;
    private int numObjetos;

    public GrupoFiguras() {
        objetos = new Object[MAX];
        numObjetos = 0;
    }

    public void insertar(Object o) {
        if (numObjetos<MAX) objetos[numObjetos++] = o;
    }

    public String toString() {
        String res = "";
        for (int i=0; i<numObjetos; i++) res += objetos[i] + "\n";
        return res;
    }
}
```

Con esta aproximación, se obtiene un código más conciso y menos sensible a cambios en la aplicación. Esta solución parece suficiente, sin embargo, no carece de problemas. Se presupone que el parámetro del método `insertar(Object)` sólo recibe referencias a los tipos descendientes de `Figura`. En realidad, el tipo `Object` permite que el método reciba referencias a cualquier objeto. Una forma de controlarlo consiste en verificar que el tipo del parámetro `o` es una de las figuras. Esto se comprueba utilizando la instrucción `instanceof` y en el caso de que no se trate de una de las figuras, no se produce inserción alguna.

```
public void insertar(Object o) {
    if (o instanceof Circulo ||
        o instanceof Rectangulo || o instanceof Cuadrado)
        if (numObjetos<MAX) listaObjetos[numObjetos++] = o;
}
```

Esta implementación también carece de independencia, ya que cada nuevo tipo de figura requiere cambios en la condición del primer `if`.

Tercera aproximación: usando herencia

Haciendo uso de la herencia, se restringe el tipo de la lista y el parámetro del método `insertar(Object)` al tipo `Figura`. De esta forma, el parámetro de `insertar(Figura)` puede referenciar a la variedad de tipos de objetos descendientes de `Figura`, entre los que se encuentran `Círculo`, `Rectángulo` y `Cuadrado`.

```
public class GrupoFiguras {  
    private final int MAX = ...;  
    private Figura[] figuras;  
    private int numFiguras;  
  
    public GrupoFiguras() {  
        figuras = new Figura[MAX];  
        numFiguras = 0;  
    }  
  
    public void insertar(Figura f) {  
        if (numFiguras<MAX) figuras[numFiguras++] = f;  
    }  
  
    public String toString() {  
        String res = "";  
        for (int i=0; i<numFiguras; i++) res += figuras[i] + "\n";  
        return res;  
    }  
}
```

Con esta aproximación se aprovechan las ventajas que comporta el uso de la herencia, las variables polimórficas y la sobrescritura para que el código sea independiente del conjunto de tipos de figuras. Cuando se desee trabajar con nuevos tipos de figura, bastará con definir una nueva clase descendiente de la clase `Figura`.

No obstante, esta solución tampoco es completamente satisfactoria ya que permite almacenar objetos de tipo `Figura`:

```
GrupoFigura g = new GrupoFigura();  
g.insertar(new Figura());
```

El problema reside en que la clase `Figura` se define como una clase auxiliar para definir las propiedades comunes a todas las figuras, pero no se pretende que exista un objeto de esta clase.

Otra carencia de las dos últimas aproximaciones, es la imposibilidad de realizar el enlace dinámico ya que todos los objetos del array son de tipo `Object` o de tipo `Figura`. Por ejemplo, si además de implementar el método `toString()`, se intenta

calcular el área de todas las figuras, se requiere el uso de casting para localizar el método que calcula el área de cada tipo de figura y localizar los métodos `area()` de cada clase:

```
public double area() {
    double areaTotal = 0;
    for (int i=0; i<numObjetos; i++) {
        if (objetos[i] instanceof Circulo)
            areaTotal += ((Circulo)objetos[i]).area();
        if (objetos[i] instanceof Rectangulo)
            areaTotal += ((Rectangulo)objetos[i]).area();
        if (objetos[i] instanceof Cuadrado)
            areaTotal += ((Cuadrado)objetos[i]).area();
    }
    return areaTotal;
}
```

Como se verá más adelante, para evitar el uso de castings y la instrucción `instanceof`, se pueden definir clases en las que no se permite crear instancias y en las que se pueden definir métodos cuya implementación se aplaza a sus clases derivadas.

14.4 Más herencia en Java: control de la sobrescritura

Como ya se ha señalado en las secciones precedentes, en el diseño de una jerarquía de clases se puede emplear la *sobrescritura* para que las clases derivadas especialicen su comportamiento en lugar de definir métodos nuevos. De este modo se pueden aprovechar al máximo todas las ventajas que el uso de las variables polimórficas comporta. Ahora bien, con los mecanismos estudiados hasta el momento para implementar la herencia, el diseñador no posee más que un control parcial de la sobrescritura, pues ni puede impedir que se produzca ni puede forzar u obligar a realizarla. Para conseguir prohibir o imponer la sobrescritura, y así convertir a la herencia en un verdadero instrumento de reutilización del software, se requieren los modificadores Java `final` y `abstract`, respectivamente; a su presentación y estudio están dedicadas las siguientes subsecciones.

14.4.1 Métodos y clases finales

Como es sabido, una clase **Derivada** puede especializar a su clase **Base** cambiando el significado de los métodos heredados mediante sobrescritura. Ahora bien, si se desea que un método de la superclase, `f`, permanezca *invariante* en la jerarquía y, por tanto, prohibir su sobrescritura, dicho método se debe definir como `final` añadiendo a su cabecera de definición el modificador `final`. Así, cualquier intento

de sobrescribir este método en una derivada provocará un error de compilación `f in Derivada cannot override f in Base; overridden method is final.`

También un atributo y una clase Java pueden ser finales; por ejemplo,

- el atributo PI de `Math` es *final*, pues representa a la constante π ;
- todas las clases envoltorio de los tipos primitivos de Java (`Integer`, `Boolean`, `Character`, `Double`, etc.) y la clase `String`, ubicadas en el paquete `java.lang` son *finales*. Obsérvese en la documentación de cualquiera de estas clases que una clase final se ubica como una hoja del árbol de herencia Java para expresar que *no puede ser extendida* o reutilizada vía herencia y que, ya a nivel puramente sintáctico, basta poner el modificador `final` en su cabecera de definición para que automáticamente todas sus componentes sean definidas como finales.

El uso de componentes `final` no sólo evita su redefinición accidental sino que también permite generar código más eficiente puesto que el intérprete Java resuelve su función asignada en tiempo de compilación (estáticamente), al igual que con una componente `static`, y no en tiempo de ejecución (dinámicamente).

14.4.2 Métodos y clases abstractos

En una jerarquía de clases algunos métodos pueden cambiar su significado mediante sobrescritura mientras que otros, los finales, permanecen invariantes. Una situación intermedia es aquella en la que se desea compaginar la herencia forzosa de `final` con la especialización de la sobrescritura. Por ejemplo, parece razonable que el método `área()` forme parte de la especificación de la clase `Figura` y, por lo tanto, sea heredado por todas sus subclases; pero sólo existen fórmulas de cálculo del área específicas para cada tipo de `Figura`, por lo que el área de una `Figura` sólo se puede calcular en las derivadas `Círculo`, `Rectángulo`, etc.

Para imponer que todas las subclases de una superclase dada hereden un determinado método pero, al mismo tiempo, que cada subclase lo especialice, dicho método se debe definir como *abstracto*. Para ello,

1. En la implementación de la superclase se define un método cuya cabecera contiene el modificador `abstract` y sin cuerpo, es decir, donde el bloque determinado por `{}` se substituye por `;`. Por ejemplo, la definición de `área` en `Figura` sería:

```
public abstract double area();
```

2. El método definido en la superclase debe ser implementado en todas sus subclases, esto es, definido en cada subclase con la misma cabecera de la superclase pero con un cuerpo ad-hoc, en lugar del `;`

Una clase que tiene al menos un método abstracto es una *clase abstracta* y así se debe declarar; por ejemplo, **Figura** es abstracta, por lo que se define:

```
public abstract class Figura { ... }
```

Una clase abstracta no puede ser instanciada, esto significa que no se pueden crear objetos de la clase vía operador **new**, aunque sí se pueden declarar variables referencia. Por ejemplo, si la clase **Figura** es abstracta, la creación de un objeto vía el operador **new** (**Figura fEstandar = new Figura();**), provoca el error de compilación **Figura is abstract - cannot be instantiated**; sin embargo, no produce error la creación de un objeto de tipo **Circulo** a través de una referencia de tipo **Figura** (**Figura c = new Circulo();**).

Una subclase debe implementar todos los métodos abstractos de la superclase, al menos que ésta a su vez sea una clase abstracta; si no lo hace, el compilador lo detectará y dará un error. Por ejemplo, si **Circulo** no implementa **area**, se produce el siguiente mensaje de error **Circulo is not abstract and does not override abstract method area() in Figura**.

La clase **Figura** modificada como abstracta quedaría como sigue:

```
import java.awt.geom.*;
public abstract class Figura {
    protected String color;
    protected Point2D.Double posicion;

    public Figura(Color color, double x, double y) {
        this.color = color; posición = new Point2D.Double(x,y);
    }

    public String getColor() { return color; }
    public Point2D.Double getPosicion() { return posicion; }
    public abstract double area();
    public String toString() {
        return ", color " + color + " y posición: " + posicion;
    }
}
```

Así se pueden insertar varios tipos de figura en un grupo con la seguridad de que todas implementan su método **area** y que en el grupo de figuras sólo se insertan objetos de las clases derivadas de **Figura**. El cálculo del área total de todas las figuras se implementa con el método **area()** de la clase **GrupoFiguras**.

```
public double area() {
    double areaTotal = 0.0;
    for (int i=0; i<numFiguras; i++)
        // se invoca al método de cada figura
        areaTotal += figuras[i].area();
    return areaTotal;
}
```

La ejecución de la siguiente secuencia de instrucciones nos daría el área total del grupo de figuras.

```
GrupoFiguras g = new GrupoFiguras();
g.insertar(new Círculo(4.5,Color.blue,3.0,5.0));
g.insertar(new Rectángulo(4,6,Color.blue,3.0,4.0));
g.insertar(new Cuadrado(4.5,Color.blue,3.0,5.0));
// se invoca al método area de GrupoFiguras
System.out.println(g.area());
```

14.4.3 Interfaces y herencia múltiple

Una clase Java cuyos métodos son todos abstractos recibe el nombre específico de *interfaz* y se utiliza para describir un comportamiento o funcionalidad específica pero sin implementación. Una interfaz se caracteriza porque sus atributos sólo pueden ser públicos y finales, no posee métodos constructores al carecer de estructura y, finalmente, sus métodos son públicos y han de ser implementados obligatoriamente en cualquiera de sus derivadas -por lo que se dice que éstas implementan la interfaz en lugar de extenderla.

Los pasos para definir una clase interfaz I en Java, y para indicar que otra clase D la implementa, son los siguientes:

- En la cabecera de I se utiliza la palabra `interface` en lugar de `class`.
- En la cabecera de cada método de I no se deben escribir los modificadores `public` y `abstract` pues lo son por definición de interfaz.
- En la cabecera de la clase D que implementa I debe incluirse `implements I`:

```
public class D implements I { ... }
```

En el caso en el que la clase D extienda a B:

```
public class D extends B implements I { ... }
```

Si D implementa varias interfaces, sus nombres se separan por comas.

Es muy importante señalar ahora que, gracias a que una clase puede implementar tantas interfaces como sea necesario, la interfaz es el mecanismo con el que Java implementa la *herencia múltiple*, esto es, el hecho de que una clase derivada herede de más de una clase base distinta de `Object`.

- En el cuerpo de la clase D que implementa I se deben sobrescribir obligatoriamente todos los métodos definidos en I, salvo si D es una clase abstracta o interfaz, pues de lo contrario el intérprete Java lo advierte.

Algunas interfaces predefinidas en Java son `Cloneable`, `Comparable`, `Runnable` y `Serializable`.

14.5 Organización de las clases en Java

14.5.1 La librería de clases del Java

Un aspecto importante del lenguaje Java es la *librería de clases predefinidas*, ya que incluye un grupo muy amplio de tipos y operaciones relacionados con problemas comunes de la programación actual. Así, la librería de clases del Java permite tratar aspectos tan distintos como el tratamiento numérico, la gestión fiable de comunicaciones remotas y la realización de interfaces de usuario gráficas, entre otros muchos.

La librería de clases del lenguaje se encuentra organizada de forma jerárquica, teniendo como clase base la denominada `Object`. Así, todas las clases del Java son, de una forma u otra, descendientes de la clase `Object` y, por ello, heredan, a veces sobrescribiéndolos, los métodos de dicha superclase.

La herencia entre clases predefinidas es muy habitual en Java, pudiéndose representar la jerarquía de clases del lenguaje mediante un árbol de bastante profundidad. Véase, por ejemplo, la jerarquía correspondiente a la clase `JFrame`, tal y como aparece en la documentación del lenguaje:

```
java.lang.Object
|
+--java.awt.Component
|
+--java.awt.Container
|   |
|   +--javax.swing.JComponent
|       |
|       +-- javax.swing.JList
+--java.awt.Window
|
+--java.awt.Frame
|
+--javax.swing.JFrame
```

Cada una de las clases del ejemplo: `Object`, `Component`, `Container`, `Window`, `Frame` y `JFrame`, heredan en sus definiciones los atributos y métodos de las clases precedentes, sobrescribiéndolos cuando así lo necesitan, de forma que las funcionalidades de una clase quedan definidas por las de las clases que extienden, junto con las aportadas por ella misma.

Nótese que los nombres de las clases del ejemplo anterior vienen antepuestos por las etiquetas `java.lang`, `java.awt` o `javax.swing`. Dichas etiquetas no son nombres

de clases sino que son un mecanismo de agregación de clases, propio del Java, denominado *paquete* (*package*).

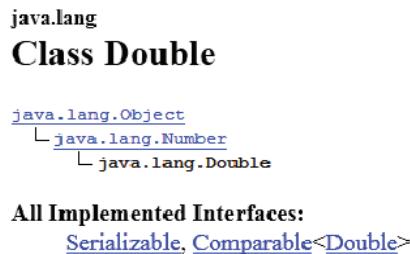


Figura 14.7: Documentación de la clase Double.

Toda la información referente a la jerarquía en la que se define una clase se puede consultar en la documentación que el propio lenguaje genera. En la figura 14.7 se muestra la referida a la clase **Double**; en la primera línea se nombra el paquete donde está definida la clase seguido del nombre de la clase. A continuación se dibuja el segmento de la jerarquía de clases que lleva desde **Object** hasta la clase **Double**, y en la última línea aparecen los interfaces que implementa. En concreto, la clase **Double** hereda de la clase **Number** y de la interfaz **Serializable** entre otros.

Seguido de esta información se muestran las siguientes tablas, que pueden aparecer o no en función de que la clase tenga o no elementos visibles desde fuera de la clase:

Field Summary con información sobre los atributos públicos y protegidos (**protected**) ya sean variables de instancia o de clase (**static**).

Fields inherited from class ... con los atributos heredados de otras clases; se muestra una tabla por cada clase de la que se heredan atributos.

Constructor Summary con los constructores de la clase.

Method Summary con una breve descripción de los métodos accesibles.

Methods inherited from class ... con métodos heredados de otras clases; se muestra una tabla por cada clase de la que se heredan métodos.

Methods inherited from class java.lang.Object se muestran los métodos heredados de la clase **Object**.

14.5.2 Uso de packages

En Java los paquetes (**packages**) se utilizan para poder agrupar un conjunto de clases que tienen funcionalidades comunes. Un **package** es, por lo tanto, un mecanismo del lenguaje para facilitar la organización de un conjunto de clases. Existe un grupo de paquetes predefinidos en los que se encuentran englobadas todas las clases de la librería del lenguaje. Ejemplos de paquetes son los ya mencionados en el ejemplo anterior: **java.lang** y **java.awt**.

Cuando en un fichero con instrucciones Java se desea hacer uso de una clase de un paquete, es posible hacerlo usando el nombre completo prefijado de la clase, esto es el nombre del paquete seguido del de la clase, por ejemplo se puede hacer referencia a la clase **Frame** siempre que se necesite mediante **java.awt.Frame**. Sin embargo, suele ser más simple utilizar una directiva de importación de los elementos que interesen, como por ejemplo:

```
import java.awt.*;
// Se importan todas las clases del paquete java.awt

import java.awt.Frame;
// Se importa sólo la clase java.awt.Frame
```

De entre los paquetes y clases predefinidos del lenguaje Java cabe destacar por su importancia los siguientes:

- **java.applet**: definición y gestión de las *applets*, pequeñas aplicaciones que se descargan usando la WWW y se ejecutan, de forma segura, remotamente.
- **java.awt**: es el “Abstract Window Toolkit” o herramientas de ventana abstracta. Se trata de un conjunto de clases para la realización de interfaces de usuario gráficos, independientes de la plataforma específica en que se utilice.
- **java.io**: paquete que contiene las clases relativas a la entrada/salida en Java. Modela aspectos tan importantes como los flujos (**Stream**) y los ficheros (**File**).
- **java.lang**: es el paquete que contiene la mayoría de las clases de uso más común, como por ejemplo:
 - **Math**: funciones matemáticas como logarítmicas y trigonométricas.
 - **Object**: la clase raíz en la jerarquía de clases.
 - **String**: tratamiento de cadenas de caracteres.
 - **System**: aspectos básicos de gestión del sistema.
 - **Thread**: tratamiento para concurrencia y ejecución en paralelo.

- **java.util**: incluye un grupo de clases predefinidas, muchas de ellas tipos de datos de uso común, como por ejemplo: Pilas, Colas, Colecciones, Tablas Hash, Diccionarios, etc. La clase **Date**, para manejo de fechas y horas, también se encuentra entre ellas.
- **java.net**: contiene clases relativas a aplicaciones que acceden a redes.

El lenguaje permite construir nuevos paquetes agrupando distintas clases añadiendo la cláusula `package nombrePaquete` al principio del fichero que contiene la clase. Éstas podrán ser utilizadas posteriormente haciendo uso de los mecanismos de referenciación ya vistos. La organización de las clases en paquetes tiene muchas ventajas: permite organizar el código desarrollado, facilita la posibilidad de compartir código con otros programadores al eliminar la ambigüedad en los nombres de las clases y permite el uso del denominado *modificador de visibilidad de paquete o de acceso amistoso*, que es el que sintácticamente se aplica en ausencia de modificador de visibilidad (`public`, `private` o `protected`) y que permite el acceso desde clases que formen parte del mismo paquete. Por eso en Java todas las clases se ubican en paquetes, bien en el que se indica explícitamente, bien en un paquete `anonymous` en el que se agrupan todas las clases que no están definidas en un paquete determinado.

Siguiendo las recomendaciones del propio lenguaje sólo se utilizarán clases no ubicadas en paquetes cuando se desarrolle una pequeña aplicación o aplicación de prueba o bien en los comienzos del desarrollo de programas.

14.6 Problemas propuestos

1. Tomando como modelo las clases `Circulo` y `Rectangulo` vistas en el capítulo, implementar una clase `Triangulo` que descienda de la clase `Figura`.
2. Añadir un método `perímetro` en la clase `Figura` de forma que todas las figuras tengan que implementarlo.
3. Especializar el método `perímetro` del ejercicio anterior para todas las clases que desciendan de la clase `Figura`.
4. En la clase `GrupoFiguras`:
 - Diseñar un método que calcule la suma de los perímetros de todas las figuras.
 - Diseñar un método que cree y devuelva un cuadrado con el área total de todas las figuras del grupo.
 - Diseñar un método que ordene las figuras en orden ascendente de su área.
5. Diseñar una clase `Polígono` similar a las clases `Circulo` y `Rectangulo` que guarde un número máximo de puntos dados en su constructor.
6. Rediseñar las clases `Circulo`, `Rectangulo` y `Triangulo` extendiendo la clase `Polígono`.
7. Si no se pueden crear objetos de una clase abstracta vía operador `new`, ¿qué utilidad tienen los métodos constructores de las clases abstractas? ¿Qué consecuencias tiene la siguiente definición en la clase `Figura`?

```
private Color color;
private Point2D.Double posicion;
Figura(Color color, int x, int y) {
    this.color = color;
    posicion = new Point2D.Double(x,y);
}
```

8. Dada la siguiente jerarquía de clases en el paquete `vehiculos`:

```
public class Vehiculo {
    ...
    public Vehiculo(int potencia) { ... }
    public int potencia () { ... }
    ...
}
```

```
public class Coche extends Vehiculo {  
    ...  
    public Coche(int potencia, int numPlazas) { ... }  
    public int numPlazas() { ... }  
    ...  
}  
  
public class Moto extends Vehiculo {  
    ...  
    public Moto(int potencia) { ... }  
    ...  
}
```

Diseñar en este mismo paquete una clase **Garaje** de forma que:

- En el constructor se indique el número total de plazas del **Garaje**.
 - En cada plaza se pueda guardar tanto un **Coche** como una **Moto**.
 - Tenga una función que devuelva la cuota mensual de una plaza calculada de la forma siguiente:
 - si en dicha plaza hay un **Coche**, la cuota es la potencia multiplicada por el número de plazas;
 - si en la plaza hay una **Moto**, la cuota se calcula como la potencia multiplicada por 2;
 - si no hay ningún vehículo en la plaza, la cuota es 0.
9. Se dispone de las siguientes clases en el paquete **losAnimales**:

```
public class Milpies {  
    protected int numeroDePies;  
    public Milpies() {  
        numeroDePies = 1000;  
        escribirPies();  
    }  
  
    public void escribirPies() {  
        System.out.println("Un Milpiés o Cochinilla tiene " +  
                           numeroDePies + " pies");  
    }  
}  
  
public class MilpiesEsquiador extends Milpies {  
    protected int numeroDePiesRotos;  
    public MilpiesEsquiador() {  
        numeroDePiesRotos = 100;  
    }  
}
```

```

public void escribirPies() {
    System.out.println("A un Milpiés esquiador le quedan " +
        (numeroDePies - numeroDePiesRotos) + " pies");
}
}

public class TestMilpies {
    public static void main(String[] args) {
        MilpiesEsquiador m = new MilpiesEsquiador();
    }
}

```

Indicar el motivo por el que el resultado de la ejecución del `main` de `TestMilpies` es `A un Milpiés esquiador le quedan 1000 pies`. Explicar también si el uso de `final` conseguiría cambiar el actual resultado de `TestMilpies`.

10. Sean las siguientes clases del paquete `losAnimales`:

```

public class Animal {
    public void emitirSonido() { System.out.println("Grunt"); }
}

public class Muflon extends Animal {
    public void emitirSonido() { System.out.println("MOOOO!"); }
    public void alimentarCon() { System.out.println("Hierba!"); }
}

public class Armadillo extends Animal {}

public class Guepardo extends Animal {
    public void emitirSonido() { System.out.println("Groar!"); }
}

```

- Si el siguiente programa Java se ubica también en el paquete `losAnimales`, indicar las instrucciones de su `main` que provocan error y las que no y explicar brevemente el motivo.

```

public class Test1Animal {
    public static void main(String[] args) {
        adoptar(new Armadillo());
        Object o = new Armadillo();
        Armadillo a1 = new Animal();
        Armadillo a2 = new Muflon();
    }

    private static void adoptar(Animal a) {
        System.out.println("Ven, cachorrito!");
    }
}

```

- Tracear el resultado de la ejecución del siguiente programa:

```
package losAnimales;
public class Test2Animal {
    public static void main(String[] args) {
        Animal a = new Armadillo();
        a.emitirSonido();
        a = new Muflon();
        a.emitirSonido();
        a = new Guepardo();
        a.emitirSonido();
    }
}
```

En función del resultado obtenido y siguiendo las reglas de la herencia, ¿qué modificaciones se deberían realizar en la jerarquía para exigir que todos los animales emitan el sonido **Grunt**? ¿Cómo se podría conseguir saber el tipo de alimentación de cada animal?

11. Modificar la jerarquía **Animal** para garantizar que cada clase derivada de **Animal** defina el método **alimentarCon()** y con ello conocer el tipo de alimentación de cada **Animal**.

Más Información

- [Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.
URL: <http://math.hws.edu/javanotes/>. Capítulo 5 (5.5, 5.6 y 5.7).
- [Ora11d] Oracle. *The JavaTM Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Learning the Java Language. Lesson: Language Basics - Interfaces and Inheritance.
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.
Capítulos 7 y 8.
- [Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulos 2, 3 y 4.

Capítulo 15

Tratamiento de errores

El objetivo básico que se pretende cubrir en este capítulo es el de incorporar al proceso de diseño de una aplicación Java las tareas de previsión y tratamiento de las anomalías o fallos que se pueden producir durante su ejecución, evitando así, en la medida de lo posible, su terminación abrupta o que obtenga resultados incorrectos. A los fallos o condiciones anormales que ocurren durante la ejecución de un segmento de código se las denomina *excepciones*. Al ocurrir una excepción, ésta debe ser tratada porque, de lo contrario, la ejecución de la secuencia de código se detendrá dando un error en tiempo de ejecución. En el lenguaje Java, las excepciones son objetos que describen una condición excepcional que se produce durante la ejecución de un fragmento de código. Dicho objeto se envía al causante de la excepción, que puede tratar la aparición de la misma (mediante instrucciones explícitas de tratamiento), o no tratarla dejando que sea el sistema (dando un error) el encargado de ello.

Antes de entrar en detalle con la gestión de excepciones, se definen y clasifican los diversos fallos o anomalías que se pueden producir durante la ejecución de una aplicación, presentando la jerarquía `Throwable` como el instrumento que proporciona el estándar de Java para su representación, tanto la de los fallos que sí son recuperables (*excepciones*) como la de los que no lo son (*errores*); en particular, se indica cómo reutilizar vía herencia esta jerarquía para diseñar las denominadas *excepciones de usuario* (en inglés, *user-defined exceptions*) o excepciones *ad hoc* para una aplicación, con las que prevenir y tratar los resultados atípicos que obtienen, en determinadas circunstancias, algunos de sus métodos.

15.1 Fallos de ejecución y su modelo Java

Durante la ejecución de una aplicación pueden producirse fallos o anomalías que conducen a su terminación abrupta o, lo que es aún peor, a la obtención de resultados incorrectos. Incluso a un programador novel le resultan familiares fallos como los de entrada/salida que, por ejemplo, resultan de leer datos cuyo formato o rango no se ajusta al previsto –un valor negativo como radio de un círculo, un carácter como opción de un menú numérico–, de un acceso indebido (sin los permisos oportunos o empleando un nombre incorrecto) a un fichero, etc. También son muy habituales, cuando se aprende a programar, fallos en la lógica de la aplicación provocados por los resultados atípicos que pueden obtener algunos de sus métodos como, por ejemplo, los que se producen en la aplicación que manipula una secuencia de círculos cuando se busca o borra un círculo dado que no está en una secuencia, y fallos de programación como dividir por cero, calcular la raíz cuadrada de un valor negativo y acceder a una posición inexistente de un array o a un objeto `null`. A pesar de ser de tipos diferentes, los fallos de ejecución descritos se clasifican como *excepciones* porque resulta factible o razonable evitar sus consecuencias y reconducir la ejecución de la aplicación una vez se producen; justo lo contrario sucede con los denominados *errores*, fallos de ejecución que por ser muy severos o por involucrar al soporte *hardware* de la aplicación (por ejemplo, agotar la memoria o acceder indebidamente a una de sus zonas) resultan de difícil o imposible solución y, por ello, irrecuperables.

Pues bien, asumiendo que una estricta disciplina de programación y prueba (*debugging*) elimina los fallos de programación, es más que razonable pensar que ya en el proceso de diseño de una buena aplicación Java se debe advertir de la presencia y gestionar el tratamiento de aquellas excepciones que puedan producirse al ejecutarla; para ello, antes es imprescindible introducir los instrumentos que el lenguaje Java proporciona para la representación y clasificación de cualquier fallo de ejecución.

15.1.1 La jerarquía `Throwable`

El lenguaje Java proporciona, en su paquete estándar `java.lang`, la clase `Throwable`, derivada de `Object`, que representa cualquier fallo de ejecución independientemente de su tipo. Es más, la reutilización vía herencia de la clase `Throwable` resulta obligatoria para organizar y diseñar todas las clases que componen la jerarquía `Throwable`, jerarquía que reproduce la clasificación de los fallos de ejecución anteriormente citados.

Como se detalla a continuación, es el propio lenguaje Java quien establece implícitamente el procedimiento a seguir y las características que, como mínimo, debe exhibir cualquier clase de esta jerarquía al ofrecer, ya diseñadas en distintos paquetes de su estándar, la mayoría de las subclases de `Throwable`. Para empezar por el

ejemplo quizás más completo y relevante en lo que a organización y representación de fallos de ejecución se refiere, cabe detenerse a observar en la documentación del *API* de Java [Ora11c] el árbol de la jerarquía `Throwable` en el paquete estándar `java.lang` mostrado en la figura 15.1.

Se puede observar que, en efecto, para poder incluir cualquier fallo de ejecución entre las variables que crea y manipula una aplicación, y así poder tratarlos según convenga, el lenguaje Java deriva de `Object` la clase `Throwable`; a su vez, visto que una excepción es un fallo de ejecución -recuperable- y que también un error es un fallo de ejecución -aunque irrecuperable-, las clases `Error` y `Exception` se diseñan como derivadas directas de `Throwable`, la base de la jerarquía Java de errores y excepciones.

También es importante destacar que `Exception` y `Error` son además las raíces de sendas jerarquías donde coexisten clases que representan tipos concretos de excepciones y errores junto con subárboles de clases que representan subtipos de éstos; así, por ejemplo, de `Exception` se derivan tanto `ClassNotFoundException` que representa la excepción que se produce al cargar una clase empleando un nombre incorrecto como `RuntimeException` que es la clase raíz de la jerarquía que contiene todos los fallos de programación, tan conocidos por otra parte que para identificarlos basta con fijarse en sus nombres, por ejemplo:

```
ArithmaticException
ClassCastException
IllegalArgumentException:
    NumberFormatException
IndexOutOfBoundsException:
    ArrayIndexOutOfBoundsException
    StringIndexOutOfBoundsException
NullPointerException
```

Pero, como se ha comentado anteriormente, `java.lang` no es el único paquete estándar donde se ubican componentes de la jerarquía `Throwable`. Por no alargar demasiado esta exposición, se analiza ahora el rol y la organización de algunas clases de esta jerarquía únicamente en `java.util` y `java.io`, aunque se recomienda investigar en la documentación del *API* de Java [Ora11c] para averiguar algo más sobre el tema; así pues, en las figuras 15.2 y 15.3 y se muestran los árboles `Throwable` que la documentación del *API* proporciona para los dos paquetes estándar citados.

De todas las clases que aparecen en estas figuras, dos en particular resultan reseñables por su importancia a la hora de representar las excepciones de entrada/salida. Mencionar en primer lugar la clase `IOException` en `java.io`, pues representa cualquier excepción provocada por un fallo o una interrupción en una operación de entrada o salida, como se verá en el capítulo 16. En segundo lugar, cabe destacar



Figura 15.1: Jerarquía Throwable en `java.lang`.

- [java.lang.Throwable](#) (implements [java.io.Serializable](#))
 - [java.lang.Error](#)
 - [java.io.IOException](#)
 - [java.lang.Exception](#)
 - [java.io.IOException](#)
 - [java.io.CharConversionException](#)
 - [java.io.EOFException](#)
 - [java.io.FileNotFoundException](#)
 - [java.io.InterruptedIOException](#)
 - [java.io.ObjectStreamException](#)
 - [java.io.InvalidClassException](#)
 - [java.io.InvalidObjectException](#)
 - [java.io.NotActiveException](#)
 - [java.io.NotSerializableException](#)
 - [java.io.OptionalDataException](#)
 - [java.io.StreamCorruptedException](#)
 - [java.io.WriteAbortedException](#)
 - [java.io.SyncFailedException](#)
 - [java.io.UnsupportedEncodingException](#)
 - [java.io.UTFDataFormatException](#)

Figura 15.2: Jerarquía Throwable en `java.io`.

la clase `InputMismatchException` en `java.util`, que representa la excepción que se produce cuando con un objeto `Scanner` se lee un dato cuyo formato o rango no se corresponde con el del tipo esperado como, por ejemplo, cuando se lee con `nextInt()` un número real o una cadena de caracteres en lugar de un valor `int`.

Vista su organización, restan ahora por introducir la funcionalidad básica y las características de las que dota el lenguaje Java a cualquier clase de la jerarquía `Throwable`. En lo que a funcionalidad respecta, la clase `Throwable` y sus derivadas suelen constar de dos constructores, uno con un único parámetro de tipo `String` y el constructor por defecto (sin parámetros); cualquiera de ellos construye un objeto que almacena el estado de la ejecución en el instante en el que se produce el fallo que representa. Cualquier excepción tiene una variable de instancia de tipo `String` con un mensaje que, normalmente, identifica el motivo de la excepción. El `String` argumento del primer constructor inicializa dicha variable de instancia, es decir, se utiliza para añadir información sobre el fallo acaecido. De todos los métodos heredados de la clase `Throwable` cabe destacar, por su utilidad, los métodos `getMessage`, `printStackTrace` y `toString`. El método `getMessage` devuelve el `String` con el mensaje de la excepción. El método `printStackTrace` imprime el tipo de la excepción y su traza (secuencia de llamadas en la pila de ejecución) en la salida de error estándar. Suele usarse en tareas de depuración, facilitando al programador la localización del fallo producido. El método `toString` sólo proporciona una muy breve descripción de la información que almacena el objeto sobre el que se aplica. Si el objeto se ha creado haciendo uso del constructor sin argu-

- [java.lang.Throwable](#) (implements [java.io.Serializable](#))
 - [java.lang.Error](#)
 - [java.util.ServiceConfigurationError](#)
 - [java.lang.Exception](#)
 - [java.io.IOException](#)
 - [java.util.InvalidPropertiesFormatException](#)
 - [java.lang.RuntimeException](#)
 - [java.util.ConcurrentModificationException](#)
 - [java.util.EmptyStackException](#)
 - [java.lang.IllegalArgumentException](#)
 - [java.util.IllegalFormatException](#)
 - [java.util.DuplicateFormatFlagsException](#)
 - [java.util.FormatFlagsConversionMismatchException](#)
 - [java.util.IllegalFormatCodePointException](#)
 - [java.util.IllegalFormatConversionException](#)
 - [java.util.IllegalFormatFlagsException](#)
 - [java.util.IllegalFormatPrecisionException](#)
 - [java.util.IllegalFormatWidthException](#)
 - [java.util.MissingFormatArgumentException](#)
 - [java.util.MissingFormatWidthException](#)
 - [java.util.UnknownFormatConversionException](#)
 - [java.util.UnknownFormatFlagsException](#)
 - [java.lang.IllegalStateException](#)
 - [java.util.FormatterClosedException](#)
 - [java.util.MissingResourceException](#)
 - [java.util.NoSuchElementException](#)
 - [java.util.InputMismatchException](#)
- [java.util.TooManyListenersException](#)

Figura 15.3: Jerarquía Throwable en `java.util`.

mentos, devuelve el tipo de la excepción. Si se ha creado mediante el constructor con un argumento, entonces el resultado es la concatenación de tres `String`: el tipo de la excepción, “:” y el resultado de `getMessage` para este objeto. Por ello, lo más aconsejable, en la mayoría de las ocasiones, es usar el constructor con un argumento; para ilustrar este punto basta con ejecutar el siguiente código:

```
Scanner teclado = new Scanner(System.in).useLocale(Locale.US);
System.out.print("Introduce el número 0.3: ");
String leido = teclado.next();
if (leido.equals("0.3")) {
    System.out.print("La lectura con teclado.nextInt() ");
    System.out.println("provocaría un:");
    System.out.println(new InputMismatchException());
    System.out.println("En concreto: ");
    System.out.println(new InputMismatchException("al leer "
        + leido + " por 0.3"));
}
```

y observar atentamente su resultado, que es:

Entrada/Salida Estándar
Introduce el número 0.3: 0.3 La lectura del dato con teclado.nextInt() provocaría un: java.util.InputMismatchException En concreto: java.util.InputMismatchException: al leer 0.3 por 0.3

Nótese entonces que el constructor sin parámetros de una clase de la jerarquía **Throwable** tiene asociado por defecto el mensaje **null**.

Pero aún más que su funcionalidad mínima, se debe destacar la característica más novedosa que un objeto **Throwable** tiene frente a los que no lo son: en lugar de ser devuelto (*return*), un objeto **Throwable** se lanza (*throw*) como resultado del método si en su ejecución se origina la excepción, para ello se utiliza la cláusula **throw** que se estudiará en la siguiente sección del capítulo. Esta característica de ser lanzado fuera de la secuencia normal de retorno de datos es precisamente la que da nombre a la clase raíz de la jerarquía y, gracias a ella, es posible ya en la fase de diseño de una aplicación advertir dónde, cuándo y qué fallo de ejecución se ha producido y, en su caso, cuál es la gestión a realizar para evitar sus consecuencias; es por ello también que en la documentación del *API* de Java [Ora11c] de muchos métodos aparece precedida por la palabra **Throws** la información relativa a sus resultados en condiciones anómalas, como se muestra en la figura 15.4 para el método **nextDouble()** de la clase **Scanner** ubicada en **java.util**.

nextDouble

public double nextDouble()

Scans the next token of the input as a **double**. This method will throw **InputMismatchException** if the next token cannot be translated into a valid double value. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the [Float](#) regular expression defined above then the token is converted into a **double** value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via [Character.digit](#), prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to [Double.parseDouble](#). If the token matches the localized NaN or infinity strings, then either "NaN" or "Infinity" is passed to [Double.parseDouble](#) as appropriate.

Returns:

the **double** scanned from the input

Throws:

[InputMismatchException](#) - if the next token does not match the *Float* regular expression, or is out of range
[NoSuchElementException](#) - if the input is exhausted
[IllegalStateException](#) - if this scanner is closed

Figura 15.4: Documentación del método **nextDouble()** en **java.util.Scanner**.

15.1.2 Ampliación de la jerarquía Throwable con excepciones de usuario

Como ya se ha comentado previamente, la jerarquía `Throwable` del estándar de Java no contempla la representación de los fallos en la lógica de una aplicación precisamente por su carácter *ad hoc*. Una excepción de este tipo se produce al ignorar el resultado atípico que obtiene en ciertas circunstancias alguno de los métodos que usa una aplicación dada. Además, existe otro tipo de errores que tampoco serían advertidos por el compilador ni se considerarían fallos de programación.

En el programa de la figura 15.5 se realiza la lectura de una serie de mediciones de lluvia que se almacenan en un array. En dicho programa pueden tener lugar dos posibles errores, uno de ejecución y otro de la semántica interna del problema. El primero sería un error de ejecución que se produciría en la lectura desde teclado de un `int`, si el usuario introdujese un entero negativo como número de mediciones. En efecto, se produciría la excepción `java.lang.NegativeArraySizeException`.

El segundo error es más sutil. El usuario podría introducir alguna de las mediciones negativa. Esto no provocaría un error de ejecución (en este punto) pero sí podría dar lugar a algún error en otro punto del programa, ya que resulta absurdo que haya llovido, por ejemplo, -5 litros.

Podría resultar útil tener una excepción que indicase precisamente esta circunstancia, por ejemplo, `ExcepcionNumeroNegativo` que se pudiera crear cuando cualquiera de ambos errores se hubiese producido, sin necesidad de esperar a que se produjese un error de ejecución en otro punto del programa debido a dichos errores.

Para poder ser tratado en Java, cualquier fallo en la lógica de una aplicación debe ser representado mediante una nueva clase de la jerarquía `Throwable`, en concreto, como una nueva clase derivada de `Exception`. A estas nuevas clases, para distinguirlas de las que ya figuran en la jerarquía `Throwable` del estándar de Java, se las denomina *excepciones de usuario* (en inglés, *user-defined exceptions*) aunque, en principio, poseen la misma funcionalidad básica y características que cualquier otra con los métodos que heredan de `Exception`.

La clase `ExcepcionNumeroNegativo` se podría definir como sigue¹:

```
public class ExcepcionNumeroNegativo extends Exception {  
    public ExcepcionNumeroNegativo() { super(); }  
    public ExcepcionNumeroNegativo(String msg) { super(msg); }  
}
```

En este caso, el código de la clase `Mediciones` podría lanzar la excepción de usuario al detectar el número negativo antes de tratar de crear el array o de almacenar un valor negativo en una componente del mismo.

¹Como en todas las clases, si no se explicitan métodos constructores, el sistema incluye por defecto el constructor sin parámetros.

```

import java.util.*;
public class Mediciones {
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in).useLocale(Locale.US);
        double[] elArray = recabarDatosLluvia(teclado);
        System.out.print("Array Mediciones: { ");
        for (int i=0; i<elArray.length; i++)
            if (i!=elArray.length-1) System.out.print(elArray[i]+", ");
            else System.out.print(elArray[i]+"}");
    }

    /** Lectura de las mediciones y construcción del array */
    public static double[] recabarDatosLluvia(Scanner t) {
        System.out.print("Introduzca el número de mediciones: ");
        int num = t.nextInt();
        double[] resultado = new double[num];
        System.out.println("Introduzca las precipitaciones medidas:");
        for (int i=0; i<num; i++) {
            System.out.print("Litros de la medición " + (i+1) + ": ");
            resultado[i] = t.nextDouble();
        }
        return resultado;
    }
}

```

Figura 15.5: Clase Mediciones.

Sin embargo, más allá de detectar o crear las excepciones, lo más importante de las mismas es el concepto de *tratamiento* de una excepción que se aborda en la siguiente sección.

15.2 Tratamiento de excepciones

Se ha visto que una excepción provoca la terminación abrupta del programa. Sin embargo, una excepción puede tratarse para evitar dicha terminación abrupta. Existen dos maneras de tratar una excepción: *capturarla* y *propagarla*. Ambos conceptos son sencillos de entender con ejemplos:

- Supóngase que en una base de datos hay que realizar un proceso muy delicado que puede que no salga bien. Si el proceso sufre algún error se perdería todo el trabajo hecho. Por ejemplo, al actualizar la base de datos, si el proceso se interrumpe por un error, la base de datos puede quedar inconsistente.

- Supóngase que en una empresa un empleado detecta un error gravísimo en la contabilidad de la misma; sus alternativas son:
 - tratar de arreglar el problema que se ha producido,
 - informar a su superior (el contable) que se encargará del problema. Éste, a su vez, puede considerar que el error es leve y solucionarlo él mismo o, por el contrario, que no tiene autoridad suficiente para resolverlo, con lo cual informará a su superior, el jefe de contabilidad que puede hacerse cargo del problema o informar al subdirector financiero y así sucesivamente.

En el primer caso, lo ideal sería que los cambios de la actualización no se hicieran efectivos hasta que el proceso se hubiese completado con éxito. De esa manera si al *tratar* de hacer el proceso hay algún problema, en realidad, no se ha perdido el resto de la información.

En el segundo caso, el empleado puede *propagar* el informe de detección del problema por toda la escala de mando de la empresa hasta que llegue al nivel adecuado en la jerarquía para resolverlo.

15.2.1 Captura de excepciones: try/catch/finally

En el primer ejemplo planteado, se pretendía resaltar que es posible que algunas de las acciones que se realizan en un programa pueden quedar comprometidas por la aparición de una excepción. La solución consiste en capturar dicha excepción y marcar el fragmento de código que se quiere que sólo se realice si la operación que puede lanzar la excepción no se produce. Por ejemplo, en la figura 15.5, la lectura del *int* que representa el número de mediciones en el método *recabarDatosLluvia* podría realizarse como sigue:

```
System.out.print("Introduzca el número de mediciones: ");
int num = t.nextInt();
double[] resultado;
try {
    resultado = new double[num];
}
catch (NegativeArraySizeException e) {
    System.out.println("Se ha producido el error " + e);
    System.out.println(num + "<0 --> se le cambia el signo.");
    System.out.println("Array de " + (-num) + " elementos.");
    num=-1;
    resultado = new double[num];
}
```

La interpretación del código anterior sería la siguiente: se intenta (*try*) efectuar un bloque de instrucciones que puede provocar una excepción. Si se tiene éxito, la ejecución del programa prosigue pero si se produce la excepción esperada, se captura (*catch*) y se arregla el problema antes de continuar; el programa no termina

de forma abrupta al producirse el error sino que, simplemente, se guardan valores válidos para continuar, de manera que el resto de trabajo realizado no se pierda.

La sintaxis de las cláusulas `try/catch/finally` es la que sigue:

```
try {
    ... // instrucciones que pueden provocar alguna excepción
}
catch ( NombreDeExcepcion1 nomE1 ) {
    ... // instrucciones a realizar si se produce
        // la excepción NombreDeExcepcion1
}
...// puede haber X catch por cada try
catch ( NombreDeExcepcionX nomEX ) {
    ... // instrucciones a realizar si se produce
        // la excepción NombreDeExcepcionX
}
finally {
    ... // bloque opcional, instrucciones a realizar tanto
        ... // si se ha producido una excepción como si no
}
```

Básicamente, el significado es como sigue: se intenta (`try`) ejecutar un bloque de código en el que pueden ocurrir errores que se representan con alguna de las excepciones que se explicitan en los bloques `catch`. Si se produce un error, el sistema lanza una excepción (`throws`) que puede ser capturada (`catch`) en base al tipo de excepción, ejecutándose las instrucciones correspondientes. Finalmente, tanto si se ha producido o no una excepción y si ésta ha sido o no tratada, se ejecutan las instrucciones asociadas a la cláusula `finally`. Al finalizar todo el bloque, la ejecución se reanuda del modo habitual.

Siempre que aparece una cláusula `try`, debe existir al menos una cláusula `catch` o `finally`. Nótese que, para una única cláusula `try`, pueden existir tantas `catch` como sean necesarias para tratar las excepciones que se puedan producir en el bloque de código del `try`. Cuando en el bloque `try` se lanza una excepción, los bloques `catch` se examinan en orden, y el primero que se ejecuta es aquel cuyo tipo sea compatible con el de la excepción lanzada. Así pues, el orden de los bloques `catch` es importante. Por ejemplo, el orden en los bloques `catch` que siguen no sería adecuado:

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

Con este orden, el segundo bloque `catch` nunca se alcanzaría, puesto que todas las excepciones serían capturadas por el primero, ya que la excepción `ExcepcionNumeroNegativo` se deriva de la clase `Exception`. Afortunadamente, el compilador advierte sobre esto. El orden correcto consiste en invertir los bloques `catch` para que la excepción más específica aparezca antes que cualquier excepción de una clase antecesora.

El bloque precedido por `finally` es opcional si hay al menos un `catch`. Contiene un grupo de instrucciones que se ejecutarán tanto si el `try` tiene éxito como si no. Aparentemente funciona igual si se sitúa una instrucción después del último `catch` o dentro del `finally`, como en el ejemplo que sigue:

```
try {
    ... // instrucciones que pueden generar una excepción
    System.out.println("Intento logrado");
}
catch (NombreExcepcion e) {
    System.out.println("Capturada!");
}
finally { System.out.println("Y el finally!"); }
```

En el ejemplo, los puntos suspensivos representan las instrucciones que pueden producir la excepción `NombreExcepcion` capturada en el `catch`. Tanto en un caso como en otro, el final de la ejecución del fragmento sería la escritura del texto “**Y el finally!**”. Sin embargo, ¿qué ocurriría si durante la ejecución de los puntos suspensivos se produjese una excepción diferente a `NombreExcepcion`, por ejemplo `OtraExcepcion`? La respuesta es que aún así, antes de la terminación abrupta del código, se efectuaría el `finally`.

(sin Excepcion):	(con NombreExcepcion):	(con OtraExcepcion):
Intento logrado	Capturada!	Y el finally!
Y el finally!	Y el finally!	

Esto puede quedar más claro si se piensa que en el `finally` pueden ir instrucciones como guardar datos en ficheros o bases de datos, etc. Es decir, aquellas instrucciones que salvaguardan la integridad de la información o de la ejecución.

Ejemplo 15.1. Supóngase que se desea definir un método para leer un entero positivo desde la entrada estándar. Dos son los errores que se prevee puedan producirse: el primero, un error de ejecución si el valor leído no es un entero y el segundo, un error en la lógica de la aplicación si el usuario introduce un entero pero negativo. El siguiente método resuelve el problema capturando la excepción `InputMismatchException` para el primer caso y utilizando una instrucción condicional para el segundo, dando opción en ambos casos a que el usuario introduzca de nuevo un valor. Nótese que, para cualquier posible lectura, siempre se ejecuta

la instrucción `tec.nextLine()` de la cláusula `finally`, permitiendo descartar el salto de línea que se almacena en el buffer de entrada cuando el usuario pulsa la tecla *Enter*.

```
static int leerEnteroPositivo(Scanner t) {
    boolean salir = false;
    int leido = 0;
    do {
        try {
            System.out.print("Introduce un entero positivo: ");
            leido = t.nextInt();
            if (leido<0)
                System.out.println("Error: no es positivo");
            else salir = true;
        }
        catch (InputMismatchException e) {
            System.out.println("Error: no es un entero");
        }
        finally {
            tec.nextLine();
        }
    } while(!salir);
    return leido;
}
```

15.2.2 Propagación de excepciones: throw versus throws

Si la excepción se produce dentro de un método, puede elegirse dónde se trata. Si se trata dentro del método se utilizan las cláusulas ya vistas `try/catch/finally` pero si no, puede decidirse su propagación hacia el punto del programa desde donde se invocó el método. Para hacer esto es necesario que el método tenga en su cabecera la cláusula `throws` y el nombre de la clase de excepción (o clases, separadas por comas) que se propagarán desde dicho método. De este modo, si se produce una excepción dentro del cuerpo del método se abortará la ejecución del mismo y se lanzará la excepción hacia el punto desde donde se invocó al método. Así, el método del ejemplo 15.1 se puede modificar para que en el caso de que se produzca un error de formato, la excepción se propague en lugar de capturarla, como sigue:

```
static int leerEnteroPositivo(Scanner t) throws InputMismatchException {
    int leido;
    do {
        System.out.print("Introduce un entero positivo: ");
        leido = t.nextInt();
        if (leido<0)
            System.out.println("Error: no es positivo");
    } while(leido<0);
    return leido;
}
```

Si la cabecera del método no incluyese la cláusula `throws`, toda la ejecución del programa acabaría en la lectura desde teclado del `int`, independientemente de desde dónde se hubiese invocado al método `leerEnteroPositivo()`. Sin embargo, pese a que no se puedan encontrar en este fragmento de código las cláusulas `try/catch/finally`, la ejecución no termina de forma abrupta en este punto. Lo que sí que sucede es que el método termina sin llegar al `return`, es decir, el método sí termina pero el programa no. La excepción se propaga porque en la cabecera del método aparece la cláusula `throws` de manera que, en otra parte del programa, podría encontrarse lo siguiente:

```
Scanner tec = new Scanner(System.in).useLocale(Locale.US);
boolean salir = false;
int resp;
while (!salir) {
    try {
        resp = leerEnteroPositivo(tec);
        System.out.println("Se ha leído el número " + resp);
        salir = true;
    }
    catch (InputMismatchException e) {
        System.out.println("Error: no es un entero");
    }
    finally {
        tec.nextLine();
    }
}
```

En este caso, el bucle `while` no tiene terminación abrupta porque se produzca una excepción en la llamada a `leerEnteroPositivo` al introducir `String` en lugar de `int`, sino que se captura y prosigue hasta que se introduzca un valor válido y se incumpla la condición de continuación del bucle.

Sin embargo, no sólo se puede escoger que el método propague una excepción; también se puede escoger el punto y las condiciones bajo las cuales se produce dicha excepción. Antes de construir (`new`) la nueva excepción se escribe la cláusula `throw` (sin “s” final) de manera que se indica el punto exacto donde se origina y propaga la excepción, como puede verse en el código de la figura 15.6, donde cuando el usuario introduce un entero negativo se lanza la excepción `ExcepcionNumeroNegativo` definida anteriormente.

```
import java.util.*;
public class LecturaEnteroPositivo {
    public static void main(String[] args) {
        Scanner tec = new Scanner(System.in).useLocale(Locale.US);
        boolean salir = false;
        int respuesta;
        while (!salir) {
            try {
                respuesta = leerEnteroPositivo(tec);
                System.out.println("Se ha leído el número " + respuesta);
                salir = true;
            }
            catch (InputMismatchException e) {
                System.out.println(e + ": Error: no es un entero");
            }
            catch (ExpcionNumeroNegativo e) {
                System.out.println(e + ": Error: no es positivo");
            }
            finally {
                tec.nextLine();
            }
        }
    }

    private static int leerEnteroPositivo(Scanner t)
        throws InputMismatchException,
               ExpcionNumeroNegativo {
        System.out.print("Introduce un entero positivo: ");
        int leido = t.nextInt();
        if (leido<0) throw new ExpcionNumeroNegativo("número " + leido);
        return leido;
    }
}
```

Figura 15.6: Clase LecturaEnteroPositivo.

Un ejemplo de ejecución del método `main` de la clase de la figura 15.6 sería el siguiente:

Entrada/Salida Estándar
Introduce un entero positivo: Hola java.util.InputMismatchException: Error: no es un entero Introduce un entero positivo: -7 ExcepcionNumeroNegativo: número: -7: Error: no es positivo Introduce un entero positivo: 10 Se ha leído el número 10

Es importante no confundir las dos cláusulas: `throws NombreExcepcion` se sitúa en la cabecera del método e indica que, si se produce una excepción de ese tipo, el método la propagará al punto de invocación del mismo; `throw` indica que se lanza la excepción que se escribe a continuación.

15.2.3 Excepciones *checked/unchecked*

Como ya se ha comentado, todas las situaciones de error se representan en Java como excepciones que hay que tratar. Sin embargo, en los capítulos previos a éste aparecen muchas líneas de código sin tratamiento de excepciones. Esto se explica porque en Java se pueden distinguir dos tipos de excepciones:

- las excepciones *unchecked* o “no comprobadas” que heredan de `RuntimeException` y
- las excepciones *checked* o “comprobadas”.

La diferencia está en que para las *checked* el compilador de Java obliga a tratar la excepción en un bloque `catch` o bien propagar la excepción en el `throws` para obligar al método que invoca a tratarla o propagarla a su vez. Para las *unchecked* esto no es obligatorio, aunque sí es posible, como se ha visto en ejemplos anteriores con la excepción `InputMismatchException`. Nótese que si una excepción de usuario se define como derivada directa de `Exception` debe tratarse como una excepción *checked*.

Las clases derivadas de `java.lang.Error` también pueden considerarse como *unchecked* puesto que su tratamiento o propagación no es obligatorio.

15.3 Problemas propuestos

1. Escribir una clase de utilidades `LecturaValidada` que permita:

- a) Leer un número entero.
- b) Leer un número entero en un rango determinado.
- c) Leer un número real.
- d) Leer un número real positivo.
- e) Leer un número real en un rango determinado.

Se deben prever todos los errores posibles que puedan suceder y tratarlos en los métodos correspondientes.

2. Dada la clase `Figura` que contiene los atributos `String color`, `String tipo` y el método `double area()`, se ha definido el método `equals` de la manera siguiente:

```
public boolean equals(Object o) {
    Figura f = (Figura) o;
    return this.color.equals(f.color) &&
           this.nombre.equals(f.nombre) &&
           this.area()==f.area();
}
```

Sin embargo, esto no es del todo correcto. Si se ejecutases las siguientes instrucciones:

```
Figura f1 = new Figura("rojo","cuadrado");
Figura f2 = new Figura("rojo","cuadrado");
Double d = new Double(1.0);
String k = "Hello";
boolean b1 = f1.equals(f2);
boolean b2 = d.equals(k);
boolean b3 = k.equals(f2);
boolean b4 = f1.equals(d);
```

`b1`, `b2` y `b3` se evaluarían, respectivamente, a `true`, `false` y `false` pero al evaluar `b4` se produciría la excepción `unchecked ClassCastException`.

Modificar el método `equals` para que si se produce la excepción `unchecked ClassCastException` ésta sea capturada y tratada en el cuerpo de dicho método y se comporte como el `equals` definido en la clase `Object`.

3. Se está implementando una aplicación para la gestión de una agenda telefónica. Se dispone de un método `menu` que muestra por pantalla un menú de opciones:

```
private int menu(Scanner teclado) {
    System.out.println("      Menú de Agenda ");
    System.out.println("-----");
    System.out.println("1.- Cargar Fichero Agenda");
    System.out.println("2.- Guardar Fichero Agenda");
    System.out.println("3.- Buscar Nombre");
    System.out.println("4.- Insertar Nuevo Nombre");
    System.out.println("5.- Eliminar Nombre");
    System.out.println("0.- Salir");
    System.out.print("Seleccione [0..5]: ");
    return teclado.nextInt();
}
```

Dicho método es invocado desde otro método como sigue:

```
Scanner tec = new Scanner(System.in).useLocale(Locale.US);
...
int opcion = menu(tec);
switch(opcion) {
    case 0: ...
    case 1: ...
    case 2: ...
    case 3: ...
    case 4: ...
    case 5: ...
}
```

Cuando el programa es probado por el usuario, se detecta que, en ocasiones, el programa aborta su ejecución porque el usuario se equivoca y escribe números que no están en el menú o texto en lugar de dichos números.

Se pide:

- a) Definir una nueva excepción de usuario `NumeroFueraDeRango` para identificar el error de que el usuario haya escrito un número de opción en el menú fuera del rango [0..5].
- b) Modificar el método `menu` para que, en caso de que el número introducido esté fuera del rango [0..5], se lance la nueva excepción conteniendo como mensaje “la opción elegida ha sido X”.
- c) Modificar el fragmento de código dado para que se detecte si el número introducido está en el rango correcto y si no es así se vuelva a presentar el menú hasta que el usuario acierte.

- d) Modificar el fragmento de código dado para que también se detecte si el usuario ha introducido un valor que no sea un entero y si no es así se vuelva a presentar el menú.
4. Dado el siguiente fragmento de código:

```

static void metodo1() throws Expcion1, Expcion2 {
    ...
}

static void metodo2(){
    try { ... (2) }
    catch (IndexOutOfBoundsException e) {
        System.out.println("texto0");
    }
}

static void metodo3() throws Expcion3, Expcion1 {
    ...
}

static void main(String[] args) {
    try {
        metodo1();
        metodo2();
        metodo3();
    }
    catch (Expcion1 e) { System.out.println("texto1"); }
    catch (Expcion2 e) { System.out.println("texto2"); }
    catch (Expcion3 e) { System.out.println("texto3"); }
    catch (InputFormatException e) {
        System.out.println("texto4");
    }
    finally { System.out.println("texto5"); }
}

```

Indicar qué aparece por pantalla si en los puntos suspensivos marcados se producen las siguientes excepciones:

- a) En (1) la excepción de usuario Expcion1.
- b) En (1) la excepción *unchecked* IndexOutOfBoundsException.
- c) En (1) la excepción de usuario Expcion2.
- d) En (2) la excepción *unchecked* InputFormatException.
- e) En (2) la excepción *unchecked* IndexOutOfBoundsException.
- f) En (3) la excepción de usuario Expcion3.
- g) En (3) la excepción *unchecked* InputFormatException.

5. Ampliar la clase **LecturaValidada** del ejercicio 1 para que permita leer un **String** que deberá pertenecer a los existentes previamente en un array de elementos de dicho tipo. En caso de que el **String** leído exista en el array, el método de lectura que se construya deberá devolver el índice con la posición en el array del **String** leído o, en caso de no existir, deberá lanzar la excepción **ElementoNoExistente** que se habrá definido previamente.

Por ejemplo, dada la declaración de la constante array:

```
public static final String[] COMPOSITORES = {"Bach", "Haydn",
                                              "Mozart", "Beethoven", "Brahms",
                                              "Mahler", "Bartok", "Webern"};
```

El método que se pide deberá, si se utiliza el array **COMPOSITORES** al ejecutarlo, leer un **String** desde el teclado, devolviendo la posición del texto encontrado en el array (en el caso del ejemplo, el nombre del compositor) o lanzar la excepción correspondiente (**ElementoNoExistente**), en caso de que éste no exista.

Más información

- [Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.
URL: <http://math.hws.edu/javanotes/>. Capítulo 3 (3.7) y Capítulo 8 (8.3).
- [Ora11d] Oracle. *The JavaTM Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Essential Java Classes. Lesson: Exceptions.
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.
Capítulo 9.
- [WP00] R. Wiener and L.J. Pinson. *Fundamentals of OOP and data structures in Java*. Cambridge University Press, 2000. Capítulo 7.

Capítulo 16

Entrada y salida: ficheros y flujos

Este capítulo introduce los conceptos necesarios y explica las principales clases Java que permiten leer, escribir y gestionar ficheros. Mediante las operaciones relativas a la Entrada y Salida (E/S) básica, los programas en Java pueden crear y manipular ficheros, bien para escribir información en ellos, bien para leer de los mismos.

Por supuesto, la E/S básica es uno de los elementos fundamentales relativos a los lenguajes de programación. En efecto, los datos de las variables definidas en un programa desaparecen cuando éste finaliza. Por lo tanto, si se desea que los datos generados por un programa puedan estar disponibles para ejecuciones posteriores (o para otros programas), una forma sencilla de hacerlo es almacenándolos en un fichero.

Un fichero es una secuencia de bytes guardados en un dispositivo de almacenamiento secundario (disco duro, USB stick, etc.). Los ficheros se identifican por un nombre y, por lo general, también llevan asociada una extensión que, en algunos sistemas operativos, puede ayudar a identificar el tipo de contenido del mismo. Por ejemplo, resulta lógico pensar que un fichero llamado *canciones.txt* contenga información de texto aunque, en realidad, la extensión de un fichero no determina necesariamente el contenido del mismo.

Es importante destacar que el manejo de ficheros es tan solo un caso particular de la gestión de E/S en Java, donde aparece el concepto de flujo (*stream*). Como ya se vió en el capítulo 7, un flujo representa una secuencia de bytes que se reciben desde una fuente (p.e., teclado, fichero, red, etc.) y se dirigen a un destino (p.e., pantalla, fichero, dispositivo, etc.). Si el flujo de datos se dirige al programa, es decir, representa una entrada de datos para el mismo, entonces se habla de flujo de entrada (al programa). Por el contrario, si el flujo de datos parte del programa,

es decir, se trata de una salida de datos, entonces se habla de flujo de salida (del programa). La noción de *flujo* es muy amplia y mediante distintas extensiones de la misma se tratan en el lenguaje Java muchos aspectos, como los relativos a conexiones remotas con otros ordenadores, manipulación de información multimedia así como E/S estándar y con ficheros.

También hay que destacar que en este capítulo se van a abordar principalmente los ficheros de acceso secuencial, en los cuales la lectura de los datos típicamente se realiza en el mismo orden en el que éstos se han guardado. Por ejemplo, una aplicación de agenda telefónica precisaría almacenar sus datos en un fichero para que estén disponibles siempre que se ejecute la aplicación. La figura 16.1 muestra una posible organización del contenido de dicho fichero, que consta de tantos registros como entradas haya en la agenda y donde cada registro incluye varios campos (nombre, dirección y teléfono). Por lo tanto, la aplicación guarda estos datos en el fichero en un determinado orden y, posteriormente, la lectura se realiza en el mismo orden en el que los datos fueron escritos. Los ficheros secuenciales son posiblemente los más comunes y por eso este capítulo se centra en ellos. No obstante, la sección 16.3.3 aborda las posibilidades que ofrecen los ficheros de acceso aleatorio.

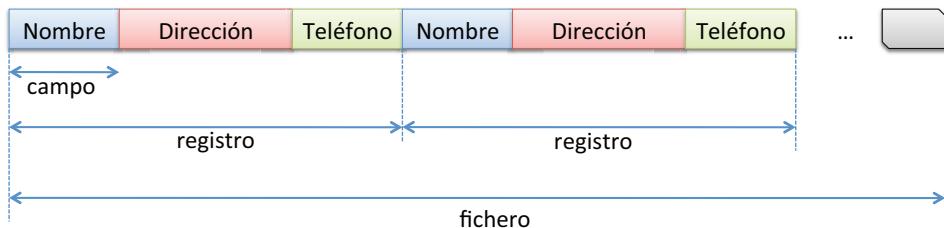


Figura 16.1: Ejemplo de fichero de acceso secuencial que incluye varios registros de una agenda telefónica.

En general, los datos pueden estar almacenados tanto en un fichero de texto como en uno binario, por lo que es muy importante conocer las diferencias, ventajas e inconvenientes de cada tipo. Además, la forma en la que se accede a los ficheros en Java es diferente según se trate de un fichero de texto o uno binario.

Ficheros de texto y binarios

Un *fichero de texto* consiste en una secuencia de caracteres almacenada mediante un esquema de codificación que suele ser ASCII (o UTF-8, que incluye todos los caracteres soportados por ASCII más otros caracteres usados por otras lenguas). Es común, por tanto, encontrar en la literatura el término *fichero ASCII* para referirse a un fichero de texto plano. Un fichero de este tipo es directamente interpretable por una persona, que puede ver el contenido del mismo y los caracteres que en él

hay escritos. Los ficheros de texto son muy portables, lo que significa que pueden ser leídos desde cualquier otro equipo con diferente sistema operativo o desde un lenguaje de programación o editor distinto al empleado para construir el fichero.

Por el contrario, un *fichero binario* consiste en una secuencia de bytes y, por lo tanto, no es directamente interpretable por una persona. Los ficheros binarios se escriben desde un programa, a través de las operaciones de E/S soportadas por un lenguaje de programación. En general, los ficheros binarios no son demasiado portables ya que dependen de la arquitectura de la máquina y del lenguaje de programación empleado para escribir el fichero, para poder interpretar correctamente la información del fichero. Afortunadamente, un fichero binario escrito desde Java puede ser leído desde cualquier otro equipo que tenga un sistema operativo diferente pero, eso sí, la lectura debe ser realizada desde un programa Java.

La principal ventaja de los ficheros de texto, por lo tanto, es la gran portabilidad. No obstante, estos ficheros requieren más tamaño que un fichero binario para representar la misma información¹. Por otra parte, las operaciones sobre ficheros binarios resultan más eficientes que sobre ficheros de texto. Finalmente, es importante recordar que, mientras que un fichero de texto es accesible mediante cualquier editor de textos, un fichero binario requiere la construcción de un programa para su lectura.

16.1 La clase `File`

La clase `java.io.File` permite abstraer el concepto de fichero y directorio desde el lenguaje de programación. Ofrece métodos para interaccionar con el sistema de archivos mediante los que se pueden obtener las principales propiedades de un fichero o directorio (nombre, ruta, si es modificable o no, etc.), así como algunas operaciones sobre los mismos (renombrar, borrar, etc.). En esta sección, se utiliza de forma indistinta el concepto de fichero o directorio ya que la clase `File` los trata de igual manera.

La figura 16.2 muestra un ejemplo de uso de la clase `File`. En primer lugar se procede a la construcción del objeto de tipo `File` (línea 4). Para ello, hay que indicarle una ruta absoluta (como en el ejemplo), que identifica de forma inequívoca un fichero en el sistema de archivos, una ruta relativa al directorio de trabajo, o bien directamente un nombre de fichero. En los dos últimos casos, se asume como directorio de trabajo aquel desde el que se pone en marcha la ejecución del programa.

¹Por ejemplo, un entero de 10 dígitos en un fichero de texto ocuparía 10 bytes (asumiendo codificación ASCII de 1 byte/carácter), mientras que en un fichero binario el mismo número ocuparía los 4 bytes correspondientes a almacenar un tipo `int`.

```

1 import java.io.*;
2 public class TestFile {
3     public static void main(String[] args) {
4         File f = new File("/tmp/file.txt");
5         if (f.exists()) System.out.println("El fichero existe!");
6         else System.err.println("El fichero NO existe!");
7
8         System.out.println("getName(): " + f.getName());
9         System.out.println("getParent(): " + f.getParent());
10        System.out.println("length(): " + f.length());
11    }
12 }
```

Figura 16.2: Ejemplo de uso de la clase `File`.

La creación de un objeto `File` no implica la construcción de un fichero en el sistema de archivos. De hecho, la ruta al fichero especificado ni siquiera tiene que corresponder con un fichero que realmente exista. La clase `File` proporciona métodos para verificar posteriormente si realmente existe un fichero en dicha ruta.

Conviene destacar que para especificar una ruta a un fichero en Windows mediante un `String`, puede hacerse tanto mediante el uso de la barra estándar del Unix, como utilizando doble contrabarra (debido al uso especial del carácter ‘\’ en Java). A continuación se muestran los dos métodos alternativos, empleados para definir el acceso a un fichero en un sistema Windows.

```
File f = new File("C:/Users/lucas/file.txt");
File f = new File("C:\\Users\\\\lucas\\\\file.txt");
```

Adicionalmente, para la definición de rutas de acceso a ficheros existen en Java, definidos en la clase `File`, los atributos de tipo `char`:

- `File.separatorChar`,
- `File.pathSeparatorChar`.

que contienen durante la ejecución de un programa el carácter utilizado en el sistema operativo en uso (Linux, MacOSX, Windows. etc.) bien como separador de ficheros bien como separador en la expresión de rutas de acceso. Pueden utilizarse en la formación de los nombres de ficheros y/o rutas, de forma que se independice el programa que se construya del sistema operativo que se vaya a utilizar.

Volviendo con el ejemplo, tras la construcción del objeto `File`, se comprueba si realmente el fichero existe en el sistema de archivos mediante el método `exists()` (línea 5). A continuación se utilizan algunos métodos para obtener el nombre del

fichero, su directorio padre y la longitud del mismo (en bytes), respectivamente (líneas 8-10).

Asumiendo que el fichero */tmp/file.txt* del ejemplo existe y contiene la cadena “Hola”, la salida del programa sería la siguiente:

Salida Estándar
<pre>El fichero existe! getName(): file.txt getParent(): /tmp length(): 4</pre>

Nótese que la longitud del fichero (4 bytes) coincide con el número de caracteres de su contenido². Si el fichero no hubiera existido, las únicas dos diferencias significativas serían el diferente mensaje mostrado (por la salida estándar de error (línea 6)) y que el tamaño del fichero sería 0. Es importante destacar que el método *length()* tan solo obtiene el valor del tamaño para ficheros, no para directorios. Calcular el tamaño de un directorio, a partir de la suma de los tamaños de sus ficheros y sus directorios, requiere típicamente el uso de un método recursivo.

La tabla 16.1 incluye un resumen de los principales constructores y métodos disponibles en la clase *File*. No obstante, se recomienda al lector consultar el *API* de Java [Ora11c] para verificar la funcionalidad completa de las clases. Concretamente, la clase *File* también permite construir directorios, crear nuevos ficheros vacíos, distinguir entre un fichero y un directorio, etc.

<code>public File (String pathname)</code>	Crea un nuevo <i>File</i> a partir de la ruta a un fichero (o directorio).
<code>public boolean delete ()</code>	Intenta eliminar el fichero (o un directorio vacío). Devuelve true en caso de éxito.
<code>public boolean renameTo (File dest)</code>	Renombra el fichero al nuevo nombre especificado. Puede involucrar mover el fichero en el sistema de archivos. Devuelve true en caso de éxito.
<code>public boolean exists ()</code>	Devuelve true si el fichero existe en el sistema de archivos.

Tabla 16.1: Principales constructores y métodos de la clase *File*.

²En realidad el tamaño depende de la codificación utilizada para guardar el fichero. En este caso se utilizó una codificación ASCII que involucra 1 byte por carácter.

16.2 Ficheros de texto

Esta sección aborda el proceso de escritura y lectura de ficheros de texto. Como un extracto de código puede valer en ocasiones más que mil palabras, la explicación se realiza a través de ejemplos comentados. Existen diversas formas de procesar este tipo de ficheros; sin embargo, una forma muy sencilla para el programador de escribir en un fichero de texto es mediante la clase `PrintWriter`. Para lectura, es posible utilizar la clase `Scanner`.

16.2.1 Escritura en un fichero de texto

El proceso de escritura en un fichero de texto puede realizarse mediante la clase `PrintWriter`. Esta clase permite gestionar la escritura en un fichero de una manera muy similar a como se muestran los datos por la salida estándar, mediante los métodos `print`, `println` y `printf`; estando los dos primeros métodos sobrecargados para los diferentes tipos de datos primitivos y `String`.

La figura 16.3 muestra un ejemplo de uso de la clase `PrintWriter` para escribir un fichero de texto. Para ello, en primer lugar se importan las clases necesarias del paquete `java.io` (línea 1). Posteriormente, se construye un nuevo objeto de tipo `PrintWriter` invocando a su constructor con un `File` que refiera al fichero de salida (línea 6). En este caso concreto, como se ha especificado una ruta relativa, el fichero se creará en el directorio de trabajo (aquél desde el que se ejecute el programa) si no existía previamente. Si el fichero ya existía previamente, entonces se borra su contenido. A partir de este momento, el objeto `pw` permitirá ahora realizar escrituras en el fichero `file2.txt`.

```

1 import java.io.*;
2 public class TestPrintWriter {
3     public static void main(String[] args) {
4         String fichero = "file2.txt";
5         try {
6             PrintWriter pw = new PrintWriter(new File(fichero));
7             pw.print("El veloz murciélagó hindú");
8             pw.println(" comía feliz cardillo y kiwi");
9             pw.println(4.815162342);
10            pw.close();
11        } catch (FileNotFoundException e) {
12            System.err.println("Problemas al abrir el fichero");
13        }
14    }
15 }
```

Figura 16.3: Ejemplo de uso de `PrintWriter` para la escritura en ficheros de texto.

El constructor de la clase `PrintWriter` puede lanzar la excepción `FileNotFoundException` si el objeto `File` especificado no referencia a un fichero regular que pueda ser escrito o no puede crearse un nuevo fichero. Por ello, se utiliza un bloque `try-catch` para gestionar de forma apropiada la excepción.

La clase `PrintWriter` contiene los métodos `print` y `println` para escribir objetos de tipo `String` y cualquier tipo primitivo en Java en el fichero. Su comportamiento es análogo a los correspondientes métodos que se utilizan habitualmente en `System.out` para mostrar por pantalla salvo que, en este caso, escriben los resultados en el fichero (líneas 7-9). En el ejemplo, se están escribiendo dos `String` y un literal de tipo `double`.

Una vez finalizadas las operaciones de escritura, es necesario cerrar el fichero para asegurarse de que los datos realmente se han escrito y liberar los recursos asociados a la gestión del fichero. En realidad, si el programador olvida cerrar un fichero, Java se encarga de hacerlo. No obstante, resulta una buena práctica de programación cerrar el fichero una vez finalizado el proceso de escritura (y de lectura). Esto es especialmente necesario dado que los mecanismos de escritura en Java usan generalmente un *buffer*, o almacenamiento temporal intermedio, que permite agrupar los datos de varias escrituras en esta zona temporal hasta que hayan suficientes datos antes de escribirlos en el disco. Este proceso se realiza con el objetivo de hacer más eficiente el proceso de escritura, ya que el acceso al disco físico resulta bastante lento. Bajo este esquema, si no se cierra el fichero y el programa aborta por alguna razón, el fichero puede quedar incompleto.

Volviendo al ejemplo, el contenido del fichero, que puede ser visualizado desde cualquier editor de textos, es el siguiente:

```
file2.txt
El veloz murciélagó hindú comía feliz cardillo y kiwi
4.815162342
```

Si ya se dispone de un fichero de texto y se desea añadir nuevos datos al final, hay que modificar la forma de abrirlo, como se muestra a continuación:

```
PrintWriter pw = new PrintWriter(new FileOutputStream(fichero,true));
```

La tabla 16.2 incluye un resumen de los principales constructores y métodos disponibles en la clase `PrintWriter`. Los métodos de dicha clase no lanzan excepciones de E/S. El usuario puede comprobar si ha ocurrido algún error tras una operación de escritura invocando al método `checkError()`.

16.2.2 Lectura de un fichero de texto

La lectura de un fichero de texto puede llevarse a cabo utilizando la clase `Scanner`. En capítulos anteriores se utilizó esta clase para poder leer valores desde la entrada

<code>public PrintWriter (File f)</code>	Crea un nuevo <code>PrintWriter</code> a partir de un objeto de tipo <code>File</code> . Lanza <code>FileNotFoundException</code> si no se puede escribir en el fichero.
<code>public PrintWriter (OutputStream out)</code>	Crea un nuevo <code>PrintWriter</code> a partir de un flujo de salida.
<code>public void println (arg)</code>	Escribe un determinado argumento <code>arg</code> en el stream de salida y termina la línea. El argumento puede ser cualquier tipo primitivo o un <code>String</code> .
<code>public void print (arg)</code>	Mismo funcionamiento que <code>println</code> pero no termina la línea.
<code>public void printf (String sf, args)</code>	Escribe los argumentos <code>args</code> siguiendo la descripción de formato en <code>sf</code> .
<code>public boolean checkError ()</code>	Escribe todos los datos pendientes en el fichero. Si se ha producido algún error durante la escritura devuelve <code>true</code> .

Tabla 16.2: Principales constructores y métodos de la clase `PrintWriter`.

estándar. El mecanismo para leer valores de un fichero de texto resulta análogo. Básicamente, esta clase permite abstraer los datos del fichero como si fueran una secuencia de elementos, donde cada uno de ellos puede ser de un tipo diferente. Así, es posible ir leyendo del fichero de texto los elementos (tipos primitivos, `String` y líneas completas) de forma secuencial, uno a uno. Cada vez que se realiza una operación de lectura de un elemento, el objeto `Scanner` pasa de forma implícita al siguiente elemento de la secuencia, que será leído en la siguiente operación de lectura.

La figura 16.4 muestra un ejemplo de utilización de la clase `Scanner` para leer de un fichero de texto llamado `cosas.txt` cuyo contenido se muestra a continuación:

```
cosas.txt
1 2
3 4
Multiplicate por cero!
```

En ella se puede observar la creación de un objeto `Scanner` para poder leer del fichero de texto (línea 7). La invocación del constructor de la clase `Scanner` puede lanzar la excepción `FileNotFoundException` si el fichero especificado no existe. Por tanto, es necesario introducir un bloque `try-catch` (líneas 6-16) para especificar las acciones a ejecutar si se produce la excepción.

```

1 import java.io.*;
2 import java.util.Scanner;
3 public class TestScanner {
4     public static void main(String[] args) {
5         System.out.println("Leemos 3 números y una línea de texto");
6         try {
7             Scanner scanner = new Scanner(new File("cosas.txt"));
8             int n1 = scanner.nextInt();
9             int n2 = scanner.nextInt();
10            int n3 = scanner.nextInt();
11            scanner.nextLine();
12            String linea = scanner.nextLine();
13            System.out.println("Números: " + n1 + "," + n2 + "," + n3);
14            System.out.println("La línea es: " + linea);
15            scanner.close();
16        } catch (FileNotFoundException ex) {
17            System.err.println("El fichero no existe." + ex);
18        }
19    }
20 }
```

Figura 16.4: Ejemplo de uso de `Scanner` para leer de un fichero de texto.

A continuación se leen tres números enteros utilizando para ello `nextInt()` (líneas 8-10). La invocación al método `nextLine()` de la línea 11 provoca leer el resto de la línea y descartar los datos leídos (el número 4 y el retorno de carro de la línea). La instrucción de la línea 12 permite leer una línea completa. Luego, el programa muestra el resultado de las lecturas por la salida estándar (líneas 13 y 14). Finalmente, el programa cierra el fichero (línea 15).

Nótese que el programa está asumiendo la estructura del fichero y la disposición de sus datos antes de realizar las correspondientes operaciones de lectura. Si se invoca al método `nextInt()` y el valor que existe en el fichero no se trata de un `int`, entonces dicho método lanza la excepción `InputMismatchException`. Como esta excepción es subclase de `RuntimeException` no es necesario gestionarla de forma explícita. Por eso no resulta necesario capturar esa excepción en el código.

A continuación se muestra el contenido de la salida estándar generada por el programa:

Salida Estándar
Leemos 3 números y una línea de texto
Números: 1,2,3
La línea es: Multiplicate por cero!

La figura 16.5 muestra otro ejemplo de utilización de la clase `Scanner` para leer de un fichero de texto llamado `carreras.txt`. Este fichero incluye el nombre de un corredor y la posición en la que ha finalizado una supuesta carrera, con respecto al resto de corredores. A continuación se muestra un extracto del mismo:

carreras.txt

Lucía 7
Enrique 4
María 3

```

1 import java.io.*;
2 import java.util.Scanner;
3 public class TestScannerWhile {
4     public static void main(String[] args) {
5         try {
6             Scanner scanner = new Scanner(new File("carreras.txt"));
7             while (scanner.hasNextLine()) {
8                 String linea = scanner.nextLine();
9                 String[] tokens = linea.split(" ");
10                System.out.println(tokens[0] + " : " + tokens[1]);
11            }
12        } catch (FileNotFoundException ex) {
13            System.err.println("El fichero no existe." + ex);
14        }
15    }
16 }
```

Figura 16.5: Ejemplo de uso de `Scanner` para leer de un fichero de texto con detección de terminación de fichero.

En ella se puede observar la creación de un objeto `Scanner` para poder leer del fichero de texto (línea 6). Se observa el uso de un bucle cuya guarda utiliza el método `hasNextLine()` del objeto `Scanner`. Esto permite el progreso del bucle hasta que no queden más líneas en el fichero de texto por leer. A continuación se lee cada una de las líneas mediante el método `nextLine()` (línea 8). Cada línea se divide mediante el método `split(String)` de la clase `String`, usando como separador el espacio. Esto permite obtener todos los tokens (las palabras individuales) de la línea (línea 9) en un array. Finalmente, se muestran los valores obtenidos (nombre del corredor y posición) por la salida estándar (línea 10).

La tabla 16.3 incluye un resumen de los principales constructores y métodos disponibles en la clase `Scanner`. Se muestran también algunas de las principales excepciones que pueden lanzar dichos métodos.

<code>public Scanner (File f)</code>	Crea un nuevo <code>Scanner</code> a partir de un objeto de tipo <code>File</code> . Lanza <code>FileNotFoundException</code> si el fichero no es accesible.
<code>public Scanner (InputStream source)</code>	Crea un nuevo <code>Scanner</code> a partir de un flujo de entrada (véase sección 16.4).
<code>public String next ()</code>	Obtiene el siguiente elemento leído como un <code>String</code> . Lanza <code>NoSuchElementException</code> si no quedan más elementos por leer. Lanza <code>IllegalStateException</code> si el flujo del <code>Scanner</code> estaba cerrado.
<code>public String nextLine ()</code>	Se lee el resto de línea completa, descartando el salto de línea. Devuelve el resultado como un <code>String</code> . Lanza <code>NoSuchElementException</code> si no quedan más elementos por leer. Lanza <code>IllegalStateException</code> si el flujo del <code>Scanner</code> estaba cerrado.
<code>public int nextInt ()</code> <code>public long nextLong ()</code> <code>public short nextShort ()</code> <code>public byte nextByte ()</code> <code>public float nextFloat ()</code> <code>public double nextDouble ()</code> <code>public boolean nextBoolean ()</code>	Devuelve el siguiente elemento como un <code>int</code> siempre que se trate de un <code>int</code> . Ídem para <code>long</code> , <code>short</code> , <code>byte</code> , <code>float</code> , <code>double</code> y <code>boolean</code> . Lanza <code>InputMismatchException</code> en caso de no poder obtener un valor del tipo apropiado. Lanza <code>NoSuchElementException</code> si no quedan más elementos por leer. Lanza <code>IllegalStateException</code> si el flujo del <code>Scanner</code> estaba cerrado.
<code>public boolean hasNextInt ()</code> <code>public boolean hasNextLong ()</code> <code>public boolean hasNextShort ()</code> <code>public boolean hasNextByte ()</code> <code>public boolean hasNextFloat ()</code> <code>public boolean hasNextDouble ()</code> <code>public boolean hasNextBoolean ()</code> <code>public boolean hasNext ()</code> <code>public boolean hasNextLine ()</code>	Devuelve <code>true</code> si el siguiente elemento a obtener se puede interpretar como un <code>int</code> . Ídem para <code>long</code> , <code>short</code> , <code>byte</code> , <code>float</code> , <code>double</code> y <code>boolean</code> . También para detectar si existe un <code>String</code> o una línea de texto. Lanza <code>IllegalStateException</code> si el flujo del <code>Scanner</code> estaba cerrado.
<code>public Scanner useLocale (Locale l)</code>	Establece la configuración local del <code>Scanner</code> a la configuración especificada por el <code>Locale l</code> .
<code>public Scanner useDelimiter (String p)</code>	Establece el conjunto de delimitadores del <code>Scanner</code> a un patrón construido a partir del <code>String p</code> .
<code>public void close ()</code>	Cierra el <code>Scanner</code> .

Tabla 16.3: Principales constructores y métodos de la clase `Scanner`.

16.3 Ficheros binarios

Esta sección aborda el proceso de escritura y lectura de ficheros binarios, a través de ejemplos comentados. La forma más cómoda para el programador de escribir en un fichero binario es mediante la clase `DataOutputStream`. Para lectura, es posible utilizar la clase `DataInputStream`³.

16.3.1 Escritura en un fichero binario

La escritura se realizará mediante la clase `DataOutputStream` que permite la escritura de tipos primitivos y `String` en un fichero binario.

La figura 16.6 muestra un ejemplo de escritura en un fichero binario de tres valores de diferentes tipos (un `String`, un `int` y un `double`). Por lo tanto, un fichero binario no tiene porqué ser homogéneo sino que puede almacenar diferentes tipos de datos. En primer lugar, se procede a la construcción del objeto `DataOutputStream` (líneas 8 y 9). A continuación, se utilizan los métodos correspondientes para guardar los diferentes valores, dependiendo del tipo de datos: `writeUTF`, para escribir un `String`; `.writeInt` para escribir un entero y `writeDouble` para escribir un valor decimal de doble precisión.

Finalizadas las escrituras, resulta conveniente cerrar el fichero para liberar los recursos asociados y garantizar la escritura de los datos en disco. El fichero generado en disco, denominado *calificaciones.dat*, contiene la secuencia de bytes que representan los valores almacenados. Por lo tanto, no se trata de un fichero visualizable ni editable directamente. Debe ser leído desde código Java.

Las líneas 14-18 del ejemplo se centran en la posterior lectura de los datos escritos en el fichero para verificar su correcta funcionalidad. Nótese el paralelismo entre la nomenclatura de los métodos utilizados para escribir en un fichero y los usados para leer del mismo. No obstante, en la siguiente sección se aborda y describe de forma individualizada el proceso de lectura de un fichero binario.

La salida estándar generada por el programa se muestra a continuación:

Salida Estándar	
Nombre leído: IIP	
Convocatoria leída: 1	
Nota leída: 7.8	

³En realidad también se podrían utilizar las clases `ObjectInputStream` y `ObjectOutputStream`, con idéntica semántica y funcionalidad y que, además, permiten la escritura de objetos completos en ficheros, tal y como se verá en la sección 16.5.

```

1 import java.io.*;
2 public class Calificaciones {
3     public static void main(String[] args) {
4         String fichero = "calificaciones.dat";
5         String nombre = "IIP";
6         int conv = 1; double nota = 7.8;
7         try {
8             DataOutputStream out =
9                 new DataOutputStream(new FileOutputStream(fichero));
10            out.writeUTF(nombre);
11            out.writeInt(conv);
12            out.writeDouble(nota);
13            out.close();
14            DataInputStream in =
15                new DataInputStream(new FileInputStream(fichero));
16            System.out.println("Nombre leído: " + in.readUTF());
17            System.out.println("Convocatoria leída: " + in.readInt());
18            System.out.println("Nota leída: " + in.readDouble());
19            in.close();
20        } catch (IOException e) {
21            System.err.println("Problemas con el fichero.");
22            e.printStackTrace();
23        }
24    }
25 }
```

Figura 16.6: Ejemplo de escritura y posterior lectura de un fichero binario.

16.3.2 Lectura de un fichero binario

Para exemplificar la lectura de un fichero binario, la figura 16.7 muestra un código que permite leer de un fichero binario que incluye un conjunto de medidas de precipitación para una ciudad concreta a lo largo de un mes. Como no todos los días llueve, tan solo se incluyen datos para los días que ha llovido. Cada dato es el número de litros por metro cuadrado registrados en dicha ciudad para ese día concreto. Por lo tanto, para cada ciudad hay un número variable de datos. El fichero tiene la siguiente estructura:

lluvias.dat
Nombre_Ciudad N Dato_0 Dato_1 ... Dato_N-1

Se observa que en primer lugar aparece el nombre de la ciudad, a continuación el número de datos de precipitación (N) y, luego, los N datos de precipitación. Recuérdese que, al tratarse de un fichero binario, no es posible visualizar los datos del mismo con un editor de ficheros, sino que debe leerse desde un programa Java.

```

1 import java.io.*;
2 public class TestDataInputStream {
3     public static void main(String[] args) {
4         String fichero = "lluvias.dat";
5         String ciudad;
6         int nDatos;
7         float[] lluvias;
8         try {
9             DataInputStream in =
10                 new DataInputStream(new FileInputStream(fichero));
11             ciudad = in.readUTF();
12             nDatos = in.readInt();
13             lluvias = new float[nDatos];
14             for (int i=0; i<nDatos; i++)
15                 lluvias[i] = in.readFloat();
16             in.close();
17             System.out.println("Ciudad: " + ciudad);
18             System.out.println("Nº de datos: " + nDatos);
19             System.out.println("Primer dato: " + lluvias[0]);
20         } catch (IOException ex) {
21             System.err.println("Problemas al leer: " + ex);
22         }
23     }
24 }
```

Figura 16.7: Ejemplo de lectura de un fichero binario.

En el ejemplo, en primer lugar se crean las variables que almacenarán los datos leídos del fichero (líneas 5-7). El array `lluvias` tan solo se declara, pero no se inicializa, puesto que hasta que no se lee el fichero se desconoce el número de datos que deberá almacenar. A continuación, se construye el objeto de tipo `DataInputStream` que permite leer los datos del fichero binario. Posteriormente, se lee el nombre de la ciudad y el número de datos de precipitación que vendrán a continuación (líneas 11 y 12). En ese preciso momento ya es posible construir el array para ajustarlo al número de datos que se leerán. En este sentido, comienza un bucle de lectura de los valores de precipitación (líneas 14 y 15) que permite leer los datos del fichero y almacenarlos en el array. Finalizado el proceso de lectura, se procede a cerrar el fichero (línea 16). Por último, se muestran algunos datos leídos por la salida estándar (líneas 17-19).

La invocación del constructor de `FileInputStream` puede lanzar la excepción `FileNotFoundException` si ocurre algún problema tratando de abrir el fichero. Además, los métodos de lectura de tipos pueden lanzar la excepción `IOException` en caso de problemas al realizar la operación. Como `FileNotFoundException` es subclase de `IOException`, es posible realizar la gestión de excepciones que aparece

en las líneas 8-20. No obstante, el programador podría decidir realizar un tratamiento diferenciado de ambas excepciones para distinguir entre el error provocado por no encontrar un fichero y el resultante de un error al realizar una operación de E/S, por ejemplo, derivado de un fallo en el sistema de almacenamiento.

16.3.3 Ficheros binarios de acceso aleatorio

Las secciones anteriores se han centrado en ficheros de acceso secuencial, donde los datos deben ser leídos en el mismo orden en el que fueron escritos. Sin embargo, existen aplicaciones que pueden beneficiarse de acceder a cualquier punto del fichero sin necesidad de haber leído previamente todos los datos anteriores. Volviendo al ejemplo que se mostró en la figura 16.1, si se conoce el punto exacto del registro al que se pretende acceder en el fichero es mucho más eficiente leer únicamente ese registro sin necesidad de tener que procesar todos los registros anteriores. Esto podría ser posible si se conociese el tamaño de cada uno de los campos, lo que permitiría saber el tamaño de cada registro y, por lo tanto, la posición dentro del fichero de cada registro.

Java dispone de la clase `RandomAccessFile` que permite el acceso aleatorio a un fichero binario que representa un conjunto de bytes. Se dispone de un puntero con una granularidad de 1 byte que el programador puede mover por el fichero para realizar operaciones de lectura y escritura en cualquier punto del fichero. Para ello, es necesario que los datos guardados en el fichero tengan un tamaño coherente con su tipo de datos. Por ejemplo, un `int` se almacena con un tamaño de 4 bytes, un `double` requiere 8 bytes y un `boolean` se guarda como 1 byte. En el caso especial de los `String`, se escriben primero dos bytes que indican el número de bytes que vienen a continuación y que representan el `String`. Es importante saber que cada operación de lectura y escritura provoca el avance del puntero del fichero tantos bytes como se hayan leído o escrito. En este sentido, conocer el tamaño exacto de cada campo en el fichero facilita calcular el desplazamiento necesario dentro del fichero para acceder a la posición de un determinado dato.

La figura 16.8 muestra un ejemplo de uso de ficheros de acceso aleatorio en Java. En primer lugar, se procede a la creación del fichero indicando que se va a utilizar tanto para lectura como para escritura (`rw = read / write`) (línea 5). A continuación, se escriben dos valores enteros, que ocupan 4 bytes cada uno en el fichero (líneas 7 y 8) y se consulta la longitud del fichero, que será 8 bytes, así como la posición del puntero, que estará en el 8º byte (líneas 9 y 10). En este momento, se desplaza el puntero al 4º byte, que representa el comienzo del almacenamiento del número 89. En efecto, los bytes 0, 1, 2 y 3 se emplean para almacenar el entero 65, por lo que el 4º byte es el comienzo del número 89 (línea 12). En ese momento, se procede a leer un entero, que será el número 89 (línea 13). A continuación se sobrescribe el valor 89 por el 77 (líneas 16 y 17) y se verifica la lectura de ese nuevo entero (líneas 18-21). Finalmente, se lleva el puntero al final del fichero y se escribe un

```
1 import java.io.*;
2 public class TestRandomAccessFile {
3     public static void main(String[] args) {
4         try {
5             RandomAccessFile raf = new RandomAccessFile("data", "rw");
6             System.out.println("Se escriben dos enteros (65 y 89):");
7             raf.writeInt(65);
8             raf.writeInt(89);
9             System.out.println("Longitud: " + raf.length());
10            System.out.println("Puntero: " + raf.getFilePointer());
11            System.out.println("Moviendo el puntero al 4º byte");
12            raf.seek(4);
13            int a = raf.readInt();
14            System.out.println("Entero leído: " + a);
15            System.out.println("Machacando el entero por 77");
16            raf.seek(4);
17            raf.writeInt(77);
18            System.out.println("Verificando el valor sobreescrito");
19            raf.seek(4);
20            int b = raf.readInt();
21            System.out.println("Entero leído: " + b);
22            System.out.println("Llevando el puntero al final");
23            raf.seek(raf.length());
24            raf.writeDouble(178.54);
25            System.out.println("Releyendo el valor escrito");
26            raf.seek(raf.getFilePointer() - 8);
27            double c = raf.readDouble();
28            System.out.println("Valor leído: " + c);
29            raf.close();
30        } catch (IOException ex) {
31            System.err.println("Problemas durante la E/S" + ex);
32        }
33    }
34 }
```

Figura 16.8: Ejemplo de lectura y escritura en fichero de acceso aleatorio.

valor en doble precisión (`double`) que requiere 8 bytes de almacenamiento (líneas 22-24). Por ello, para releer el valor no hay más que atrasar el puntero 8 bytes y realizar una operación de lectura de un `double` (líneas 25-28).

La ejecución del programa muestra el siguiente resultado por la salida estándar:

Salida Estándar
Se escriben dos enteros (65 y 89): Longitud: 16 Puntero: 8 Moviendo el puntero al 4º byte Entero leído: 89 Machacando el entero por 77 Verificando el valor sobrescrito Entero leído: 77 Llevando el puntero al final Releyendo el valor escrito Valor leído: 178.54

Nótese que el constructor de la clase `RandomAccessFile` puede lanzar la excepción `FileNotFoundException` mientras que los métodos pueden lanzar `IOException`. Como la primera es subclase de la segunda, se utiliza un bloque `try-catch` general de gestión de excepciones.

Como siempre, se recomienda al lector consultar el *API* de Java [Ora11c] para obtener más información sobre la clase `RandomAccessFile`.

16.4 Flujos

En los capítulos anteriores se han mostrado las formas más convenientes para acceder a ficheros tanto de texto como binarios. No obstante, la E/S en Java permite funcionalidades mucho más allá de leer y escribir en ficheros. Como se comentó al inicio del capítulo, la E/S está basada en flujos, que no son más que una secuencia de bytes que parten de un origen y se dirigen a un destino. Los flujos que se originan en el programa se llaman *flujos de salida* mientras que los que sirven como entrada de datos al programa se llaman *flujos de entrada*. En Java existen dos grandes categorías de flujos:

- *Flujos de bytes*. Permiten manejar de forma eficiente la E/S de bytes. Se usan generalmente al leer o escribir datos binarios.
- *Flujos de caracteres*. Permiten gestionar la E/S de caracteres. Se usan al leer o escribir datos de texto. Se utiliza *Unicode* como esquema de codificación, soportando así la diversidad de caracteres de diferentes lenguas.

Cada categoría de flujos supone una jerarquía diferente de clases. Por lo tanto, el número de clases involucradas en la E/S en Java es bastante elevado. Cabe destacar que, en el nivel más bajo, la E/S está orientada a bytes. Los flujos de caracteres tan solo proporcionan una capa de abstracción por encima para poder gestionar los caracteres.

16.4.1 Flujos de bytes

Los flujos de bytes están representados por dos clases abstractas `InputStream` y `OutputStream`, encargadas de los flujos de entrada y de salida, respectivamente.

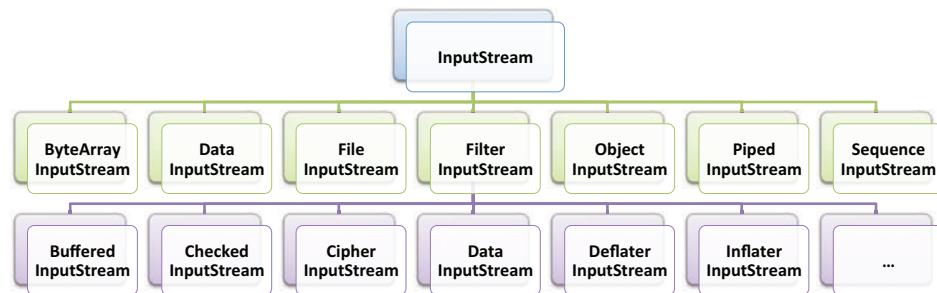


Figura 16.9: Jerarquía de clases a partir de `InputStream` (flujos binarios).

La figura 16.9 muestra gran parte de la jerarquía de clases de `InputStream`, que permite la definición y uso de flujos de entrada al programa. Por ejemplo, la clase `FileInputStream` permite definir un flujo de entrada de bytes para leer desde un fichero binario. No obstante, a partir de ese flujo tan solo es posible leer un conjunto de bytes, no es posible interpretar directamente los valores como los correspondientes tipos que puedan estar representando ese grupo de bytes. Por ello existen determinadas subclases que permiten procesar esa información y manipular los tipos de datos que realmente están representando esos grupos de bytes. Este es el caso de la clase `DataInputStream`, utilizada anteriormente, que permite la lectura de datos de distintos tipos a través de un flujo de bytes.

Es posible realizar funcionalidad avanzada mediante estas clases, como la lectura de un flujo de bytes procedentes de una entrada cifrada cuyos datos sean descifrados al mismo tiempo que se leen (`CipherInputStream`). También es posible leer desde un flujo de entrada de bytes procedentes de una fuente de datos comprimida (`DeflaterInputStream`). Se recomienda al lector acudir al *API* de Java [Ora11c] para investigar sobre el funcionamiento del resto de clases. La jerarquía de clases para `OutputStream` es prácticamente idéntica a la mostrada, salvo que todas las clases acaban con el sufijo `OutputStream`.

Para exemplificar la E/S sobre un elemento que no sea un fichero, la figura 16.10 muestra un ejemplo de lectura a partir de una *URL*. El ejemplo permite leer de una conexión al servidor disponible en <http://www.google.com>. En primer lugar se construye el objeto URL (línea 7), y se abre una conexión con el servidor (línea 8). Posteriormente, se obtiene un *InputStream* para poder leer desde la conexión con el servidor. Ese *InputStream* se pasa como argumento a un objeto *Scanner* para poder ir leyendo línea a línea (línea 9). El bucle de las líneas 10-13 permite ir recuperando las líneas y mostrarlas por pantalla.

```

1 import java.net.*;
2 import java.io.IOException;
3 import java.util.Scanner;
4 public class TestURL {
5     public static void main(String[] args) {
6         try {
7             URL url = new URL("http://www.google.com");
8             URLConnection con = url.openConnection();
9             Scanner sc = new Scanner(con.getInputStream());
10            while (sc.hasNextLine()) {
11                String l = sc.nextLine();
12                System.out.println(l);
13            }
14        } catch (IOException ex) {
15            System.err.println("Error: " + ex);
16        }
17    }
18 }
```

Figura 16.10: Ejemplo de lectura a partir de una URL.

A continuación se muestra un extracto del resultado obtenido por la salida estándar. Se trata del código HTML recuperado por la conexión al servidor.

Salida Estándar

```
<!doctype html><html><head><meta http-equiv="content-type"
content="text/html; charset=ISO-8859-1"><title>Google</title>
...
```

16.4.2 Flujos de caracteres

Los flujos de caracteres en Java están representados por las clases abstractas `Reader` y `Writer`. Ambos operan sobre flujos de caracteres codificados en *Unicode*, soportando así los múltiples caracteres de lenguas internacionales.

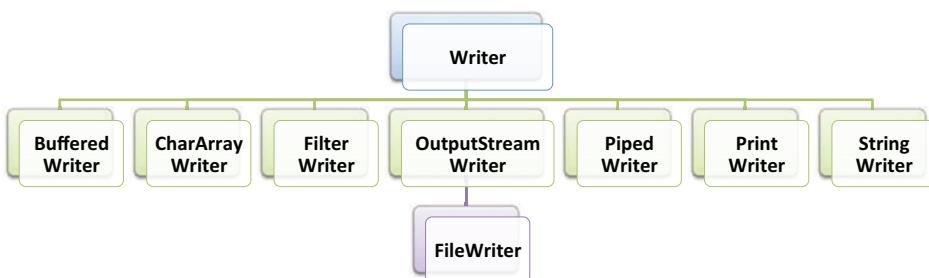


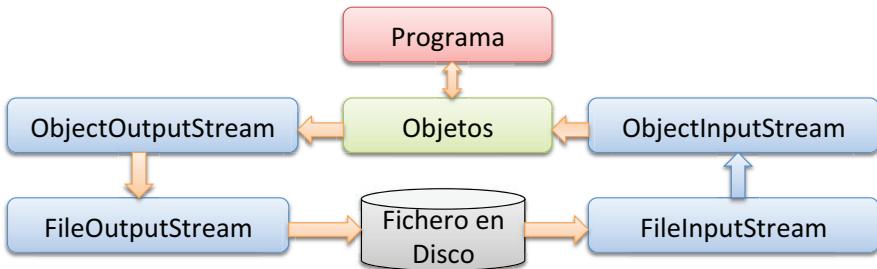
Figura 16.11: Jerarquía de clases a partir de la clase `Writer` (flujos de caracteres).

La figura 16.11 muestra la jerarquía de clases que heredan de `Writer`. Aquí aparece la clase `PrintWriter` que ha sido utilizada en las secciones anteriores para escritura en ficheros de texto. Esta jerarquía de clases proporciona funcionalidades adicionales. Por ejemplo, la clase `PipedWriter` permite comunicar dos procesos (o dos hilos de ejecución) a través de una tubería para que comparten datos basados en texto entre ellos. Se invita al lector a que consulte el *API* de Java [Ora11c] para conocer la funcionalidad del resto de clases.

16.5 E/S de objetos

En las secciones anteriores se ha abordado el proceso de lectura y escritura de tipos primitivos y `String`. No obstante, en un paradigma de orientación a objetos como es el que ofrece Java, los programadores trabajan con objetos que tienen un determinado estado representado por los valores de los atributos. Estos atributos pueden ser tanto tipos primitivos y `String` como referencias a otros objetos.

Java permite almacenar objetos completos (todos sus atributos, incluidas las referencias a otros objetos) en un flujo de salida, que puede ser almacenado en un fichero. Esto permite almacenar grupos de objetos que están en memoria en un archivo. Posteriormente, es posible leer dicho fichero para volver a obtener en memoria todos esos objetos. De esta manera, un grupo de objetos puede sobrevivir a la ejecución de un programa. Todo este proceso, que a priori es complejo, se gestiona en Java de una manera muy sencilla para el programador. El proceso anterior está descrito en la figura 16.12, que menciona las clases necesarias para llevarlo a cabo.

**Figura 16.12:** E/S de objetos en ficheros binarios.

Las clases cuyos objetos se quieren guardar en el disco deben implementar la interfaz `Serializable`. Se trata de una interfaz que no contiene métodos, por lo que tan solo es una forma de etiquetar a aquellas clases cuyos objetos pueden estar involucrados en operaciones de E/S. Por ejemplo, la figura 16.13 muestra la clase `Persona`, que define tres atributos de diferentes tipos para representar el nombre, la edad y la altura (admite decimales para mayor precisión) y que implementa la interfaz `Serializable`.

```

1 import java.io.Serializable;
2 public class Persona implements Serializable {
3     String nombre;
4     int edad;
5     double altura;
6
7     public Persona(String n, int e, double a) {
8         this.nombre = n; this.edad = e; this.altura = a;
9     }
10
11    public String toString() {
12        return nombre + ", " + edad + ", " + altura;
13    }
14 }
```

Figura 16.13: Ejemplo de clase que implementa la interfaz `Serializable`.

Para poder almacenar los objetos en disco se utiliza la clase estándar `ObjectOutputStream` que permite convertir los objetos a un flujo de bytes que es escrito mediante un `FileOutputStream`. Análogamente, la lectura de objetos se hace a partir de un `FileInputStream` encargado de leer el flujo de bytes que son posteriormente utilizados para la creación del objeto en memoria a partir del `ObjectInputStream`. Nótese que el uso de flujos permitiría canalizar la escritura de objetos hacia un flujo de salida que conectase con otro proceso, lo que permitiría

ría transferir objetos de un proceso a otro, incluso en diferentes máquinas. Por lo tanto, este proceso no se restringe exclusivamente a ficheros.

La figura 16.14 muestra un ejemplo donde se realiza la E/S de un objeto de tipo **Persona** (aunque podrían haber sido múltiples objetos, incluso de diferentes tipos).

```

1 import java.io.*;
2 public class TestObjectIO {
3     public static void main(String[] args) {
4         String filename = "obj.data";
5         Persona p = new Persona("Luisa Garcia", 25, 179.45);
6         try {
7             ObjectOutputStream ous =
8                 new ObjectOutputStream(new FileOutputStream(filename));
9             ous.writeObject(p);
10            ous.close();
11        } catch (IOException ex1) {
12            System.err.println("Error al escribir: " + ex1);
13            return;
14        }
15        try {
16            ObjectInputStream ois =
17                new ObjectInputStream(new FileInputStream(filename));
18            Persona p2 = (Persona) ois.readObject();
19            ois.close();
20            System.out.println("Persona leida: " + p2.toString());
21        } catch (IOException ex2) {
22            System.err.println("Error al leer: " + ex2);
23        } catch (ClassNotFoundException ex3) {
24            System.err.println("Clase no encontrada: " + ex3);
25        }
26    }
27 }
```

Figura 16.14: Ejemplo de uso de E/S de objetos.

En primer lugar se procede a la creación del flujo de salida de objetos mediante la clase **ObjectOutputStream** (líneas 7 y 8). A continuación, se procede a la escritura del objeto utilizando el método **writeObject** (línea 9). Finalmente, se procede a cerrar el flujo de salida, como es habitual. Por lo tanto, la escritura de objetos en ficheros es muy transparente en Java. Además, es posible almacenar clases cuyos atributos sean referencias a otras clases definidas por el usuario. En este caso, se procesan de forma recursiva todos los objetos, almacenando todos los objetos involucrados. Para ello, todas las clases involucradas deben implementar la interfaz **Serializable**.

Un ejemplo algo más elaborado, en el que también se utiliza la E/S de objetos, puede verse en las figuras 16.15, 16.16 y 16.17 en las que se presenta de forma abreviada las clases `ItemAgenda`, para representar información básica de una entrada individual de una agenda, `Agenda`, que modela una lista de tales entradas básicas (gestionada internamente mediante un array de dimensión ampliable) y la clase `GestorPrueba`, que permite interactuar de forma elemental con las primeras, mediante la creación, almacenamiento y recuperación de una `Agenda` con algunos elementos.

```

1 import java.io.*;
2 public class ItemAgenda implements Serializable {
3     private String nom, tel;
4     private int postal;
5
6     public ItemAgenda(String n, String t, int p) {
7         nom = n; tel = t; postal = p;
8     }
9
10    public String toString() {
11        return nom + ":" + tel + "(" + postal + ")";
12    }
13
14    // Otros métodos de la clase ItemAgenda ...
15 }
```

Figura 16.15: La clase `ItemAgenda` abreviada.

Como se ve, un objeto `Agenda` puede contener varios `ItemAgenda`. Como la política de almacenamiento consiste en guardar por completo un objeto `Agenda`, para poder recuperarlo posteriormente, es necesario explicitar que ambas clases implementan el interface `Serializable`.

La gestión del almacenamiento y recuperación se ha definido en la propia clase `Agenda`, cuyos objetos deben ser almacenados y/o recuperados (puede verse en la figura 16.16). Para el almacenamiento se ha definido el método de objeto `guardarAgenda(String)` (línea 26), mientras que para recuperarla se ha definido el método de clase `leerAgenda(String)` (línea 35).

Cuando se ejecuta el `main` de la clase `GestorPrueba`, se construye una `Agenda`, a partir de algunos `ItemAgenda` (líneas 4 a 10, en la figura 16.17) para, a continuación, almacenarla en el fichero de objetos `agenda1.dat` del que se lee y recupera posteriormente.

```

1 import java.io.*;
2 public class Agenda implements Serializable {
3     public static final int MAX = 8;
4     private ItemAgenda[] elArray;
5     private int num;
6
7     public Agenda() { elArray = new ItemAgenda[MAX]; num = 0; }
8
9     public void insertar(ItemAgenda b) {
10         if (num>=elArray.length) duplicaEspacio();
11         elArray[num++]=b; }
12
13    public String toString() {
14        String res = "";
15        for (int i=0; i<num; i++) res += elArray[i] + "\n";
16        res += "=====";
17        return res; }
18
19    private void duplicaEspacio() {
20        ItemAgenda[] aux = new ItemAgenda[2*elArray.length];
21        for (int i=0; i<elArray.length; i++) aux[i] = elArray[i];
22        elArray = aux; }
23
24 // Otros métodos de la clase Agenda ...
25
26    public void guardarAgenda(String fichero) {
27        try {
28            ObjectOutputStream oos = new ObjectOutputStream(
29                            new FileOutputStream(new File(fichero)));
30            oos.writeObject(this); oos.close();
31        } catch (IOException fex) {
32            System.err.println("Error al guardar: " + fex.getMessage()); }
33    }
34
35    public static Agenda leerAgenda(String fichero) {
36        Agenda aux = null;
37        try {
38            ObjectInputStream ois = new ObjectInputStream(
39                            new FileInputStream(fichero));
40            aux = (Agenda)ois.readObject(); ois.close();
41        } catch (IOException ex) {
42            System.err.println("Error al recuperar: " + ex.getMessage());
43        } catch (ClassNotFoundException ex) {
44            System.err.println("Clase no coincidente:" + ex.getMessage()); }
45        return aux;
46    }
47}

```

Figura 16.16: La clase Agenda abreviada.

```

1 import java.io.*;
2 public class GestorPrueba {
3     public static void main(String[] args) {
4         ItemAgenda i1 = new ItemAgenda("Enrique Perez", "622115611", 46022);
5         ItemAgenda i2 = new ItemAgenda("Rosalía", "963221153", 46010);
6         ItemAgenda i3 = new ItemAgenda("Juan Duato", "913651228", 18011);
7
8         // Creación de la Agenda a1 ...
9         Agenda a1 = new Agenda();
10        a1.insertar(i1); a1.insertar(i2); a1.insertar(i3);
11
12         // Escribir en el fichero y mostrar:
13         a1.guardarAgenda("agenda1.dat");
14         System.out.println("AGENDA ALMACENADA:");
15         System.out.println(a1);
16
17         // Leer del fichero y mostrar ...
18         Agenda rec = Agenda.leerAgenda("agenda1.dat");
19         System.out.println("AGENDA RECUPERADA:");
20         System.out.println(rec);
21     }
22 }
```

Figura 16.17: Clase GestorPrueba. Creación, almacenamiento y recuperación de una agenda.

En su ejecución el programa muestra, escribiéndola, la Agenda almacenada y recuperada:

<pre> AGENDA ALMACENADA: Enrique Perez: 622115611 (46022) Rosalía: 963221153 (46010) Juan Duato: 913651228 (18011) =====</pre>	Salida Estándar
<pre> AGENDA RECUPERADA: Enrique Perez: 622115611 (46022) Rosalía: 963221153 (46010) Juan Duato: 913651228 (18011) =====</pre>	

Si a continuación se lista el contenido del directorio en curso, se tiene:

<pre> \$ ls -l *.dat -rw-r--r-- 1 profesor PRG 276 2012-04-05 18:00 agenda1.dat</pre>	Sistema
---	----------------

Obsérvese que el fichero que contiene el objeto, *agenda1.dat*, ocupa 276 bytes.

16.6 Excepción `EOFException`. Determinación del final de un fichero binario

Se ha estudiado ya, en algún programa anterior, la lectura de una secuencia de datos almacenada en un fichero binario. Así, por ejemplo, en el programa visto en la sección 16.3.2, se recupera una secuencia de valores de un fichero de objetos gracias a que se conoce inicialmente el número de elementos que contiene el propio fichero.

Puede ocurrir, sin embargo, que el número de elementos del fichero no sea conocido inicialmente. Se plantea entonces la cuestión de cuándo interrumpir la lectura del fichero sin tratar de acceder más allá del final del mismo o de, al menos, recuperar adecuadamente el programa en caso de sobrepasar dicho límite.

En general, cuando en un flujo o fichero binario, bien de datos elementales (como `FileInputStream`), bien de objetos (como `ObjectInputStream`) se intente acceder más allá del final del mismo se provocará una `IOException`.

Adicionalmente, si las operaciones de lectura son de alguno de los tipos de datos elementales (tal como `readInt()`, `readDouble()`, `readBoolean()`, etc.) de las clases `FileInputStream` o `ObjectInputStream`, entonces, en el caso de intentar acceder más allá del final del fichero se producirá una excepción `EOFException`, que es subclase de `IOException`.

Gestionando adecuadamente la excepción correspondiente, es posible determinar si se ha llegado o no al final del fichero y, con ello, acabar el tratamiento. En la figura 16.18 se muestra, a título de ejemplo, una clase Java que, tras escribir un número aleatorio de valores aleatorios en un fichero, mediante un `ObjectOutputStream`, vuelve a leerlos, mostrando los valores almacenados en la pantalla, tras abrir el fichero mediante un `ObjectInputStream`.

Como puede verse (líneas 28 a 38 de la figura 16.18) la lectura se ha organizado mediante un bucle cuya terminación vendrá dada por el acceso al final del fichero, que provoca la `EOFException` que, correspondientemente, es tratada.

Nótese, líneas 27 a 41, que se ha anidado un bloque `try ... catch` dentro de otro. Mediante el más interno se gestiona el posible fin del fichero, cuya lectura se hace mediante un bucle finalizado exclusivamente por la aparición de la `EOFException`. Con el bloque más externo es posible controlar condiciones problemáticas de entrada, tales como la inexistencia del fichero, etc.

```
1 import java.io.*;
2 class MultiplesElementos {
3
4     public static void main(String[] args) {
5         String fichero = "ejemplo.dat";
6         escribir(fichero);
7         leer(fichero);
8     }
9
10    public static void escribir(String fich) {
11        try {
12            ObjectOutputStream oos = new ObjectOutputStream(
13                            new FileOutputStream(fich));
14            int alea = (int)(Math.random()*10+5);
15            for (int i=1; i<=alea; i++) {
16                int val = (int)(Math.random()*10);
17                oos.writeInt(val); System.out.print(val + " ");
18            }
19            System.out.println("\nFinal de escritura");
20            oos.close();
21        } catch (IOException fex) {
22            System.err.println("Error al guardar: " + fex.getMessage());
23        }
24    }
25
26    public static void leer(String fich) {
27        try {
28            ObjectInputStream ois = new ObjectInputStream(
29                            new FileInputStream(fich));
30            try {
31                while (true) {
32                    int val = ois.readInt();
33                    System.out.print(val + " ");
34                }
35            } catch (EOFException ef) {
36                System.out.println("\nFinal del fichero");
37            }
38            ois.close();
39        } catch (IOException fex) {
40            System.err.println("Error al guardar: " + fex.getMessage());
41        }
42    }
43}
```

Figura 16.18: Ejemplo de tratamiento de fin de fichero (*EOFException*).

Si se ejecuta el código de la clase, se provocará una salida similar a la siguiente:

Salida Estándar										
4	7	8	6	7	3	7	0	2	5	8
Final de escritura										
4	7	8	6	7	3	7	0	2	5	8
Final del fichero										

16.7 Problemas propuestos

1. Construir un programa Java que reciba como argumento de línea de comandos la ruta a un fichero y que muestre por pantalla información básica sobre el mismo (como mínimo el nombre del fichero, directorio donde se encuentra y su tamaño expresado en kbytes).
2. Escribir un método estático que escriba en un fichero binario los números del 1 al 999.
3. Escribir un método estático que lea el fichero generado por el programa del ejercicio 2 y sume dichos números. Comprobar que el resultado es correcto implementando un bucle adicional que realice dicha suma.
4. Construir un programa que escriba en un fichero de texto los números del 1 al 999 y posteriormente los vuelva a leer de ese fichero para realizar la suma de los mismos. Verificar que el resultado es correcto. Comprobar la diferencia de tamaños entre el fichero generado en el ejercicio 2 y el generado por este ejercicio.
5. Construir un programa que permita buscar palabras en un fichero de texto. Se debe mostrar el número de línea y su contenido, para cada línea que contenga la palabra buscada.
6. Desarrollar un programa que permita eliminar todas las ocurrencias de una palabra dada en un fichero de texto. El programa recibirá como argumentos de línea de comandos la ruta al fichero así como la palabra en cuestión. Este código producirá automáticamente un nuevo fichero con la siguiente nomenclatura: Si el fichero de entrada se llama *fichero.txt*, el fichero generado se llamará *fichero_2.txt*.
7. Escribir un método estático que reciba como entrada el nombre de un fichero de texto y devuelva estadísticas básicas sobre el mismo (como mínimo se debe incluir el número de palabras, el número de caracteres totales del texto y la longitud media de una palabra medida en nº de caracteres).
8. Modificar el programa ejemplo de la agenda telefónica (figuras 16.15, 16.16 y 16.17) de forma que:
 - un elemento individual de la agenda (un **ItemAgenda**) mantenga además del nombre, teléfono y código postal de un contacto su dirección postal. Tras hacerlo, ¿es necesario modificar algo en las operaciones de lectura y escritura en fichero de la clase **Agenda**?
 - Escribir operaciones en la clase **Agenda** para efectuar una búsqueda de un contacto por nombre o teléfono. Ambas operaciones devolverán el primer **ItemAgenda** que cumpla la condición en caso de que exista o **null** en el caso de que no sea así.

- Crear un nuevo programa principal que, mediante el uso de un menú, permita almacenar la agenda en curso en un fichero, añadir un nuevo contacto cuyos datos se pedirán al usuario, eliminar un contacto dado su número de teléfono y, finalmente, recuperar una agenda desde un fichero dado.
9. Una estación meteorológica necesita gestionar las medidas diarias de la pluviosidad en una determinada zona a lo largo de un año con las siguientes características:
- Se ha decidido construir una clase, denominada **Pluviometro** que tenga como atributos dos arrays, uno para almacenar el número de días de cada mes y otro para guardar las medidas de pluviosidad de dichos días. Por comodidad para el programador se ha decidido prescindir de usar la posición 0 de los arrays, para que el índice coincida con el número de mes o el número de día, de manera que las posiciones [0] de los arrays no se usarán.
 - **diasM** es un array de 13 **int** tal que **diasM[i]** es el numero total de días del mes **i** siendo $1 \leq i \leq 12$, de tal manera que como se ha comentado, **dia[0]** no se usará.
 - **lluvia** es un array bidimensional con 13 filas. **lluvia[i]** es un array de **diasM[i]+1** valores de tipo **double** (dicho de otra manera su longitud es **lluvia[i].length==diasM[i]+1**) tal que **lluvia[i][j]** representa la medida del día **j** del mes **i**, siendo $1 \leq i \leq 12$ y $1 \leq j \leq diasM[i]$. Las posiciones **lluvia[0][j]** y **lluvia[i][0]** no se usarán.

Se pide escribir un programa con la siguiente funcionalidad:

- a) leer los datos de pluviosidad desde un fichero *pluvio.dat* en el que cada línea tiene el siguiente formato:

```
dia mes medida  
...
```

y almacenarlos en la matriz **lluvia**, validando los valores de día y mes leídos. Las medidas no tienen por qué estar ordenadas cronológicamente. Un ejemplo de algunas líneas del fichero es el siguiente:

```
...  
24 11 312.12  
15 3 6.756  
14 8 12.5  
15 1 31.3  
16 3 212.0  
17 3 87.9  
18 3 3.56  
23 6 11.11  
...
```

- b) dada la matriz `lluvia` y cierto mes `m`, determinar la cantidad máxima llovida en un solo día a lo largo de dicho mes así como el día en que esta se produjo.
- c) dada la matriz `lluvia`, cierto mes `m` y una cantidad `lt` de litros, determinar un día de dicho mes en que la pluviosidad haya superado dicha cantidad. Si no existe, indicarlo con un mensaje.
- d) dada la matriz `lluvia` y cierto mes `m`, determinar si hubo al menos tres días consecutivos en dicho mes con una pluviosidad mayor a 100 litros cada uno de ellos.
- e) mostrar por pantalla las medidas del fichero de entrada `pluvio.dat` pero ordenadas cronológicamente.
10. Se desea modificar la solución al problema anterior, de forma que la lectura de los datos se produzca de un fichero binario (un `DataInputStream` o un `ObjectInputStream`) en lugar de un fichero de texto, tal y como se planteó antes.
- La lectura de los datos se efectuará en triadas de valores, enteros los dos primeros, que representarán, respectivamente, el día y mes de la medida; siendo el tercer valor uno en coma flotante (un `double`) que contendrá la cantidad llovida.
- Para determinar el momento en el que se produzca el final del fichero, se **deberá utilizar** la excepción `EOFException` tal y como se menciona en el capítulo (ejemplo de la figura 16.18).
11. Se tienen los siguientes datos referentes a la última vuelta ciclista local:

- `ciclistas`: array con los nombres de cada ciclista.
- `tiempos`: matriz en la que en cada fila `i` se tienen los tiempos de `ciclistas[i]` en cada una de las cinco etapas, el tiempo máximo empleado en una etapa es 180 minutos (se consideran valores enteros).

Se pide escribir un programa con la siguiente funcionalidad:

- a) diseñar la clase `VueltaCiclista` que tenga como atributos los arrays anteriormente mencionados.
- b) leer los datos de un fichero de texto con el formato:

```

nº de participantes
nombre t1 t2 t3 t4 t5
otronombre t1 t2 t3 t4 t5
...

```

- c) dado el nombre de un ciclista, mostrar por pantalla los tiempos empleados por este en cada una de las etapas si ha participado o el mensaje “*No ha participado en esta vuelta*” en caso contrario.

- d) mostrar por pantalla el nombre del ciclista ganador de la vuelta y el tiempo que este empleó. Gana la vuelta el ciclista cuya suma de tiempos de las cinco etapas es menor.
- e) mostrar por pantalla los ciclistas y sus tiempos ordenados según el tiempo empleado.

En todos los casos, los tiempos se mostrarán en horas y minutos.

12. Para resolver el problema anterior desde la *perspectiva de la programación orientada a objetos*, se plantea la siguiente organización de la información, que deberá ser implementada adecuadamente:

- Una clase **Ciclista**, mediante la que se mantendrá información relativa a cada uno de los mismos, en particular su **nombre** así como sus **tiempos**, array de 5 elementos enteros en los que, en cada uno de ellos, se mantendrá el tiempo empleado por el ciclista en cubrir la etapa correspondiente (el tiempo máximo empleado en una etapa es 180 minutos).
- Se deberá diseñar esta clase definiendo, además, los métodos constructores, consultores y modificadores que se consideren pertinentes.
- Una clase **VueltaCiclista** que contendrá un array **ciclistas**, de elementos de la clase **Ciclista**. Se considerará que el array tiene los elementos estrictamente necesarios; esto es, no existen posiciones del array no ocupadas.

Se deberá diseñar esta clase definiendo, además de los métodos que se consideren pertinentes (tales como constructores, etc.), métodos para almacenar y recuperar los datos de una **VueltaCiclista** en y desde un fichero de objetos (esto es, usando **ObjectInputStream** y **ObjectOutputStream**, así como los métodos necesarios para, al igual que en el problema anterior:

- dado el nombre de un ciclista, mostrar por pantalla los tiempos empleados por este en cada una de las etapas si ha participado o el mensaje “*No ha participado en esta vuelta*” en caso contrario.
- mostrar por pantalla el nombre del ciclista ganador de la vuelta y el tiempo que este empleó. Gana la vuelta el ciclista cuya suma de tiempos de las cinco etapas es menor.
- mostrar por pantalla los ciclistas y sus tiempos ordenados según el tiempo empleado.

13. Si se han resuelto los dos problemas anteriores, se está en posición de discutir las mejoras (y tal vez inconvenientes) que haya podido introducir la *solución orientada a objetos*.

Se pide señalar las diferencias más significativas entre las dos soluciones al problema de la vuelta ciclista, desde el punto de vista del almacenamiento y recuperación de la información en memoria externa. Tratar de responder a

la cuestión planteada, determinando cómo la posible variación de elementos en las clases, altera la organización de los ficheros y/o de las operaciones encargadas de su lectura o escritura.

¿Cuál de las organizaciones de los datos parece más cómoda para trabajar si tiene que sufrir modificaciones posteriores?

14. Se desea gestionar la información sobre los visitantes a cierto parque de atracciones *Gran Aventura*:

- de cada **visitante** se conoce sus apellidos, nombre, edad y un código como, por ejemplo, “GA325”.
- Existen **atracciones** que tienen cierto nombre, tales como: *DragonKhan*, *Furius*, *TutukiSplash*, *Stampida*.
- Además, se conocen los visitantes que han participado en cada una de las atracciones ya que se mantienen los códigos de aquellos visitantes que hayan accedido a cada una de las mismas.

Se pide escribir un programa para la gestión básica del parque de atracciones para lo que se deberá:

a) diseñar una clase **Visitante**, para mantener los datos individuales de cada uno de ellos. La clase deberá incluir los métodos de gestión (constructores, consultores y modificadores que se consideren pertinentes).

b) diseñar una clase **Atraccion** mediante la que se mantenga sus elementos propios, tales como su **nombre** y una lista de los visitantes que han accedido a los largo de un día a la misma (dicha lista se puede implementar mediante un array parcialmente completo de valores de tipo **Visitante**).

Esta clase contendrá, por lo menos, dos métodos, uno para añadir un nuevo visitante a los que han accedido a la atracción y otro para determinar si dado un código de visitante, ha accedido o no a la misma,

c) diseñar una clase **ParqueAtracciones**, mediante la que se mantenga una lista, implementada una vez más mediante un array, de las atracciones que están en funcionamiento a lo largo de un día. Esta clase deberá contener, por lo menos, los siguientes métodos:

- Métodos para almacenar y recuperar en un fichero de objetos los datos correspondientes a todo el parque de atracciones en un momento dado. Estos métodos recibirán como parámetro el nombre del fichero que contendrá los datos.
- Un constructor de la clase **ParqueAtracciones** que construirá uno de tales objetos a partir de los datos incluidos en un fichero de objetos (cuyo nombre recibirá el constructor como parámetro). Este constructor deberá utilizar el método diseñado en el punto anterior para recuperar la información de un parque de atracciones.

- Y, además, métodos para:
 - mostrar el número de visitantes que han accedido a cada una de las atracciones.
 - comprobar si un determinado visitante del que se conocen sus apellidos ha accedido a una determinada atracción.
 - mostrar los datos del **visitante** más joven de una atracción determinada.
 - mostrar por pantalla la lista de atracciones a las que ha accedido un determinado **visitante** del que se conocen sus apellidos.
- 15. En la clase **Agenda**, que se muestra en la figura 16.16, se ha definido el método de clase `leerAgenda(String)` que, a partir del nombre de un fichero, devuelve el objeto **Agenda** que se encuentra almacenado en el mismo.
Definir, utilizando el método anterior, un constructor de la clase **Agenda** que recibiendo como argumento el nombre del fichero en el que se ha almacenado el objeto **Agenda**, construya un objeto de dicho tipo. Modificar la clase **GestorPrueba** para utilizar el nuevo constructor.

Más información

- [Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011.
URL: <http://math.hws.edu/javanotes/>. Capítulo 11 (11.1, 11.2 y 11.3).
- [Ora11d] Oracle. *The JavaTM Tutorials*, 2011. URL: <http://download.oracle.com/javase/tutorial/>. Trail: Essential Java Classes. Lesson: Basic I/O.
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010.
Capítulo 10.
- [Sch07] H. Schildt. *Fundamentos de Java*. McGraw-Hill, 2007. Capítulo 10.

Capítulo 17

Tipos lineales. Estructuras enlazadas

Los tipos de datos lineales son aquéllos cuyos elementos están formados por linearidades o secuencias

$$d_0 d_1 \dots d_{n-1}, \quad n \geq 0$$

en los que todos los d_i son datos del mismo tipo, y sobre los que, en términos generales, se pueden hacer operaciones de inserción, búsqueda, eliminación de datos, consulta del dato que ocupa una determinada posición, etc.

Los diversos tipos de datos lineales surgen de aquellas clases de problemas en los que se manejan secuencias de datos y que necesitan una política concreta de gestión de sus elementos. Por ejemplo, en una secuencia de elementos puestos en cola para ser tratados o atendidos por orden de llegada, no se pueden permitir las mismas operaciones o métodos que sobre una lista de elementos entre los que no exista dicha restricción en su tratamiento.

En este capítulo se van a presentar tres tipos lineales: **Pila**, **Cola** y **Lista**. Estas clases se consideran básicas por ser idóneas en una gran variedad de aplicaciones informáticas, por lo que su funcionalidad e implementación está ampliamente estudiada. Por ejemplo, como se vio en el capítulo 5, Java gestiona la lista de registros de activación de las llamadas pendientes de terminar como una *pila*: todos los registros permanecen inaccesibles salvo el de la cima. En cambio, la lista de peticiones de compra electrónica de entradas se gestiona habitualmente como una *cola*.

Como se verá a lo largo del capítulo, el catálogo de métodos requerido por cada tipo lineal influye decisivamente en la forma más adecuada y eficiente de estructurar los datos. De hecho, en lugar de la representación mediante arrays que se discutió en la sección 10.4, en algunos casos será especialmente indicado el uso de las *listas o secuencias enlazadas* que se introducen en la siguiente sección.

17.1 Representación enlazada de secuencias

Una manera obvia de representar una secuencia consiste en usar un array, declarado de longitud suficientemente grande, en el que disponer consecutivamente los sucesivos elementos de la secuencia. Esta representación permite acceder directamente a cualquier elemento, independientemente de la posición que ocupe en la secuencia, pero resulta muy ineficiente cuando se realizan muchos movimientos de datos al añadir o eliminar elementos en posiciones intermedias.

17.1.1 Definición recursiva de secuencias. La clase Nodo

Una representación alternativa a la anterior se basa en disponer de memoria para los datos a medida que se van insertando en la secuencia, de modo que a diferencia de lo que sucede con las componentes de un array:

- los elementos consecutivos en la secuencia no tienen por qué aparecer consecutivos en memoria, y
- en la declaración de una secuencia no aparece limitado el número de elementos que pueden formar parte de ella.

Esta representación se basa en que todo dato tiene asociado un *enlace* o referencia a la posición en que se encuentra en el heap el siguiente dato de la secuencia, como se muestra en la figura 17.1.

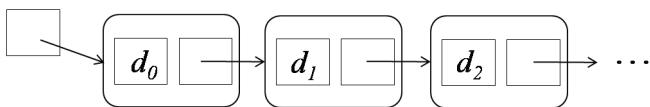


Figura 17.1: Representación enlazada de la secuencia $d_0d_1d_2\dots$

Genéricamente, se denomina *nodo* al objeto que agrupa un *dato* d de un tipo T cualquiera y un enlace *siguiente*, como se representa gráficamente en la figura 17.2.

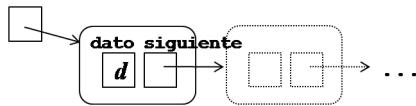


Figura 17.2: Estructura de un nodo.

Los nodos ayudan a definir recursivamente las secuencias enlazadas en términos de ellas mismas. Así pues, como puede verse en la figura 17.3, una *secuencia enlazada* de un número cualquiera $n \geq 0$ de datos es:

- la secuencia de $n = 0$ datos, en cuyo caso vale `null`, o
- una secuencia enlazada de $n \geq 1$ datos, en cuyo caso es un objeto nodo con un dato seguido por una secuencia enlazada de $n - 1$ datos.

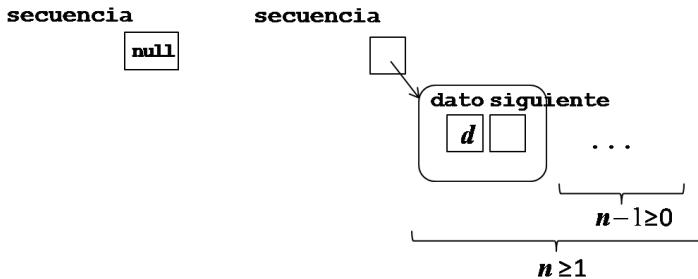


Figura 17.3: Secuencia enlazada de 0 o más datos.

En el caso particular en que el dato es de tipo `int` la clase `NodoInt` es:

```
/** Clase NodoInt: Nodo cuyo dato es un int */
class NodoInt {

    int dato;
    NodoInt siguiente;

    NodoInt(int d) {
        dato = d;
        siguiente = null;
    }

    NodoInt(int d, NodoInt s) {
        dato = d;
        siguiente = s;
    }
}
```

De los atributos de la clase cabe resaltar:

- El atributo **siguiente** se declara de la propia clase `NodoInt`; Java lo permite, debiéndose entender como una definición recursiva.
- Los atributos **dato** y **siguiente** se han declarado *friendly* para que algunas clases, como las de la sección 17.2, puedan acceder a ellos y manejarlos en la implementación de sus operaciones.

La clase se completa con dos métodos constructores:

- `NodoInt(int)`, que crea un nodo sin ningún otro a continuación.
- `NodoInt(int,NodoInt)`, que permite crear un nodo que antepone un dato entero a otro nodo previamente creado.

Las figuras 17.4(a) y 17.4(b) muestran el resultado de asignar a una variable `NodoInt sec` el nodo `new NodoInt(d)` y el nodo `new NodoInt(d,s)`, respectivamente.

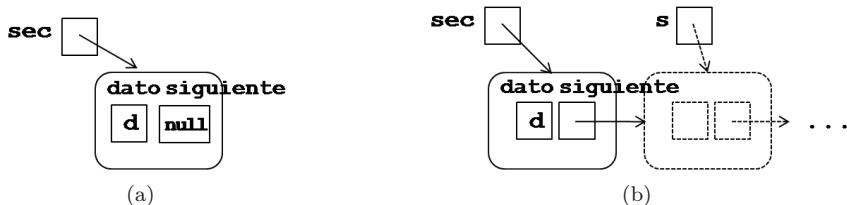


Figura 17.4: Los dos constructores de `NodoInt`.

Ejemplo 17.1. Supóngase una variable `sec` declarada de tipo `NodoInt` y la siguiente secuencia de instrucciones:

```

NodoInt sec = null;
sec = new NodoInt(10);
sec = new NodoInt(5,sec);
sec = new NodoInt(-2,sec);

```

La variable `sec` va pasando por los estados que se muestran en la figura 17.5. El número de sus datos va aumentando desde 0 hasta 3; dado que el constructor sitúa cada nuevo nodo a la cabeza de `sec`, los datos quedan en el orden -2 5 10.

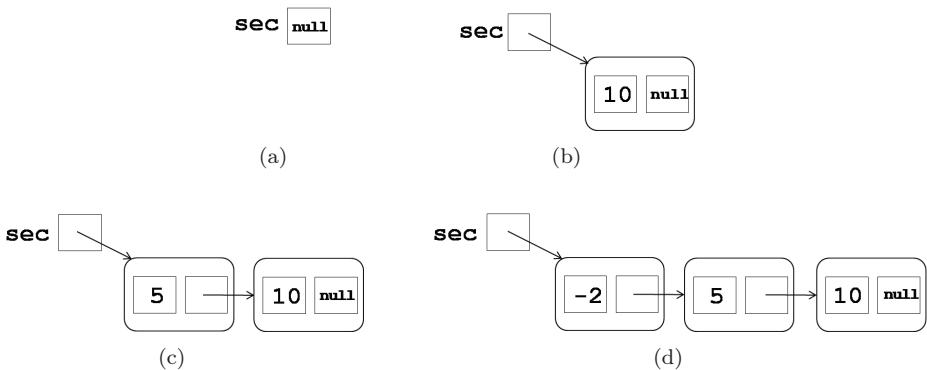


Figura 17.5: Formación de una secuencia enlazada de tres elementos.

El primer constructor de `NodoInt` es un caso particular del segundo y, de hecho, se podría haber definido como:

```
NodoInt(int d) {  
    this(d,null);  
}
```

Así, en el ejemplo 17.1 las primeras líneas de código se podrían haber escrito también como:

```
NodoInt sec = null;  
sec = new NodoInt(10,sec);
```

con idéntico resultado.

Ejemplo 17.2. Dado un entero $n \geq 1$, el siguiente código crea la secuencia de los n primeros impares, desde 1 hasta $2n - 1$ inclusive:

```
NodoInt sec = null;
for (int i=2*n-1; i>=1; i-=2) sec = new Nodo(i.sec);
```

En los ejemplos 17.1 y 17.2 se muestra cómo el segundo constructor `NodoInt(int, NodoInt)` facilita la inserción de un nuevo elemento en la cabeza de la secuencia, disponiendo de la memoria a medida que aumenta la talla de la secuencia. Pero el uso explícito de los enlaces permite acceder a otras posiciones de la secuencia, como en el ejemplo siguiente.

Ejemplo 17.3. El siguiente código se supone escrito en una clase con acceso *friendly* a los atributos de `NodoInt`. El primer nodo creado se sitúa como antes en cabeza de la secuencia `sec` y pasa a ser también el último de la secuencia. Las dos siguientes inserciones añaden el nuevo nodo a continuación de `ultimo` y después se actualiza en consecuencia dicha variable (figura 17.6):

```
NodoInt sec = null, ultimo = null;
sec = new NodoInt(10); ultimo = sec;
ultimo.siguiente = new NodoInt(5); ultimo = ultimo.siguiente;
ultimo.siguiente = new Nodo(-2); ultimo = ultimo.siguiente;
```

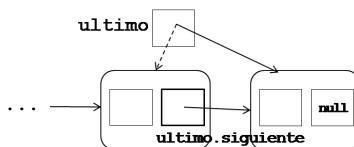


Figura 17.6: Uso explícito del enlace `siguiente` en una secuencia enlazada.

Así, la variable `sec` va pasando por los estados que se muestran en la figura 17.7. El número de sus datos va aumentando como en el ejemplo de la figura 17.5 desde 0 hasta 3, pero quedan en el orden 10 5 -2.

La clase `NodoInt` se considera como una clase subsidiaria que, al igual que los arrays, ayuda a dar estructura a los componentes o atributos de diversos tipos de datos. La clase se ha declarado *friendly* dado que su cometido básico es dar soporte a los enlaces con que se materializan las secuencias enlazadas. Su repertorio de métodos se limita a sus dos constructores y los atributos se dejan *friendly* para que se puedan manipular explícitamente, como en el ejemplo 17.3, por las clases que incluyan a `NodoInt` en su paquete.

Las clases externas a `NodoInt` pueden precisar encapsular en un método el tratamiento de una secuencia enlazada. Hay que tener en cuenta que al ser la secuencia un parámetro del método, se deberá terminar devolviendo como resultado la propia secuencia si dicho parámetro sufre localmente algún cambio.

Ejemplo 17.4. Se tiene un programa que manipula secuencias enlazadas de enteros que pueden empezar por un código entre 90 y 95 inclusive. Se necesita actualizar estas secuencias para que los códigos se limiten a tres, según:

- Los tres primeros códigos (entre 90 y 92 inclusive) se cambiarán a 91, y los tres últimos (entre 93 y 95 inclusive) se cambiarán a 92,
- las que no disponen de este código se hará que empiecen por 90.

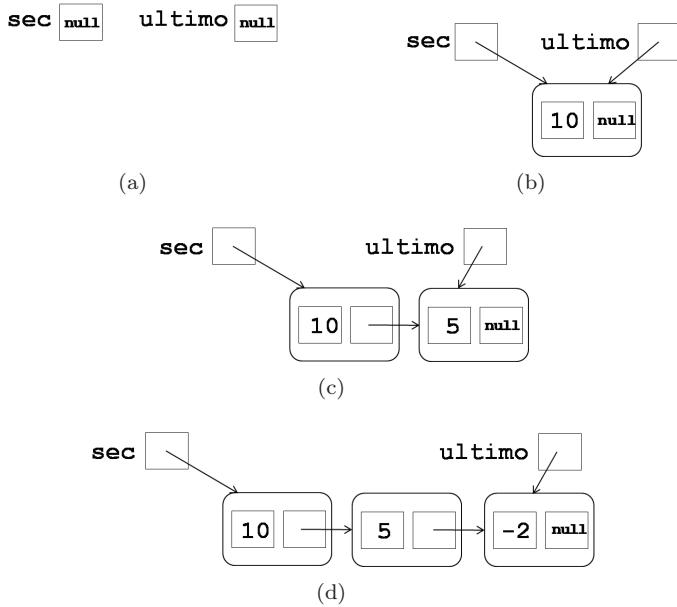


Figura 17.7: Formación de una secuencia enlazada de tres elementos. Los nuevos elementos se enlazan al final de la secuencia.

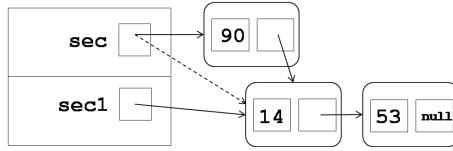
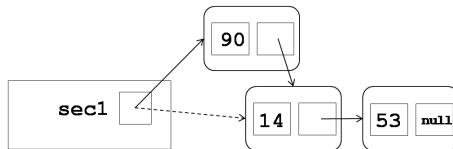
Para ello se define el siguiente método

```
public static NodoInt norma90(NodoInt sec) {
    if (sec!=null) {
        if (90<=sec.dato && sec.dato<=92) sec.dato = 91;
        else if (93<=sec.dato && sec.dato<=95) sec.dato = 92;
        else sec = new NodoInt(90,sec);
    }
    return sec;
}
```

El código anterior realiza en algunos casos cambios en nodos que se encuentran en el montículo, pero en un caso se modifica localmente el parámetro `sec`, por lo que hay que devolver `sec` para que dicho cambio sobreviva a la llamada al método. La figura 17.8 ilustra este caso mostrando los cambios producidos por

```
sec1 = norma90(sec1);
```

en donde `sec1` es una secuencia iniciada con los datos 14 53.

(a) Modificación de la variable local `sec`.(b) Asignación a `sec1` del resultado de la llamada.Figura 17.8: Actualización de `sec1` por el método `norma90`.

17.1.2 Recorrido y búsqueda en secuencias enlazadas

A diferencia de lo que sucede con los arrays, en las secuencias enlazadas no se tiene acceso directamente al elemento que ocupa una posición determinada en la secuencia, sino que sólo se puede acceder a un nodo desde su anterior. Es por ello que la mayoría de operaciones sobre estas secuencias requieren un recorrido o una búsqueda desde el primer nodo en adelante.

La estructura general de los algoritmos de recorrido y búsqueda en secuencias enlazadas es análoga a la de los esquemas de recorrido y búsqueda ascendentes en arrays (en sección 10.3.1 del capítulo 10). En un esquema iterativo se usa una variable que, como en el ejemplo anterior, indica en cada momento el primer elemento de la subsecuencia que todavía no se ha revisado (véase figura 17.9). Dicha variable:

- se sitúa inicialmente sobre el primer elemento,
- en cada pasada del bucle da acceso al dato a procesar, y después se actualiza al siguiente elemento,
- en el estado final, el valor `null` de dicha variable indica que no queda ningún dato por procesar.

En la discusión de los esquemas que vienen a continuación se suponen declarados un array `a` y un índice `ultimo` dentro del rango de `a`; por otra parte se supone `sec` de tipo `NodoInt`.

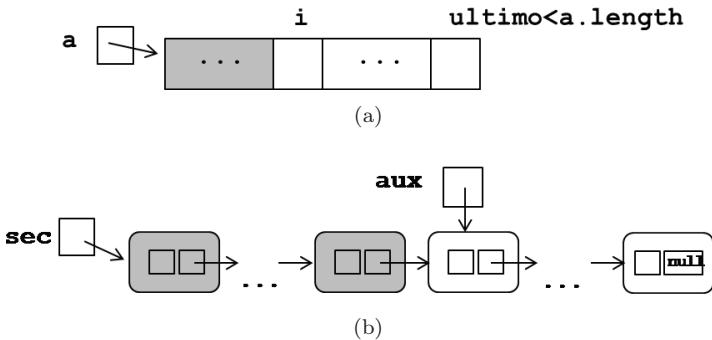


Figura 17.9: Recorridos de arrays y secuencias enlazadas.

Esquemas de recorrido

Los esquemas generales de un *recorrido de un array y de una secuencia enlazada* se muestran a continuación:

```

int i = 0;
while (i<=ultimo) {
    tratar(a[i]);
    i++;
}
NodoInt aux = sec;
while (aux!=null) {
    tratar(aux.datos);
    aux = aux.siguiente;
}

```

donde `tratar(a[i])` y `tratar(aux.datos)` indican, respectivamente, la operación a realizar con el elemento i -ésimo del array y con el dato del nodo `aux`. En ambos esquemas, el recorrido de n elementos es $\Theta(n)$.

Ejemplo 17.5. Se tiene una secuencia enlazada `sec` de enteros. Se desea que los valores de la lista saturen a un cierto valor `maximo`, es decir, que todos los valores $>\maximo$ se cambien a `maximo`.

```

NodoInt aux = sec;
while (aux!=null) {
    if (aux.datos>maximo) aux.datos = maximo;
    aux = aux.siguiente;
}

```

Esquemas de búsqueda

La búsqueda en una secuencia enlazada es, como en los arrays, una variación de un recorrido. El esquema general de una *búsqueda en un array* de un elemento que cumpla una cierta propiedad se muestra a continuación:

```
int i = 0;
while (i<=ultimo && !propiedad(a[i]))
    i++;
// Resolución de la búsqueda
if (i<=ultimo) ... // a[i] cumple la propiedad
else ...           // ningún elemento cumple la propiedad
```

donde `propiedad(a[i])` comprueba si el elemento i -ésimo del array cumple la propiedad enunciada.

El esquema general de una *búsqueda en una secuencia enlazada* de un elemento que cumpla una cierta propiedad es el siguiente:

```
NodoInt aux = sec;
while (aux!=null && !propiedad(aux.dato))
    aux = aux.siguiente;
// Resolución de la búsqueda
if (aux!=null) ... // aux.dato cumple la propiedad
else ...           // ningún elemento cumple la propiedad
```

donde `propiedad(aux.dato)` comprueba si el dato del nodo `aux` cumple la propiedad enunciada. El uso del operador cortocircuitado `&&` en la guarda del bucle asegura que sólo se accede al dato después de comprobar que `aux!=null`.

En ambos esquemas coinciden las cotas de complejidad $\Omega(1)$ y $O(n)$, siendo la talla n la longitud de la secuencia.

Ejemplo 17.6. El siguiente código busca la posición de la primera aparición en la secuencia del dato `d`. Si no aparece, la posición toma el valor -1.

```
NodoInt aux = sec; int i = 0, pos;
while (aux!=null && aux.dato!=d) {
    aux = aux.siguiente; i++;
}
if (aux!=null) pos = i; else pos = -1;
```

Este algoritmo se podría aplicar casi directamente a la búsqueda de un dato `d` en secuencias cuyos nodos contuviesen objetos en lugar de datos enteros o de otro tipo

primitivo. Bastaría con sustituir la comparación `!=` propia de los tipos primitivos por su correspondiente `!aux.dato.equals(d)`.

El esquema de búsqueda también es aplicable a resolver el acceso secuencial a una determinada posición, propio de las secuencias enlazadas.

Ejemplo 17.7. El acceso al i -ésimo elemento se logra buscando el nodo i -ésimo, y accediendo a sus atributos sólo en el caso en que dicho nodo exista. En el siguiente algoritmo la posición del nodo `aux` se va registrando en la variable entera `k`:

```
NodoInt aux = sec; int k = 0;
while (aux!=null && k<i) {
    aux = aux.siguiente;
    k++;
}
```

Si el bucle acaba con `aux!=null` entonces necesariamente `k==i`, y `aux` es el i -ésimo nodo. En caso contrario, dicho nodo no existe.

El coste de esta operación es $\Theta(i)$ si $i < n$, siendo n el número de elementos. En caso contrario, este algoritmo recorre toda la secuencia con un coste $\Theta(n)$ antes de detectar que dicha posición no existe. En resumen, el coste es $\Theta(\min(i, n))$.

En el caso concreto en que se desee acceder al último elemento de la secuencia, aquel que no tiene siguiente, es innecesario comprobar repetidamente que `aux` es diferente de `null`, ya que si se comprueba al inicio que existe al menos un nodo, la existencia de un último nodo está asegurada:

```
if (sec!=null) {
    NodoInt aux = sec;
    while (aux.siguiente!=null)
        aux = aux.siguiente;
    System.out.println("El último dato es "+ aux.dato);
}
else System.out.println("La secuencia está vacía");
```

17.1.3 Inserción y borrado en secuencias enlazadas

Gracias al uso explícito de los enlaces entre nodos, las operaciones de inserción y borrado en cualquier posición de una secuencia se resuelven sin realizar ningún movimiento en memoria de los datos ya existentes en la secuencia.

En primer lugar se va a considerar el problema de insertar un dato *d* en una determinada secuencia *sec*. Se pueden dar dos casos según dónde se deba hacer la inserción:

- El nuevo nodo se inserta en la primera posición, lo que incluye el caso de insertar en una secuencia vacía (figuras 17.10(a) y 17.10(b)):

```
sec = new NodoInt(d,sec);
```

- Se inserta en cualquier otra ubicación, es decir, se inserta en alguna subsecuencia de *sec*. Ello supone que se debe insertar detrás de algún nodo, de modo que si *ant* es una referencia al nodo detrás del cual se debe realizar la inserción:

```
ant.siguiente = new NodoInt(d,ant.siguiente);
```

inserta en cabeza de la subsecuencia *ant.siguiente*. Ello incluye el caso de insertar después del último nodo (figuras 17.11(a) y 17.11(b)).

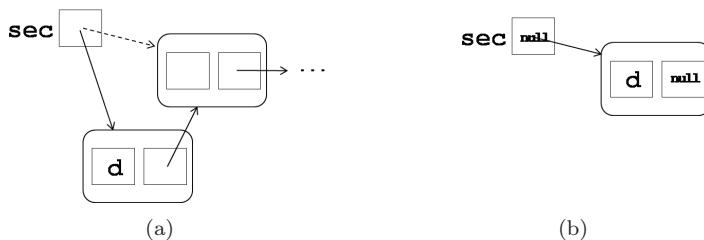


Figura 17.10: Inserción en cabeza.

Ejemplo 17.8. Dada una secuencia enlazada con un cierto número *n* de nodos, se desea insertar el elemento *d* en la posición *i*, siempre que $0 \leq i \leq n$.

El siguiente código comprueba si *i==0*, en cuyo caso la inserción es en cabeza. Sino, busca el nodo que ocupa la posición *i-1* para insertar a continuación un nuevo nodo con el dato *d*:

```
if (i==0) sec = new NodoInt(d,sec);
else {
    NodoInt aux = sec; int k = 0;
    while (aux!=null && k<i-1) {
        aux = aux.siguiente;
        k++;
    }
    if (aux!=null)
        aux.siguiente = new NodoInt(d,aux.siguiente);
}
```

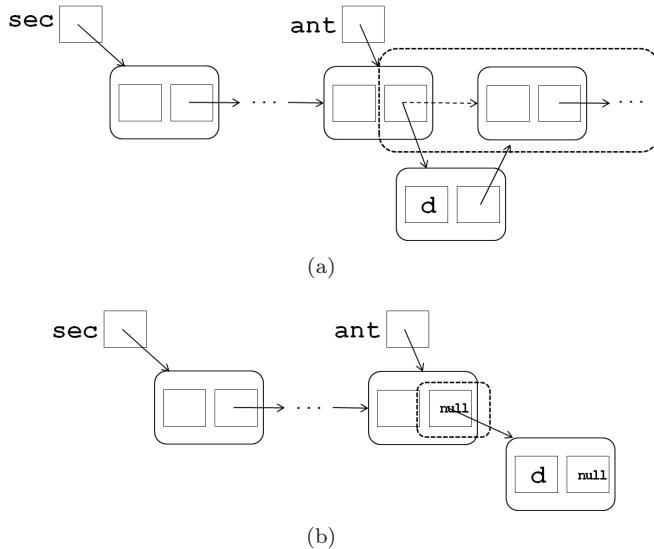


Figura 17.11: Inserción detrás de un nodo.

Si la búsqueda termina con fracaso, el número de nodos de `sec` ha resultado ser menor que `i`, y no se realiza ninguna inserción.

El coste $\Theta(\min(i, n))$ de la operación es debido a la búsqueda de la posición i -ésima, pues la inserción del nuevo nodo se resuelve con coste $\Theta(1)$ al no tener que realizar ningún movimiento de los datos preexistentes en la secuencia.

Aunque el tratamiento natural de las linealidades es en términos generales el iterativo, tiene interés considerar la versión recursiva. Para este ejemplo, se basa en el siguiente análisis de casos:

- Secuencia con $n = 0$ nodos, `sec==null`. Se inserta en cabeza de `sec`, y solamente si `i==0`.
- Secuencia con $n > 0$ nodos. Si `i==0`, se inserta en cabeza de `sec`, sino el problema se reduce a insertar en la posición `i-1` de la subsecuencia `sec.siguiiente`.

Lo que da lugar al siguiente método:

```
public static NodoInt insertar(NodoInt sec, int d, int i) {
    if (sec==null) { if (i==0) sec = new NodoInt(d); }
    else if (i==0) sec = new NodoInt(d,sec);
    else sec.siguiiente = insertar(sec.siguiiente,d,i-1);
    return sec;
}
```

Ejemplo 17.9. Supóngase una secuencia enlazada `sec` cuyos elementos están ordenados de menor a mayor. Se desea insertar un nuevo dato `d` en la secuencia, manteniéndola ordenada.

Para encontrar la ubicación del nuevo nodo, el siguiente código busca, usando una referencia `aux`, el primer elemento mayor o igual que `d`.

Acabada la búsqueda, la inserción debe distinguir si corresponde realizar una inserción en cabeza, o detrás de algún nodo. El primer caso se da cuando la búsqueda acaba con `aux==sec`: todos los elementos son mayores o iguales que `d`, lo que incluye el caso en que la secuencia esté vacía.

El segundo caso se da cuando en la secuencia hay al menos un elemento menor que `d`. Para facilitar la inserción, se usa una variable adicional `ant` que referencia en cada momento el nodo anterior a `aux`; al final, la inserción detrás de `ant` sitúa el nuevo nodo a continuación de todos los nodos con datos menores que `d`.

```
NodoInt aux = sec,
        ant = null; // el primer nodo no tiene anterior definido
while (aux!=null && aux.dato<d) {
    ant = aux;
    aux = aux.siguiente;
}
if (aux==sec) // ant==null
    sec = new NodoInt(d,sec);
else ant.siguiente = new NodoInt(d,aux);
```

La versión recursiva se basa en el siguiente análisis de casos:

- Secuencia con $n = 0$ nodos, `sec==null`. Se inserta en cabeza de `sec`. El dato `d` es el primero que se inserta en `sec`.
- Secuencia con $n > 0$ nodos. Si `sec.dato>=d`, se inserta en cabeza de `sec`, sino el problema se reduce a insertar ordenadamente el dato `d` en la subsecuencia `sec.siguiente`.

Lo que da lugar al siguiente método:

```
public static NodoInt insertarOrd(NodoInt sec, int d) {
    if (sec==null) sec = new NodoInt(d);
    else {
        if (sec.dato>=d) sec = new NodoInt(d,sec);
        else sec.siguiente = insertarOrd(sec.siguiente,d);
    }
    return sec;
}
```

La eliminación de un nodo en una secuencia no vacía se resuelve igualmente sin necesidad de mover los otros datos. Como en la inserción, se distinguen los siguientes casos:

- El nodo a eliminar es el primero de la secuencia (figura 17.12(a)):

```
sec = sec.siguiente;
```

En el caso particular de que sólo hubiera un nodo, `sec` se haría `null`, es decir, la secuencia vacía.

- El nodo a eliminar tiene un anterior:

```
ant.siguiente = nodo.siguiente;
```

en donde `nodo` es una referencia al nodo a eliminar y `ant` es una referencia al nodo anterior (figura 17.12(b)).

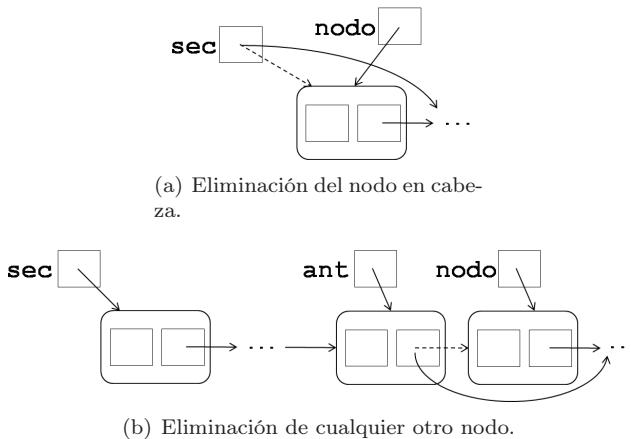


Figura 17.12: Eliminación de un nodo.

Ejemplo 17.10. Dada una secuencia enlazada `sec` se desea eliminar, si existe, la primera ocurrencia de un cierto dato `d`. Si dicho elemento no aparece, no se hace nada.

En primer lugar, mediante un esquema de búsqueda se localiza, si existe, el nodo cuyo dato es `d`. Si la búsqueda acaba con éxito, se elimina el nodo encontrado diferenciando si dicho nodo es el primero o tiene un anterior.

```
NodoInt aux = sec, ant = null;
while (aux!=null && aux.dato!=d) {
    ant = aux;
    aux = aux.siguiente;
}
if (aux!=null) // Éxito en la búsqueda
    if (ant==null) // aux es el primer nodo
        sec = aux.siguiente;
    else ant.siguiente = aux.siguiente;
```

El siguiente análisis de casos da forma recursiva al algoritmo:

- Secuencia con $n = 0$ nodos, `sec==null`. El dato `d` no está en `sec`, no se realiza ningún borrado.
- Secuencia con $n > 0$ nodos. Si se encuentra `d` en cabeza, se borra el primer nodo de `sec`; sino el problema se reduce a borrar la primera ocurrencia de `d` en la subsecuencia `sec.siguiente`.

Lo que conduce al siguiente método:

```
public static NodoInt borrar(NodoInt sec, int d) {
    if (sec!=null) {
        if (sec.dato==d) sec = sec.siguiente;
        else sec.siguiente = borrar(sec.siguiente,d);
    }
    return sec;
}
```

Ejemplo 17.11. Se tiene una secuencia enlazada `sec` con valores enteros. El siguiente código elimina todos los valores menores que un cierto `umbral`.

```
NodoInt aux = sec, ant = null;
while (aux!=null) {
    if (aux.dato<umbral) {
        if (aux==sec) sec = sec.siguiente;
        else ant.siguiente = aux.siguiente;
    }
    else ant = aux;
    aux = aux.siguiente;
}
```

El siguiente método resuelve el problema recursivamente. Este método se basa en un análisis por casos semejante al del ejemplo 17.10, excepto que en el caso

general, además de eliminar si procede el primer nodo, siempre hay que completar recursivamente el borrado en la subsecuencia `sec.siguiente`:

```
public static NodoInt borrarMenores(NodoInt sec, int umbral) {
    if (sec!=null) {
        NodoInt result = borrarMenores(sec.siguiente,umbral);
        if (sec.dato<umbral) sec = result;
        else sec.siguiente = result;
    }
    return sec;
}
```

17.2 Tipos lineales

En esta sección se estudian los tipos de datos *Pila*, *Cola*, y *Lista con punto de interés*, así como lo que se conoce como sus interfaces de operaciones, es decir, las funcionalidades respectivas de cada tipo. Para todos ellos se presentan dos posibles implementaciones en Java: una con arrays, y otra con secuencias enlazadas.

17.2.1 Pilas

Una pila (*stack* en inglés) es una secuencia en la que el acceso al primer elemento se realiza siguiendo un criterio LIFO (*Last In First Out*). Los elementos de una pila siempre se eliminan de ella en orden inverso al que fueron colocados, de modo que el último en entrar es el primero en salir, y viceversa, el primero en entrar es el último en salir.

Un ejemplo típico es la secuencia de registros de activación que coexisten en memoria, que se gestiona como una pila, y de ahí el nombre que recibe la zona de memoria en la que se ubican estos registros (véase la figura 17.13).

El tipo de datos *Pila* presenta la siguiente interfaz de operaciones disponibles: crear una pila, apilar un nuevo elemento sobre la pila, desapilar el elemento que se encuentra en la cima de la pila, consultar (sin desapilar) el valor del dato que se encuentra en la cima de la pila, preguntar si una pila está vacía y, por último, obtener el número de elementos de la pila.

En un contexto de programación orientada a objetos, podemos identificar la operación de crear una pila como un método constructor de la clase *Pila*. Las operaciones de conocer el dato que hay en la cima, saber si la pila está vacía o no, y cuántos elementos contiene, se implementan por medio de métodos consultores que no alteran el estado de la pila. Finalmente, las operaciones de apilar y desapilar sí permiten modificar la estructura de la pila añadiendo o eliminando elementos, respectivamente, y se implementan por medio de métodos modificadores de la clase

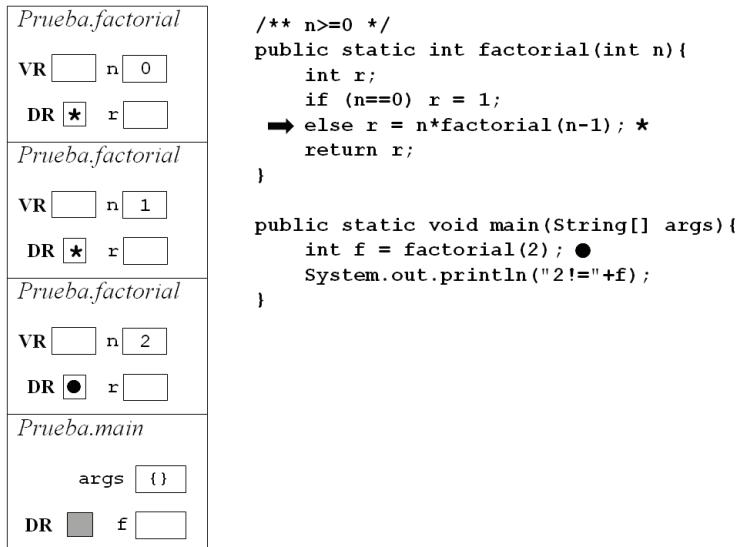


Figura 17.13: Pila de registros de activación.

Pila. El interfaz genérico de una **Pila** se puede implementar en Java siguiendo el esquema de la tabla 17.1, donde **Tipo** representa el tipo de los datos de la **Pila**.

<code>public Pila ()</code>	Crea una nueva Pila vacía.
<code>public void apilar (Tipo elemento)</code>	Apila el elemento sobre la Pila .
<code>public Tipo desapilar ()</code>	Desapila el elemento de la cima de la Pila y lo devuelve.
<code>public Tipo cima ()</code>	Devuelve (sin desapilarlo) el elemento de la cima de la Pila .
<code>public boolean esVacia ()</code>	Devuelve <code>true</code> si la Pila está vacía. <code>false</code> en caso contrario.
<code>public int talla ()</code>	Devuelve el número de elementos ($n \geq 0$) de la Pila .

Tabla 17.1: Interfaz de la clase **Pila**.

Es interesante constatar que todas estas operaciones se pueden implementar en Java de manera que cada método presente un coste constante, independientemente del número de elementos de la **Pila**.

Ejemplo 17.12. Se tiene una expresión **s** de tipo **String** en la que pueden aparecer subexpresiones encerradas entre parejas de `'(' ','')`, `{',''}`, siguiendo las reglas habituales de parentización. El siguiente método verifica si la expre-

sión está bien o mal parentizada. Por ejemplo, el método devolvería `true` para "`{(a+b)/(c-a})*(a-b)`", y `false` para "`{(a+b)/(c-a})*(a-b"`.

```
public static boolean expresionBienParentizada(String s) {
    Pila p = new Pila(); //los elementos de p son int
    for (int i=0; i<s.length(); i++) {
        char c = s.charAt(i);
        if (c == '(' || c == '{') p.apilar(i);
        else if (c == ')' || c == '}') {
            if (!p.esVacia()) { //quedan paréntesis por cerrar
                char parC = s.charAt(p.desapilar());
                if ((parC == '(' && c == ')') ||
                    (parC == '{' && c == '}')) return false;
            } else return false;
        }
    }
    if (p.esVacia()) // se han cerrado todos los paréntesis
        return true; else return false;
}
```

El método se basa en que, leyendo la expresión de izquierda a derecha, sólo puede aparecer un paréntesis de cierre de un determinado tipo si el último paréntesis de apertura pendiente de ser cerrado es del mismo tipo.

Para ello, se realiza un recorrido ascendente de la expresión, y cada vez que se encuentra un paréntesis de apertura se apila su posición sobre una pila (creada inicialmente de un cierto tipo `Pila` cuyos elementos son de tipo `int`). De esta manera, en la cima de la pila se tiene cuál es el último paréntesis de apertura pendiente de ser cerrado.

En cambio, si se encuentra un paréntesis de cierre, para que la expresión esté bien parentizada el paréntesis de la cima de la pila debe ser el correspondiente paréntesis de apertura, en cuyo caso se extrae de la pila y se da por cerrado. Si no es así, y la pila está vacía o el paréntesis encontrado no es el apropiado, entonces se concluye que la expresión está mal parentizada.

Cuando se termina el recorrido de toda la expresión, para que la expresión estuviera bien parentizada todos los paréntesis que se hubieran apilado se tendrán que haber cerrado con su correspondiente pareja, dejando finalmente la pila vacía.

Implementación mediante arrays

La clase `Pila` se puede definir mediante los siguientes campos o atributos: un array (`elArray`) para almacenar los datos, un índice al mismo (`cima`) que señale la cima de la pila (los datos se disponen consecutivamente entre las posiciones 0 y `cima`),

y una constante que defina la dimensión inicial del array (MAX). Por simplificación y coherencia con el resto del capítulo, el código que se muestra a continuación representa una pila cuyos datos son de tipo int:

```
public class PilaIntArray {
    private int[] elArray;
    private int cima;
    private static final int MAX = ...;

    ...
}
```

Inicialmente, el método constructor permite crear una pila vacía. Para ello, creamos el array de tamaño MAX y asignamos un valor de -1 al índice que apunta a la cima de la pila. Esto nos facilitará la implementación del método **apilar**, por ejemplo, ya que podemos programar el mismo algoritmo a la hora de apilar un nuevo elemento, independientemente de si anteriormente la pila estaba vacía o no, es decir, en ambos casos se almacena en la posición **cima+1**.

```
public PilaIntArray() {
    elArray = new int[MAX];
    cima = -1;
}

public void apilar(int x) {
    if (cima+1 == elArray.length)
        duplicaArray();
    cima++;
    elArray[cima] = x;
}
```

donde **duplicaArray** es un método privado que permite redimensionar la longitud de **elArray**, incrementando el espacio de memoria asignado al mismo.

```
private void duplicaArray() {
    int[] aux = new int[2*elArray.length];
    for (int i=0; i<elArray.length; i++) aux[i] = elArray[i];
    elArray = aux;
}
```

Las operaciones de **desapilar** y **cima** requieren que la pila no esté vacía. Ambas devuelven el dato que se encuentra en el índice **cima**. La diferencia estriba en que **desapilar** decrementa el valor del índice **cima** una posición a la izquierda. Nótese que al **desapilar**, el dato que estaba en la cima de la pila sigue estando físicamente

almacenado en el array, pero como el índice **cima** se ha decrementado, dicho valor queda almacenado fuera de la región de interés correspondiente a la pila, y por lo tanto es como si efectivamente se desapilase. Además, dicho valor se sobrescribirá si la pila crece de nuevo hasta ese tamaño.

```
/** La talla de la pila debe ser >0 */
public int desapilar() {
    // cima >= 0
    int x = elArray[cima];
    cima--;
    return x;
}

/** La talla de la pila debe ser >0 */
public int cima() {
    // cima >= 0
    return elArray[cima];
}
```

Por último, determinar el número de elementos de la pila n , y por extensión, deducir si está vacía o no, sólo depende del índice **cima**, siendo $n = \text{cima}+1$, dado que todos los datos se encuentran en el array entre las posiciones $[0..\text{cima}]$.

```
public boolean esVacia() {
    return (cima== -1);
}

public int talla() {
    return cima+1;
}
```

Implementación mediante representación enlazada

La implementación de una pila mediante una secuencia enlazada de nodos establece como atributos un objeto **NodoInt** que representa la **cima** de la pila, y un entero **talla** que indica el número total de datos que contiene.

```
public class PilaIntEnla {
    private NodoInt cima;
    private int talla;

    ...
}
```

El método constructor crea una pila vacía, es decir, una secuencia nula donde no hay ningún dato almacenado.

```
public PilaIntEnla() {  
    cima = null;  
    talla = 0;  
}
```

Para apilar un dato, se utiliza el constructor `NodoInt` más apropiado, en este caso, el que crea un nuevo nodo enlazándolo delante de la `cima` actual. El nuevo nodo introducido en la estructura es ahora la nueva `cima` de la pila.

```
public void apilar(int x) {  
    cima = new NodoInt(x,cima);  
    talla++;  
}
```

Las operaciones de `desapilar` y `cima` requieren que la pila no esté vacía. Ambas acceden al dato que se encuentra en el nodo `cima` y lo devuelven. En el caso de `desapilar`, la referencia a la `cima` se actualiza al nodo `siguiente` de la `cima` actual. El resto de métodos (`esVacia` y `talla`) son sencillos de implementar.

```
/** La talla de la pila debe ser >0 */  
public int desapilar() {  
    // cima != null  
    int x = cima.dato;  
    cima = cima.siguiente;  
    talla--;  
    return x;  
}  
  
/** La talla de la pila debe ser >0 */  
public int cima() {  
    // cima != null  
    return cima.dato;  
}  
  
public boolean esVacia() {  
    return (cima==null); // o return (talla==0);  
}  
  
public int talla() {  
    return talla;  
}
```

Comparación de implementaciones

La complejidad temporal de todas las operaciones en ambas implementaciones es constante e independiente del tamaño del problema: $T(n) \in \Theta(1)$.

En cuanto a la complejidad espacial, la implementación con arrays presenta el inconveniente de tener que estimar adecuadamente el tamaño máximo del array, y además, la reserva de un espacio que en muchos casos no se utilizará. Este consumo adicional de espacio no tendrá demasiada importancia si el tipo de las componentes del array es relativamente pequeño, como es el caso de una pila de enteros o de objetos (las componentes del array son referencias). Por otro lado, la representación enlazada requiere un espacio de memoria adicional para almacenar los enlaces.

17.2.2 Colas

Una cola (*queue*) es una colección de datos del mismo tipo en la que el acceso se realiza siguiendo un criterio FIFO (*First In First Out*), es decir, el primer elemento que llega (que entra en la cola) es el primero en ser atendido (en ser eliminado de la cola).

En la vida real, las colas se utilizan muy a menudo como política de gestión de un modelo de negocio cliente-servidor. Por ejemplo, los usuarios de un comercio suelen hacer *cola* frente a las cajas a la hora de comprar un producto. Los ordenadores, a su vez, también gestionan muchos procesos mediante colas como, por ejemplo, la impresión de documentos (véase la figura 17.14).



The screenshot shows a Windows-style application window titled "Impresora". The menu bar includes "Impresora", "Documento", "Ver", and "Ayuda". The main area is a table displaying four documents in the print queue:

Nombre del documento	Estado	Páginas	Tamaño	Enviado
Resumen.pdf	Imprimiendo...	1	32,3 KB/32,4 KB	15:06:11
Java Printing		1	8,10 KB	15:06:20
C:\Documents and Settings...		14	19,2 MB	15:06:34
Manual.pdf		1	62,4 KB	15:07:05

At the bottom of the window, it says "4 documentos en la cola".

Figura 17.14: Cola de impresión.

El tipo de datos Cola presenta la siguiente interfaz de operaciones disponibles: crear una cola, encolar un nuevo elemento a la cola, desencolar el elemento que se encuentra a la cabeza de la cola, consultar (sin desencolar) el valor del dato que está el primero de la cola, preguntar si una cola está vacía, y por último, obtener

el número de elementos de la cola. Siguiendo una aproximación similar a la de la clase **Pila**, este interfaz se implementa en Java por medio de:

- Un método constructor de objetos **Cola**.
- Sendos métodos modificadores para los procesos de encolar y desencolar.
- Varios métodos consultores que no alteran la estructura de una **Cola**.

El interfaz genérico de una **Cola** se implementa en Java siguiendo el esquema de la tabla 17.2, donde **Tipo** representa el tipo de los datos de la **Cola**.

<code>public Cola ()</code>	Crea una nueva Cola vacía.
<code>public void encolar (Tipo elemento)</code>	Inserta el elemento al final de la Cola .
<code>public Tipo desencolar ()</code>	Extrae el elemento de la cabeza de la Cola y lo devuelve.
<code>public Tipo primero ()</code>	Devuelve (sin desencolarlo) el elemento a la cabeza de la Cola .
<code>public boolean esVacia ()</code>	Devuelve true si la Cola está vacía. false en caso contrario.
<code>public int talla ()</code>	Devuelve el número de elementos ($n \geq 0$) de la Cola .

Tabla 17.2: Interfaz de la clase **Cola**.

Nuevamente, se debe constatar que todas estas operaciones se pueden implementar en Java de manera que cada método presente un coste constante, independientemente del número de elementos de la cola.

Ejemplo 17.13. El siguiente método devuelve **true** si el elemento **x** se encuentra en la cola **c**, y **false** en caso contrario. A pesar de que la semántica del método representa un concepto de búsqueda, la resolución precisa realizar un recorrido, ya que el método debe preservar la estructura de la cola, restaurándola a su estado original tras las modificaciones introducidas por el propio método durante el proceso de búsqueda.

```
public static boolean buscar(Cola c, int x) {
    boolean exito = false;
    for (int i=0; i<c.talla(); i++) {
        int dato = c.desencolar();
        if (dato == x) exito = true;
        c.encolar(dato)
    }
    return exito;
}
```

Implementación mediante arrays

La clase `Cola` se puede definir mediante los siguientes campos o atributos: un array *circular*¹ (`elArray`) para almacenar los datos, dos índices al mismo (`primero` y `ultimo`) que señalen el principio y el final de la cola (los datos se disponen consecutivamente entre ambos índices del array), un contador del número de elementos en cola (`talla`), y una constante que defina la dimensión inicial del array (`MAX`). El siguiente código es de una cola de datos de tipo `int`:

```
public class ColaIntArray {
    private int[] elArray;
    private int primero, ultimo, talla;
    private static final int MAX = ...;

    ...
}
```

Inicialmente, el método constructor permite crear una cola vacía. Para ello, creamos el array de tamaño `MAX` y asignamos un valor de `-1` al índice que apunta al último de la cola. Esto facilita la implementación de otros métodos, ya que por ejemplo, el algoritmo para encolar un nuevo elemento es el mismo, independientemente de si anteriormente la cola estaba vacía o no, es decir, en ambos casos se almacenaría en la posición `(ultimo+1)%elArray.length`. Por este motivo, `primero` se ha de inicializar a `0`, para que al encolar el primer elemento, dicho dato sea a su vez el que esté a la cabeza de la cola (`primero`). El atributo `talla` también se inicializa a `0` por razones obvias.

```
public ColaIntArray() {
    elArray = new int[MAX];
    talla = primero = 0;
    ultimo = -1;
}

public void encolar(int x) {
    if (talla == elArray.length)
        duplicaArray();
    ultimo = (ultimo+1)%elArray.length;
    elArray[ultimo] = x;
    talla++;
}
```

donde `duplicaArray` es un método privado que permite redimensionar la longitud de `elArray`, incrementando el espacio de memoria asignado al mismo.

¹Para reutilizar todas las posiciones del array sin desplazar elementos, se considera el array como si fuera circular, sin principio ni fin, donde tras la posición $n - 1$ va la posición 0.

```
private void duplicaArray() {
    int[] aux = new int[2*elArray.length];
    for (int i=0, pos=primero; i<elArray.length; i++) {
        aux[i] = elArray[pos];
        pos = (pos+1)%elArray.length;
    }
    primero = 0;
    ultimo = elArray.length-1;
    elArray = aux;
}
```

Las operaciones **desencolar** y **primero** requieren que la cola no esté vacía. Ambas devuelven el dato que se encuentra en el índice **primero**. La diferencia estriba en que **desencolar** incrementa el valor del índice **primero** una posición a la derecha.

```
/** La talla de la cola debe ser >0 */
public int desencolar() {
    // talla > 0
    int x = elArray[primero];
    primero = (primero+1)%elArray.length;
    talla--;
    return x;
}

/** La talla de la cola debe ser >0 */
public int primero() {
    // talla > 0
    return elArray[primero];
}
```

Por último, el número de elementos de la cola n lo indica el atributo **talla**, y a partir del mismo también se puede deducir si la cola está vacía o no.

```
public boolean esVacia() {
    return (talla==0);
}

public int talla() {
    return talla;
}
```

Implementación mediante representación enlazada

La implementación de una cola mediante una secuencia enlazada de nodos establece como atributos dos objetos **NodoInt** representando el inicio (**primero**) y el final

(*ultimo*) de la cola, respectivamente, y un entero *talla* que indica el número total de datos que contiene. El acceso directo al último nodo de la secuencia nos permite poder **encolar** un nuevo nodo detrás del mismo sin tener que recorrer toda la secuencia desde el primero de sus nodos, es decir, implementar dicha operación con un coste constante, independiente de la talla.

```
public class ColaIntEnla {
    private NodoInt primero, ultimo;
    private int talla;

    ...
}
```

El método constructor crea una cola vacía, es decir, una secuencia nula donde no hay ningún dato almacenado.

```
public ColaIntEnla() {
    primero = ultimo = null;
    talla = 0;
}
```

Para encolar un nuevo dato, hay que diferenciar si la cola está vacía o no. Si no está vacía, el nuevo nodo se ha de enlazar a continuación del último nodo. En caso contrario, la cola está vacía, y por tanto, el nuevo nodo es el **primero**. En todo caso, el nuevo nodo será ahora el que ocupe el último lugar en la cola.

```
public void encolar(int x) {
    NodoInt nuevo = new NodoInt(x);
    if (ultimo!=null) ultimo.siguiente = nuevo;
    else primero = nuevo;
    ultimo = nuevo;
    talla++;
}
```

Las operaciones **desencolar** y **primero** requieren que la cola no esté vacía. Ambas acceden al dato que se encuentra en el nodo **primero**, devolviéndolo. Al **desencolar**, la referencia a **primero** se actualiza al nodo **siguiente** a éste. Si sólo hubiera 1 elemento en la cola (los nodos **primero** y **ultimo** coinciden), **desencolar** afecta asimismo al atributo **ultimo**, dejándolo también como **null**. El resto de métodos (**esVacia** y **talla**) no precisan discusión.

```
/** La talla de la cola debe ser >0 */
public int desencolar() {
    // primero != null
    int x = primero.dato;
    primero = primero.siguiente;
    if (primero==null) ultimo = null;
    talla--;
    return x;
}
```

```
/** La talla de la cola debe ser >0 */
public int primero() {
    // primero != null
    return primero.dato;
}

public boolean esVacia() {
    return (primero==null);
}

public int talla() {
    return talla;
}
```

Comparación de implementaciones

La complejidad temporal de todas las operaciones en ambas implementaciones es constante e independiente de la talla del problema: $T(n) \in \Theta(1)$.

En cuanto a la complejidad espacial, al igual que sucedía con las pilas, la implementación con arrays presenta el inconveniente de estimar el tamaño del array, y además, la reserva de un espacio que en muchos casos no se utilizará. Pero del mismo modo, este consumo adicional de espacio no tendrá demasiada importancia si el tipo de las componentes del array es relativamente pequeño, como es el caso de una cola de números enteros o de objetos Java (referencias). En este caso también, la representación enlazada requiere un espacio de memoria adicional para almacenar los enlaces.

17.2.3 Listas con punto de interés

Una lista es una secuencia en la que el acceso se puede realizar en cualquier punto. En una lista se puede buscar un elemento, insertar, o eliminar un elemento, en una posición dada.

Además, en una lista suele existir el concepto de posición o elemento activo, conocido como el *punto de interés* (o *cursor*) de la lista. Si la lista tiene n datos, $n \geq 0$, numerados de 0 a $n - 1$, el cursor puede estar:

- en una posición $0 \leq i \leq n - 1$: *sobre el elemento i-ésimo*,
- en la posición $i = n$: *a la derecha del todo*.

Dicho cursor se puede mover, posición a posición, a lo largo de la lista.

El ejemplo paradigmático de lista con punto de interés o cursor es la línea de comandos del sistema (figura 17.15(a)), que se edita como una lista de caracteres en la que se puede borrar el carácter remarcado por el cursor, o insertar un carácter por delante del cursor, etc. Por ejemplo:

```
W:\_  
W:\_E_  
W:\_Ej_  
W:\_Eje_  
W:\_Ejen_  
W:\_Ejen  
W:\_Eje_  
W:\_Ejem_
```

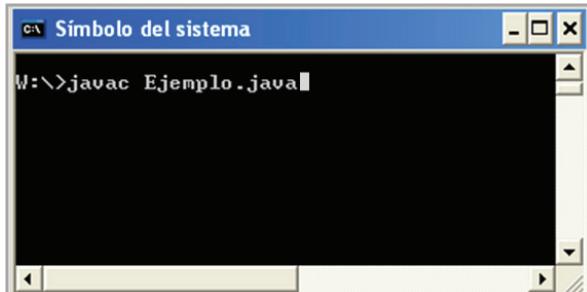
Otros ejemplos de listas que usan un cursor para acceder a sus elementos son las listas de opciones de un menú desplegable, como el de la figura 17.15(b).

En el modelo que se va a presentar, el movimiento del cursor se limita exclusivamente a avanzarlo una posición a la derecha o hacerlo retroceder al inicio. El conjunto de operaciones del tipo disponibles es el siguiente: crear una lista, situar el punto de interés al principio de la lista, hacer avanzar una posición el punto de interés, consultar si el punto de interés está al final de la lista, insertar un nuevo elemento en el punto de interés, eliminar el elemento que se encuentra en el punto de interés, consultar (sin eliminarlo) el valor del dato que está en el punto de interés, obtener el número de elementos de la lista, y por último, preguntar si una lista está vacía. Siguiendo una aproximación similar a la de la clases **Pila** y **Cola**, este interfaz se implementa en Java por medio de:

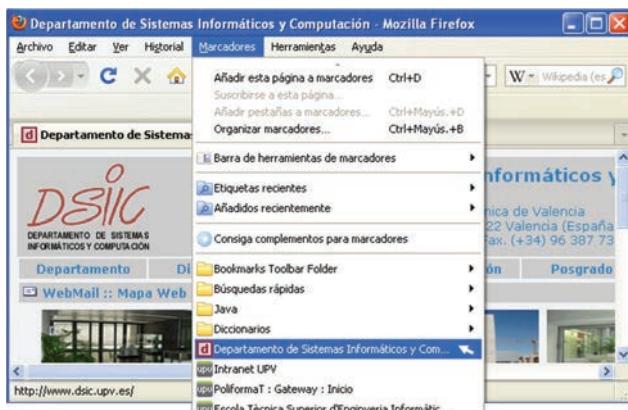
- Un método constructor de objetos **Lista**.
- Dos métodos modificadores para desplazar el punto de interés.
- Otros dos métodos modificadores para los procesos de insertar y eliminar.
- Varios métodos consultores que no alteran la estructura de una **Lista**.

El interfaz genérico de una **Lista** se implementa en Java siguiendo el esquema de la tabla 17.3, donde **Tipo** representa el tipo de los datos de la **Lista**.

Ejemplo 17.14. Este es un ejemplo de uso de una lista con cursor o punto de interés a través de un esquema genérico de búsqueda de un elemento sobre la misma. El siguiente método devuelve **true** si el elemento **x** se encuentra en la lista **l**. En ese caso, sitúa el punto de interés de **l** en la primera ocurrencia de **x** en **l**. En caso contrario, el punto de interés se queda al final de **l** y devuelve **false**.



(a) Línea de comandos del sistema.



(b) Lista de opciones de un menú desplegable.

Figura 17.15: Listas con cursor o punto de interés.

```
public static boolean buscar(Lista l, int x) {
    l.inicio();
    while (!l.esFin() && l.recuperar() != x)
        l.siguiente();
    if (!l.esFin()) return true;
    else return false;
}
```

Implementación mediante arrays

La clase **Lista** se puede definir mediante los siguientes campos o atributos: un array (**elArray**) para almacenar los datos, un índice al mismo (**PI**) para representar el punto de interés, un contador (**talla**) del número de elementos en la lista (los datos se disponen consecutivamente entre las posiciones 0 y **talla-1**), y una

<code>public Lista ()</code>	Crea una nueva <code>Lista</code> vacía.
<code>public void inicio ()</code>	Sitúa el cursor al principio de la <code>Lista</code> .
<code>public void siguiente ()</code>	Desplaza el cursor una posición a la derecha de su posición actual.
<code>public void insertar (Tipo elemento)</code>	Inserta el <code>elemento</code> en el punto de interés de la <code>Lista</code> .
<code>public Tipo eliminar ()</code>	Elimina el elemento del punto de interés de la <code>Lista</code> , devolviéndolo.
<code>public Tipo recuperar ()</code>	Devuelve (sin eliminarlo) el elemento que está en el cursor.
<code>public boolean esVacia ()</code>	Devuelve <code>true</code> si la <code>Lista</code> está vacía. <code>false</code> en caso contrario.
<code>public int talla ()</code>	Devuelve el número de elementos ($n \geq 0$) de la <code>Lista</code> .
<code>public boolean esFin ()</code>	Devuelve <code>true</code> si el cursor está al final de la <code>Lista</code> . <code>false</code> en caso contrario.

Tabla 17.3: Interfaz de la clase `Lista`.

constante que defina la dimensión inicial del array (`MAX`). El siguiente código es de una lista de datos de tipo `int`:

```
public class ListaPIIntArray {
    private int[] elArray;
    private int PI, talla;
    private static final int MAX = ...;

    ...
}
```

Inicialmente, el método constructor permite crear una lista vacía. Para ello, creamos el array de tamaño `MAX` y asignamos un valor de 0 al atributo `talla`. El índice asociado al punto de interés lo inicializamos al final de la lista (vacía, en este caso), cuyo valor es `talla`, ya que la lista se almacena en `[0..talla-1]`.

```
public ListaPIIntArray() {
    elArray = new int[MAX];
    talla = PI = 0;
}
```

Situar el punto de interés al principio de la lista no tiene mayor complejidad.

```
public void inicio() {
    PI = 0;
}
```

Sin embargo, mover el punto de interés una posición a la derecha de la actual requiere que dicho punto de interés no se encuentre todavía al final de la lista.

```
/** El cursor no debe estar al final de la lista */
public void siguiente() {
    // PI < talla
    PI++;
}
```

Al insertar un nuevo dato en el punto de interés, hay que desplazar a la derecha todos los datos desde dicho punto de interés hasta el final de la lista. Esto implica que el coste en el peor caso sea lineal respecto a la talla de la lista. El punto de interés debe mantenerse en el mismo elemento.

```
public void insertar(int x) {
    if (talla == elArray.length)
        duplicaArray();
    for (int k=talla-1; k>=PI; k--)
        elArray[k+1] = elArray[k];
    elArray[PI] = x;
    PI++;
    talla++;
}
```

donde `duplicaArray` es un método privado que permite redimensionar la longitud de `elArray`, incrementando el espacio de memoria asignado al mismo.

```
private void duplicaArray() {
    int[] aux = new int[2*elArray.length];
    for (int i=0; i<elArray.length; i++) aux[i] = elArray[i];
    elArray = aux;
}
```

Las operaciones `eliminar` y `recuperar` requieren que el punto de interés no esté al final. Ambas acceden al elemento que se encuentra en el cursor, devolviéndolo. En el caso de `eliminar`, hay que desplazar a la izquierda todos los datos desde dicho punto de interés hasta el final de la lista. Esto implica nuevamente que el coste en el peor caso sea lineal respecto a la talla de la lista.

```
/** El cursor no debe estar al final de la lista */
public int eliminar() {
    // PI < talla
    int x = elArray[PI];
    for (int k=PI+1; k<talla; k++)
        elArray[k-1] = elArray[k];
    talla--;
    return x;
}
```

```
/** El cursor no debe estar al final de la lista */
public int recuperar() {
    // PI < talla
    return elArray[PI];
}
```

Finalmente, saber si la lista está vacía o no, cuántos elementos $n \geq 0$ contiene la lista, o si el cursor está o no al final de la misma, son operaciones de consulta que dependen directamente del atributo **talla**.

```
public boolean esVacia() {
    return (talla==0);
}

public int talla() {
    return talla;
}

public boolean esFin() {
    return (PI==talla);
}
```

Implementación mediante representación enlazada

La implementación de una lista mediante una secuencia enlazada de nodos establece como atributos dos objetos **NodoInt** representando el primer nodo (**primero**) y el del punto de interés (**PI**) de la lista, respectivamente, y un entero **talla** que indica el número total de datos almacenados en la misma. Los esquemas de inserción y eliminación de la sección 17.1.3 muestran que se requiere además un atributo (**antPI**) que apunte al nodo anterior al del punto de interés. Nótese que disponer del atributo **antPI** permitiría prescindir del atributo **PI**, dado que éste último está accesible a través de la expresión **antPI.siguiente** (véanse los problemas 11 y 12 de este capítulo). Sin embargo, para simplificar la discusión de las operaciones, se ha decidido mantener ambos atributos en esta representación.

```
public class ListaPIIntEnla {
    private NodoInt primero, PI, antPI;
    private int talla;

    ...
}
```

El método constructor crea una lista vacía, es decir, una secuencia nula donde no hay ningún dato almacenado.

```
public ListaPIIntEnla() {
    primero = PI = antPI = null;
    talla = 0;
}
```

Por un lado, llevar el punto de interés al principio de la lista es una simple asignación entre la referencia del nodo cursor y la del primer nodo de la lista.

```
public void inicio() {
    PI = primero;
    antPI = null;
}
```

Por otro lado, desplazar el punto de interés al siguiente nodo respecto del actual requiere que dicho punto de interés no se encuentre todavía al final de la lista.

```
/** El cursor no debe estar al final de la lista */
public void siguiente() {
    antPI = PI;
    PI = PI.siguiente;
}
```

Al insertar un nuevo dato, hay que distinguir si el cursor está al inicio o no. Si el cursor está al inicio, el nuevo nodo será el primero de la lista. En cambio, en cualquier otra situación, éste se situará entre el nodo cursor y su predecesor. En ambos casos, el nuevo nodo será el nuevo nodo predecesor del nodo cursor.

```
public void insertar(int x) {
    if (PI==primero) {
        primero = new NodoInt(x,PI);
        antPI = primero;
    }
    else {
        antPI.siguiente = new NodoInt(x,PI);
        antPI = antPI.siguiente;
    }
    talla++;
}
```

Las operaciones **eliminar** y **recuperar** requieren que el punto de interés no esté al final. Ambas acceden al elemento que se encuentra en el nodo cursor,

devolviéndolo. En caso de `eliminar` el primer dato de la lista, `primero` se actualiza al nodo `siguiente` a éste. En cualquier otro caso, los nodos anterior y posterior al nodo cursor quedan enlazados.

```
/** El cursor no debe estar al final de la lista */
public int eliminar() {
    // PI != null
    int x = PI.dato;
    if (PI==primero) primero = primero.siguiente;
    else antPI.siguiente = PI.siguiente;
    PI = PI.siguiente;
    talla--;
    return x;
}

/** El cursor no debe estar al final de la lista */
public int recuperar() {
    // PI != null
    return PI.dato;
}
```

Finalmente, las consultas sobre si la lista está vacía o no, o sobre cuántos elementos hay almacenados en total, son operaciones que dependen del atributo `talla`, mientras que consultar si el punto de interés está o no al final de la lista se implementa como una comparación entre la referencia `PI` y el valor `null`.

```
public boolean esVacia() {
    return (talla==0);
}

public int talla() {
    return talla;
}

public boolean esFin() {
    return (PI==null);
}
```

Comparación de implementaciones

La complejidad temporal de las operaciones `insertar` y `eliminar` difiere según la implementación: el coste es, en el caso peor, lineal con la talla de la lista en la implementación mediante arrays, mientras que dicho coste es constante en la implementación de nodos enlazados. El resto de operaciones presentan un coste temporal constante en ambas implementaciones.

La representación naïf con arrays presentada en esta sección tiene una alternativa más eficiente, mostrada en la figura 17.16. Esta representación usa dos arrays que funcionan a modo de pilas, de manera que si en la lista $d_0d_1\dots d_{n-1}$ el PI se encuentra sobre un cierto d_i :

- `elArrayIzq[0..antPI]` contiene los elementos de $d_0d_1\dots d_{i-1}$,
- `elArrayDer[0..PI]` contiene los elementos de $d_{n-1}d_{n-2}\dots d_i$.

Con ello, la inserción delante del PI se resuelve con coste $\Theta(1)$ situando el nuevo dato en `elArrayDer[antPI+1]` y actualizando dicho índice `antPI`, mientras que la eliminación del elemento en el PI se resuelve, también con coste $\Theta(1)$, simplemente decrementando el índice PI.

El movimiento del PI al siguiente elemento de la lista se resuelve con el trasvase de `elArrayDer[PI]` a `elArrayIzq[antPI+1]` y la correspondiente actualización de ambos índices; además, se permitiría implementar el desplazamiento al anterior elemento con un trasvase análogo, pero en sentido contrario.

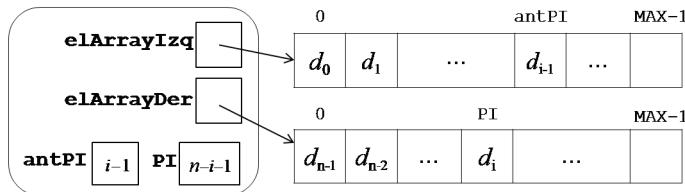


Figura 17.16: Representación mediante dos arrays de una lista con punto de interés.

Cabe señalar que la operación de movimiento del PI de un elemento al anterior también se podría resolver de modo directo, con coste $\Theta(1)$, con una implementación de *secuencias doblemente enlazadas*, en las que los nodos contuviesen referencias a los nodos anterior y siguiente (véase figura 17.17).

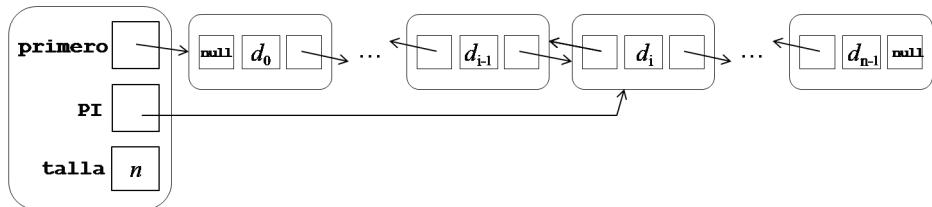


Figura 17.17: Representación mediante una secuencia doblemente enlazada de una lista con punto interés.

En cualquier caso, la discusión sobre el coste espacial es la misma que para las pilas y colas.

17.3 Problemas propuestos

1. Implementar una clase `NodoString` análoga a `NodoInt` excepto que sus datos deben ser de tipo `String`. Diseñar una clase `PalabrasOrd`, cuyos objetos tengan un atributo `sec` de la clase `NodoString`, y un atributo entero `talla` que cuente el número de elementos en `sec`. Los elementos en `sec` se deben mantener ordenados ascendentemente según el orden de `String`.

Teniendo esto en cuenta, hay que implementar los siguientes métodos en la clase `PalabrasOrd`:

- a) `public int talla()` que devuelva el número de elementos en la secuencia.
- b) `public void insertarOrd(String s)` que inserte ordenadamente `s` en la secuencia.
- c) `public boolean eliminar(String s)` que elimine la primera ocurrencia de `s` que aparezca en la secuencia. Si `s` no apareciera en la secuencia se debe devolver `false` y `true` en caso contrario.

La clase se debe incluir en el mismo paquete que contenga la clase `NodoString`, para manejar explícitamente los enlaces de los nodos en la implementación de sus métodos.

2. Escribir una clase `OperacionesEnla` que implemente los siguientes métodos, utilizando explícitamente los enlaces de los nodos:

- a) `public static int maximo(NodoInt sec)` que, siendo `sec` una secuencia con al menos un nodo, busque el máximo elemento de `sec`.
- b) `public static int[] toArray(NodoInt sec)` que devuelva en un array del tamaño justo los elementos de la secuencia `sec`.
- c) `public static NodoInt moverADerecha(NodoInt sec)` que desplace todos los elementos de una secuencia `sec` una posición hacia la derecha. El último elemento deberá pasar a ser el primero. Por ejemplo, si la secuencia es 2, -23, 4, 12, 9, 55, debe retornar 55, 2, -23, 4, 12, 9.
- d) `public static NodoInt moverAIzq(NodoInt sec)` que desplace todos los elementos de una secuencia `sec` una posición hacia la izquierda. El primer elemento deberá pasar a ser el último. Por ejemplo, si la secuencia es 2, -23, 4, 12, 9, 55, debe retornar -23, 4, 12, 9, 55, 2.
- e) `public static NodoInt invertir(NodoInt sec)` que invierta el orden de los elementos de una secuencia `sec` con un coste lineal. Por ejemplo, si la secuencia es 2, -23, 4, 12, 9, 55, debe retornar 55, 9, 12, 4, -23, 2.

- f) `public NodoInt menoresQue(NodoInt sec,int e)` que, con un coste lineal, devuelva una secuencia enlazada con los elementos menores que `e`, y en el mismo orden que aparecen en `sec`. Por ejemplo, si `sec` es `2, -23, 4, 12, 9, 55` y `e` es `10`, debe retornar `2, -23, 4, 9`. El coste deberá ser lineal con la longitud de `sec`.

La clase se debe escribir en un paquete que contenga `NodoInt`.

3. En un paquete que contenga la clase `NodoInt`, escribir una clase `OrdEnla` que, utilizando explícitamente los enlaces de los nodos, implemente los siguientes métodos:

- a) `public static NodoInt insDir(NodoInt sec)` que ordene `sec` mediante una estrategia de inserción directa, es decir, insertando ordenadamente los sucesivos datos de `sec` en la secuencia resultante.
- b) `private static int longitud(NodoInt sec)` que cuente y devuelva el número de elementos de `sec`.
- c) `private static NodoInt mezcla(NodoInt sec1,NodoInt sec2)` que, dadas dos secuencias enlazadas de enteros ordenadas ascendente, devuelva una tercera con la mezcla natural o fusión ordenada de ambas (sección 13.5.1). El coste ha de ser $\Theta(n)$, siendo n el número total de elementos a fusionar.
- d) `private static NodoInt mergesort(NodoInt sec,int n)` que ordene `sec` siguiendo una estrategia recursiva análoga a la del método de ordenación de arrays del mismo nombre (sección 13.4). El segundo parámetro ha de ser la longitud de `sec`.

En el caso general del método, al partir la secuencia enlazada en dos subsecuencias de longitud similar, si el número de elementos es par, las dos subsecuencias tendrán la misma longitud; en caso contrario, la primera subsecuencia se quedará con un nodo más que la segunda. Una vez ordenadas las dos subsecuencias, se usará el método `mezcla` para completar la ordenación.

- e) `public static NodoInt mergesort(NodoInt sec)` que ordene `sec` usando el método privado del mismo nombre.

Realizar el análisis de la complejidad de `insDir` y de `mergesort`.

4. Escribir un método recursivo y otro iterativo para invertir una pila.
5. Escribir un método para calcular la suma de los elementos de una pila de enteros.
6. Escribir un método recursivo que, dada una pila de enteros, obtenga una nueva pila de enteros con los mismos elementos que la original pero cambiados de signo.

7. Se dice que una pila p es *sombrero* de una pila q si todos los elementos de p están en q en el mismo orden y en posiciones más próximas a la cima. Escribir un método recursivo para comprobar si una pila p es *sombrero* de otra pila q .
8. Escribir un método recursivo para transformar una pila en una cola, de forma que el elemento situado en el tope de la pila quede el último en la cola.
9. Escribir un método para mostrar por pantalla el contenido de una cola.
10. La implementación de una cola mediante nodos enlazados (sección 17.2.2) contempla 2 atributos (`primero` y `ultimo`) para referenciar a los nodos que ocupan la primera y la última posición de la cola, respectivamente. Implementar una versión alternativa que sólo tenga 1 atributo referencia, en este caso al último nodo de la cola, mediante la definición de una secuencia circular, es decir, haciendo que el `siguiente` del `ultimo` nodo de la secuencia sea el `primero` de ésta, en lugar de `null` como es habitual. Hacerlo de modo que el coste de todas las operaciones permanezca en $\Theta(1)$.
11. La implementación enlazada de listas con punto de interés (sección 17.2.3) contempla dos atributos (`PI` y `antPI`) para referenciar al nodo en el punto de interés y a su predecesor, respectivamente. Ya en dicha sección, se dice que la referencia `PI` es prescindible dado que dicho nodo es accesible como el `siguiente` del nodo `antPI`. Implementar una versión alternativa que use únicamente la referencia `antPI`, y de manera que el coste de todas las operaciones se mantenga constante.

12. Una técnica conocida de estructuración de listas con punto de interés que facilita la implementación de las operaciones del problema anterior (manteniendo una sola referencia `antPI`), consiste en usar un nodo ficticio al principio de la lista, que no albergaría ningún elemento de la lista. Con ello se consigue que cualquier inserción o eliminación, incluso al inicio de la lista, se realice a continuación de un nodo preexistente. La construcción de la lista vacía se escribiría entonces como:

```
public ListaPIIntEnla() {
    primero = new NodoInt(0);
    antPI = primero;
    talla = 0;
}
```

Se pide completar la implementación del resto de operaciones de la clase basándose en esta representación.

13. Se supone una palabra representada respectivamente como una pila y como una cola de caracteres. Escribir para ambas representaciones una método `capicua(p)`, tal que devuelva `true` si p es una palabra capicúa y `false` en

caso contrario. Comparar la complejidad del método en ambas representaciones.

14. La siguiente función invierte una cola:

```
public static void invertirCola(Cola c) {  
    if (!c.esVacia()) {  
        int x = c.desencolar();  
        invertirCola(c);  
        c.encolar(x);  
    }  
}
```

Estudiar su complejidad temporal en los siguientes casos:

- La implementación de la cola se ha realizado mediante una representación enlazada.
- La implementación se ha realizado mediante una representación con arrays no circular.
- La implementación se ha realizado mediante una representación con arrays circular.

Comparar el coste del algoritmo en las tres representaciones propuestas.

15. Añadir las siguientes operaciones a la clase `ListaPIIntEnla`:

- a) `public void borrar(x)` que borre el elemento `x` de la lista. Si `x` aparece más de una vez, borra la primera aparición. Si `x` no se encuentra en la lista, no hace nada.
- b) `public void anterior()` que cambie el punto de interés a la posición anterior a la actual en la lista. La operación no está definida si el punto de interés es el primer elemento de la lista.
- c) `private boolean buscar(NodoInt ant, int x)` que busque la primera ocurrencia de `x` desde el nodo siguiente a `ant` en adelante; si lo encuentra, mueve el punto de interés al nodo que contiene a `x`.
- d) `public boolean buscarInicio(int x)` que, invocando al método privado `buscar` con los parámetros adecuados, indique si `x` aparece en la lista. Si `x` se encuentra, sitúa el punto de interés en la primera ocurrencia de `x`. Si no aparece, el punto de interés no se mueve.
- e) `public boolean buscarSiguiente(int x)` que, invocando al método privado `buscar` con los parámetros adecuados, indique si `x` aparece en la lista desde la posición del punto de interés inclusive en adelante. Si `x` aparece, avanza el punto de interés a `x`. Si no aparece, el punto de interés no se mueve.

Estudiar su complejidad temporal.

16. Escribir un método que, dadas dos listas con punto de interés 11 y 12, devuelva como resultado otra lista 13 que contenga sólo aquellos elementos de 11 que no estén en 12 y los de 12 que no estén en 11. Sólo se permite utilizar las operaciones definidas para el tipo.

Más información

- [AGH01] K. Arnold, J. Gosling, and D. Holmes. *El lenguaje de programación Java*. Addison-Wesley, 2001. Capítulo 21.
- [Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011. URL: <http://math.hws.edu/javanotes/>. Capítulo 9 (9.2 y 9.3).
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010. Capítulo 15.
- [Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulos 6, 15 y 16.
- [Wir82] N. Wirth. *Algoritmos + Estructuras de Datos = Programas*. Ediciones del Castillo, 1982. Capítulo 4 (4.1 y 4.2).

Bibliografía

- [AGH01] K. Arnold, J. Gosling y D. Holmes. *El lenguaje de programación Java*. Addison-Wesley, 2001 (vid. págs. 125, 502).
- [BB97] G. Brassard y P. Bratley. *Fundamentos de Algoritmia*. Prentice Hall, 1997 (vid. págs. 337, 348, 369).
- [BK07] D.J. Barnes y M. Kölling. *Programación Orientada a Objetos con Java: una introducción práctica usando BlueJ*. Pearson Educación, 2007 (vid. pág. 37).
- [CGo03] J. Carretero, F. García y otros. *Problemas resueltos de programación en lenguaje Java*. Thomson, 2003 (vid. pág. 66).
- [Eck11] D.J. Eck. *Introduction to Programming Using Java, Sixth Edition*. 2011. URL: <http://math.hws.edu/javanotes/> (vid. págs. 66, 84, 163, 193, 220, 270, 401, 423, 459, 502).
- [FAM98] F.J. Ferri, J.V. Albert y G. Martín. *Introducció a l'anàlisi i disseny d'algorismes*. Universitat de València, 1998 (vid. págs. 348, 369).
- [For03] B.A. Forouzan. *Introducción a la Ciencia de la Computación, de la manipulación de datos a la teoría de la computación*. Thomson, 2003 (vid. págs. 45, 66).
- [GG05] D. Gries y P. Gries. *Multimedia Introduction to Programming Using Java*. Springer, 2005 (vid. pág. 220).
- [Han99] P.B. Hansen. *Programming for everyone in Java*. Springer, 1999 (vid. pág. 303).

- [Ora11a] Oracle. *How to Write Doc Comments for the Javadoc Tool*. 2011. URL: <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html> (vid. págs. 32, 37, 117, 125).
- [Ora11b] Oracle. *Javadoc Tool*. 2011. URL: <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html> (vid. págs. 31, 37, 115, 125).
- [Ora11c] Oracle. *JavaTM Platform, Standard Edition 6, API Specification*. 2011. URL: <http://download.oracle.com/javase/6/docs/api/> (vid. págs. 27, 37, 95, 128, 129, 132, 135, 145, 155, 163, 405, 409, 429, 441, 442, 444).
- [Ora11d] Oracle. *The JavaTM Tutorials*. 2011. URL: <http://download.oracle.com/javase/tutorial/> (vid. págs. 66, 84, 125, 145, 193, 220, 233, 270, 401, 423, 459).
- [Pyl75] Z.W. (selec.) Pylyshyn. *Perspectivas de la revolución de los computadores/Selec., comentarios e introd. de Z.W. Pylyshyn; tr. por Luis García Llorente; rev. de Eva Sánchez*. Incluye textos de H. Aiken, Ch. Babbage, J. von Neumann, C. Shannon, A.M. Turing y otros. Alianza, 1975 (vid. pág. 14).
- [Sav10] W.J. Savitch. *Absolute Java, Fourth Edition*. Pearson Education, 2010 (vid. págs. 84, 125, 145, 163, 193, 220, 270, 303, 401, 423, 459, 502).
- [Sch07] H. Schildt. *Fundamentos de Java*. McGraw-Hill, 2007 (vid. págs. 37, 125, 303, 459).
- [Tra77] B.A. Trajtenbrot. *Los algoritmos y la resolución automática de problemas*. MIR, 1977 (vid. pág. 14).
- [Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000 (vid. págs. 303, 348, 369, 401, 502).
- [Wir82] N. Wirth. *Algoritmos + Estructuras de Datos = Programas*. Ediciones del Castillo, 1982 (vid. pág. 502).
- [WP00] R. Wiener y L.J. Pinson. *Fundamentals of OOP and data structures in Java*. Cambridge University Press, 2000 (vid. pág. 423).

Índice de figuras

1.1.	Algoritmo 1 para determinar si n es primo.	3
1.2.	Algoritmo 2 para determinar si n es primo.	3
1.3.	Algoritmo 3 para determinar si n es primo.	3
1.4.	Dos algoritmos que permiten escribir los elementos repetidos de una lista.	4
1.5.	Estructura de un procesador con arquitectura Von Neumann.	6
1.6.	Proceso de compilación y ejecución de un programa.	7
1.7.	Proceso de compilación y ejecución de un programa en Java.	10
1.8.	¿Es n primo? Algoritmo 3, versión en Pascal.	11
1.9.	¿Es n primo? Algoritmo 3, versión en C/C++.	12
1.10.	¿Es n primo? Algoritmo 3, versión en Python.	12
1.11.	¿Es n primo? Algoritmo 3, versión en Java.	12
1.12.	¿Es n primo? Algoritmo 3, versión en C#.	13
2.1.	Clase <code>Circulo</code>	17
2.2.	Parte de la documentación de la clase <code>Pizarra</code>	18
2.3.	Clase <code>PrimerPrograma</code>	19
2.4.	Ejecución de la clase <code>PrimerPrograma</code>	20
2.5.	El programa <code>HolaATodos</code>	28

2.6.	Parte de la documentación de la clase Circulo	33
2.7.	Clase Punto	34
2.8.	Clase Cuadrado (incompleta).	36
3.1.	Cambio de estados al intercambiar el valor de dos variables. . . .	43
3.2.	Compatibilidad de tipos básicos.	47
4.1.	Situación de la memoria del sistema tras la creación de un objeto de tipo Circulo y su asignación a la variable circulo	73
4.2.	Las dos variables de tipo Punto , p1 y p2 , referencian al mismo objeto.	75
4.3.	Las variables p1 y p2 referencian al mismo Punto , sin embargo el segundo Punto creado queda desreferenciado.	76
4.4.	Las variables c1 y copia referencian dos Círculo distintos pero con los mismos valores.	77
5.1.	Clase ProgramaTri	87
5.2.	Extracto de la documentación de String	89
5.3.	Extracto de la documentación de String : constructores.	89
5.4.	Extracto de la documentación de Math	89
5.5.	Extracto de la documentación de System	89
5.6.	Clase PruebaMetodos	101
5.7.	Evolución de la ejecución de Prueba y la pila de llamadas. . . .	104
5.8.	Ejecución de la llamada intercambio(p.x,p.y)	106
5.9.	Ejecución de la llamada p.intercambioX(q)	107
5.10.	Clase Punto (i).	110
5.10.	Clase Punto (ii).	111
5.11.	Ejemplo de uso de la clase Punto	112
5.12.	Clase UtilPunto (i).	116

5.12.	Clase UtilPunto (ii).	117
5.13.	Documentación de la clase Punto generada por javadoc	119
5.14.	Clase ProgramaTri . Versión 2.	120
6.1.	Las variables exp1 , exp2 y exp3 referencian tres String distintos. El String "Ejemplo2" queda desreferenciado.	129
6.2.	Numeración de los caracteres de un String	132
6.3.	Redondeo de un valor a cierto número de decimales.	137
6.4.	Obtención de un valor aleatorio en el intervalo $[a, b]$	138
7.1.	Representación de los flujos de entrada y salida.	147
7.2.	Ejemplo de uso de la clase Scanner	154
7.3.	Ejemplo de posible problema con la clase Scanner	157
7.4.	Ejemplo de uso de la clase Locale con la clase Scanner	158
7.5.	Ejemplo de lectura de un dato de tipo char con Scanner	159
7.6.	Ejemplo de uso de la clase Scanner para lectura desde fichero.	160
8.1.	Diagramas de flujo de las instrucciones condicionales.	167
8.2.	Ejemplo de uso de la instrucción if...else... múltiple.	172
8.3.	Ejemplo de uso de la instrucción switch	175
8.4.	Ejemplo de uso de la instrucción switch con un String	177
8.5.	Clase Fecha	181
8.6.	Clase TestFecha	183
9.1.	Diagrama de flujo de la instrucción while	197
9.2.	Diagrama de flujo de la instrucción for	207
9.3.	Diagrama de flujo de la instrucción do ... while	210
10.1.	Reserva de memoria en un array.	224

10.2.	Índices de las componentes de un array.	225
10.3.	Inicialización de arrays.	226
10.4.	Ejemplo de arrays como argumentos, parámetros y resultados de un método.	229
10.5.	Estados de la memoria al ejecutar la clase <code>PasoParam</code>	230
10.6.	Ejemplo de copia de arrays de tipo simple y tipo referencia. . . .	231
10.7.	Representación de los datos del año. Primera versión.	232
10.8.	Representación de los datos del año. Segunda versión.	232
10.9.	Representación de una matriz bidimensional.	233
10.10.	Declaración e inicialización de una matriz bidimensional.	234
10.11.	Representación habitual de una matriz bidimensional.	236
10.12.	Estado de las variables del bucle en la ejecución de un recorrido. .	239
10.13.	Estado de las variables del bucle en la ejecución de una búsqueda. .	244
10.14.	Búsqueda binaria.	251
10.15.	Representación de un conjunto de naturales.	252
10.16.	Clase <code>Conjunto</code>	253
10.17.	Moda de un multiconjunto de naturales.	254
10.18.	Representación de una lista de elementos.	254
10.19.	Clase <code>SecuenciaDeCirculos</code> (i).	256
10.19.	Clase <code>SecuenciaDeCirculos</code> (ii).	257
10.19.	Clase <code>SecuenciaDeCirculos</code> (iii).	258
11.1.	Secuencia de llamadas del método recursivo <code>factorial</code>	277
11.2.	Evolución de la ejecución y la pila de llamadas.	279
11.3.	Árbol de llamadas del método recursivo <code>fibonacci</code>	284
11.4.	Descomposición recursiva de un array <code>a</code>	286

12.1.	Velocidad de búsqueda según la estrategia utilizada.	306
12.2.	Comportamiento asintótico de cuatro funciones típicas para valores de x hasta 100.	315
12.3.	Comportamiento asintótico de cuatro funciones típicas para valores de x hasta 10000.	316
12.4.	Ejemplo de como la función $\frac{3}{2}x^2 + 20x + 50$ queda acotada inferiormente por x^2 y superiormente por $2x^2$	318
13.1.	Selección del i-ésimo mínimo del array.	351
13.2.	Inserción del elemento i en el subarray ordenado.	354
13.3.	Intercambio del elemento j del array.	357
13.4.	Ordenación por mergesort.	359
13.5.	Traza del método <code>mezclaNatural</code>	361
14.1.	Ejemplo de jerarquía de clases.	373
14.2.	Ejemplo de jerarquía de las librerías de Java.	374
14.3.	Implementación de la jerarquía de figuras.	378
14.4.	Sobrescritura del método <code>paintComponent</code>	381
14.5.	Ejemplo de escalado.	382
14.6.	Sobrescritura del método <code>toString</code>	385
14.7.	Documentación de la clase <code>Double</code>	394
15.1.	Jerarquía <code>Throwable</code> en <code>java.lang</code>	406
15.2.	Jerarquía <code>Throwable</code> en <code>java.io</code>	407
15.3.	Jerarquía <code>Throwable</code> en <code>java.util</code>	408
15.4.	Documentación del método <code>nextDouble()</code> en <code>Scanner</code>	409
15.5.	Clase <code>Mediciones</code>	411
15.6.	Clase <code>LecturaEnteroPositivo</code>	417

16.1.	Ejemplo de fichero de acceso secuencial que incluye varios registros de una agenda telefónica.	426
16.2.	Ejemplo de uso de la clase <code>File</code>	428
16.3.	Ejemplo de uso de <code>PrintWriter</code> para la escritura en ficheros de texto.	430
16.4.	Ejemplo de uso de <code>Scanner</code> para leer de un fichero de texto. . . .	433
16.5.	Ejemplo de uso de <code>Scanner</code> para leer de un fichero de texto con detección de terminación de fichero.	434
16.6.	Ejemplo de escritura y posterior lectura de un fichero binario. . .	437
16.7.	Ejemplo de lectura de un fichero binario.	438
16.8.	Ejemplo de lectura y escritura en fichero de acceso aleatorio. . . .	440
16.9.	Jerarquía de clases a partir de <code>InputStream</code> (flujos binarios). . .	442
16.10.	Ejemplo de lectura a partir de una URL.	443
16.11.	Jerarquía de clases a partir de <code>Writer</code> (flujos de caracteres). . . .	444
16.12.	E/S de objetos en ficheros binarios.	445
16.13.	Ejemplo de clase que implementa la interfaz <code>Serializable</code>	445
16.14.	Ejemplo de uso de E/S de objetos.	446
16.15.	La clase <code>ItemAgenda</code> abreviada.	447
16.16.	La clase <code>Agenda</code> abreviada.	448
16.17.	Clase <code>GestorPrueba</code> . Creación, almacenamiento y recuperación de una agenda.	449
16.18.	Ejemplo de tratamiento de fin de fichero (<code>EOFException</code>). . . .	451
17.1.	Representación enlazada de una secuencia.	462
17.2.	Estructura de un nodo.	463
17.3.	Secuencia enlazada de 0 o más datos.	463
17.4.	Los dos constructores de <code>NodoInt</code>	464
17.5.	Formación de una secuencia enlazada de tres elementos. . . .	465

17.6. Uso explícito del enlace siguiente en una secuencia enlazada.	466
17.7. Formación de una secuencia enlazada de tres elementos.	467
17.8. Actualización de sec1 por el método norma90	468
17.9. Recorridos de arrays y secuencias enlazadas.	469
17.10. Inserción en cabeza.	472
17.11. Inserción detrás de un nodo.	473
17.12. Eliminación de un nodo.	475
17.13. Pila de registros de activación.	478
17.14. Cola de impresión.	483
17.15. Listas con cursor o punto de interés.	490
17.16. Representación mediante dos arrays de una lista con punto de interés.	496
17.17. Representación mediante una secuencia doblemente enlazada de una lista con punto interés.	496

Índice de tablas

3.1.	Palabras reservadas	45
3.2.	Tipos de números enteros.	45
3.3.	Tipos reales.	47
3.4.	Operadores aritméticos.	49
3.5.	Traza de ejecución de incrementos y decrementos en uno.	53
3.6.	Codificación <i>ASCII</i> (7 bits), 128 primeros caracteres Unicode. . .	56
3.7.	Secuencias de escape.	58
3.8.	Operadores relacionales.	59
3.9.	Operadores lógicos.	60
3.10.	Significado de los operadores lógicos.	60
3.11.	Precedencia de los operadores.	61
6.1.	Métodos <code>equals</code> y <code>compareTo</code> de la clase <code>String</code>	131
6.2.	Algunos métodos de la clase <code>String</code>	133
6.3.	Constantes públicas de la clase <code>Math</code>	135
6.4.	Algunos de los métodos matemáticos básicos en <code>Math</code>	135
6.5.	Método para obtener un valor aleatorio en <code>Math</code>	135
6.6.	Algunos de los métodos exponenciales y logarítmicos en <code>Math</code> . .	136
6.7.	Algunos de los métodos trigonométricos en <code>Math</code>	136

6.8.	Métodos de las clases envolventes numéricas que transforman de String a tipo básico.	140
6.9.	Algunos métodos de la clase Character	141
7.1.	Principales constructores y métodos de la clase Scanner	156
12.1.	Algunas funciones típicas de coste.	316
12.2.	Multiplicación a la rusa [BB97].	337
12.3.	Tiempo (<i>usegs</i>) según tipo de multiplicación.	340
14.1.	Las clases Circulo y Rectangulo	372
14.2.	Relación de herencia y atributos.	373
16.1.	Principales constructores y métodos de la clase File	429
16.2.	Principales constructores y métodos de la clase PrintWriter . . .	432
16.3.	Principales constructores y métodos de la clase Scanner	435
17.1.	Interfaz de la clase Pila	478
17.2.	Interfaz de la clase Cola	484
17.3.	Interfaz de la clase Lista	491