

2.ª EDICIÓN REVISADA
Y ACTUALIZADA

Aprende a programar con Java

Un enfoque práctico partiendo de cero

ALFONSO JIMÉNEZ MARÍN • FRANCISCO MANUEL PÉREZ MONTES



INCLUYE
GRAN NÚMERO DE
EJERCICIOS
RESUELTOS

Aprende a programar con Java

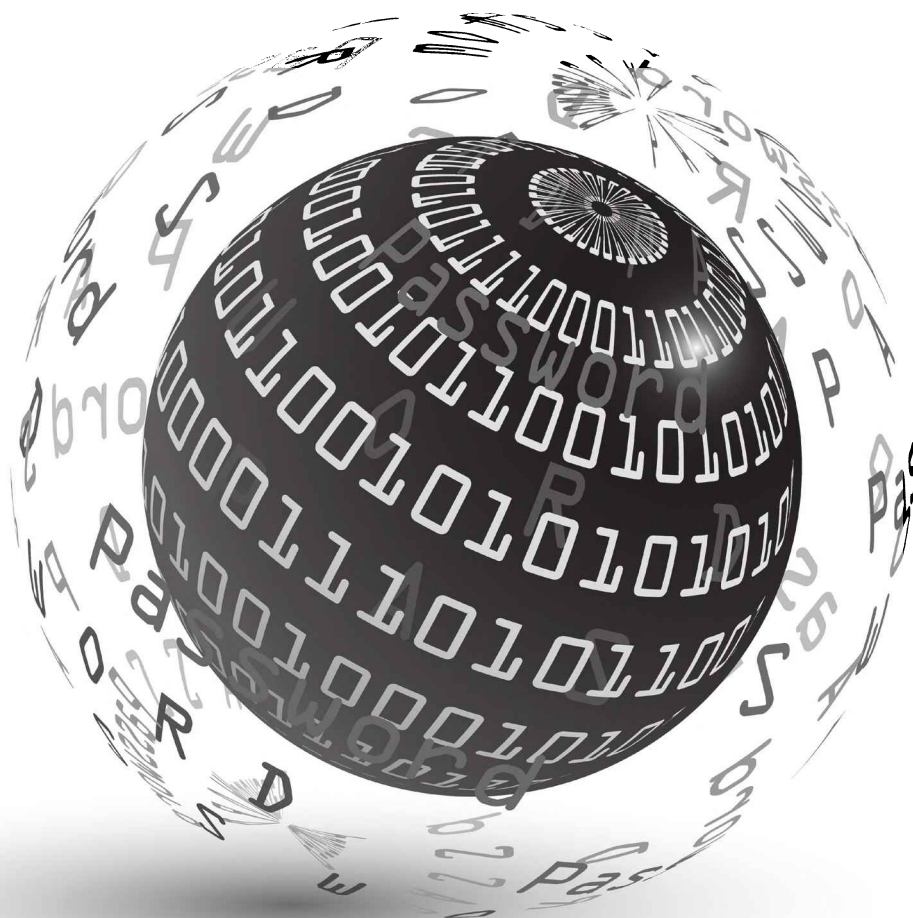
Un enfoque práctico partiendo de cero

2.ª EDICIÓN REVISADA
Y ACTUALIZADA

Aprende a programar con Java

Un enfoque práctico partiendo de cero

ALFONSO JIMÉNEZ MARÍN • FRANCISCO MANUEL PÉREZ MONTES



Paraninfo

APRENDE A PROGRAMAR CON JAVA.

Un enfoque práctico partiendo de cero

© Alfonso Jiménez Marín y Francisco Manuel Pérez Montes

Gerente Editorial:

María José López Raso

Equipo Técnico Editorial:

Alicia Cerviño González

Paola Paz Otero

Editora de Adquisiciones:

Carmen Lara Carmona

Producción:

Nacho Cabal

Preimpresión:

Francisco Manuel Pérez Montes

Diseño de cubierta:

Ediciones Nobel

Impresión:

Gráficas Summa (Llanera, Asturias)

COPYRIGHT

© 2016 Ediciones Paraninfo

2.ª edición revisada y actualizada

Paraninfo, S.A.

C/ Velázquez, 31, 3.ª dcha. / 28001 Madrid, ESPAÑA

Teléfono: 902 995 240 / Fax: 914 456 218

clientes@paraninfo.es / www.paraninfo.es

ISBN: 978-84-283-3857-8

Depósito legal: M-6361-2016

(13892)

Impreso en España / Printed in Spain

Reservados los derechos para todos los países de lengua española. De conformidad con lo dispuesto en el artículo 270 del Código Penal vigente, podrán ser castigados con penas de multa y privación de libertad quienes reprodujeran o plagiaran, en todo o en parte, una obra literaria, artística o científica fijada en cualquier tipo de soporte sin la preceptiva autorización. Ninguna parte de esta publicación, incluido el diseño de la cubierta, puede ser reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea este electrónico, químico, mecánico, electro-óptico, grabación, fotocopia o cualquier otro, sin la previa autorización escrita por parte de la Editorial.

*A mi mujer, Pepita.
Por su implementación de la interfaz Cloneable,
por su implementación del método main()
y por el control, con una sonrisa franca, de las subclases de Exception.*

Franma

A mis hijos, Ana y Héctor.

Alfonso

Prólogo a la 2.^a edición

Estimado lector:

Para los autores es una gran alegría que haya visto la luz la segunda edición de esta obra. Por una parte, es señal de que muchas personas han usado el libro para aprender programación y para aprender Java. Y por otra parte, es una nueva oportunidad que se nos brinda de poner en práctica nuestra metodología de enseñanza y de aprendizaje. Esta se resume en «*el mejor aprendizaje es la práctica*». Práctica que debe que ser poco a poco, sin prisas y asimilando los conceptos de uno en uno, en pequeños pasos.

No cabe duda de que existen estupendos textos sobre algorítmica, programación y lenguaje Java pero, en general, comienzan por la programación orientada a objetos, que nosotros no abordamos hasta la mitad del libro, ya que pensamos que antes el lector necesita comprender y asimilar un buen número de conceptos y, sobre todo, de procedimientos. Ellos son el mecanismo interno que da vida a los objetos y permite al lector ver los frutos de su esfuerzo desde el primer momento. Por otra parte, hemos intentado que la curva de aprendizaje sea continua y gradual, evitando saltos, para que en ningún momento se sienta perdido.

También queremos mencionar a nuestros alumnos, con los que mantenemos una fructífera simbiosis de aprendizaje ya que, mientras ellos aprenden a programar, nosotros aprendemos a mejorar nuestra metodología de la enseñanza de la programación. Muchos de los cambios incluidos en esta edición están inspirados en el trabajo diario con ellos que, no pocas veces nos sugieren soluciones alternativas a los problemas planteados.

En esta segunda edición, una vez pasados los primeros capítulos dedicados a los fundamentos, hemos tendido a realizar una programación de más alto nivel, de mayor abstracción. Eso no significa una mayor dificultad, sino todo lo contrario, ya que implica el uso de herramientas muy intuitivas disponibles en Java, que facilitan enormemente el trabajo de los programadores, incluso los profesionales, evitándoles tareas pesadas e ingratas en aplicaciones complejas. Tratamos casi todas las novedades de la última versión de Java. Hemos suprimido un capítulo completo que resultaba obsoleto e incluido otro totalmente nuevo con conceptos recién incorporados al lenguaje. Asimismo, hemos rehecho y actualizado casi todos los demás capítulos y anexos y prescindido de alguna herramienta propia, buscando la máxima universalidad en el código del libro.

El principal objetivo de esta edición es dotar al lector de los conocimientos y habilidades necesarias para que luego, de forma autodidacta, pueda continuar su formación con otros libros, artículos, documentación oficial de Java o en los distintos foros de internet. Todo ello con la ayuda de abundantes ejemplos y ejercicios resueltos. Esto lo hace adecuado, tanto para un aprendizaje autodidacta, como para su uso como libro de apoyo en facultades, escuelas técnicas y ciclos superiores de formación profesional.

Los ejercicios se plantean con dificultad creciente y, en la resolución de muchos de ellos se puede reutilizar la solución de ejercicios anteriores. Recomendamos al lector realizar por su cuenta todos los ejercicios que vienen resueltos en el libro. Cuando más se aprende en programación es cuando se hacen las cosas, aunque sea mal. Aquí uno puede aprender de sus propios errores más que en ningún otro sitio, ya que siempre hay otra oportunidad.

Un programador experto seguramente encontrará soluciones más eficientes y elegantes que las planteadas aquí. Hemos elegido las mejores soluciones desde un punto de vista didáctico. Para nosotros, la mejor solución es la que mejor se entiende y, por tanto, la que más nos hace aprender.

A lo largo del libro se han utilizado algunas convenciones que ayudarán al lector a una mejor comprensión y lectura,

- **Letra monoespaciada** para fragmentos de código en Java.
- Aunque no es necesario hacerlo, más por costumbre que por razones de compatibilidad, hemos eliminado cualquier vocal acentuada o ñe del código, utilizándose solo en los comentarios o en los literales de tipo cadena.
- La gran mayoría de los ejercicios funcionan con versiones antiguas de Java, pero los ejercicios de los últimos capítulos requieren tener instalada la versión 8.
- En ciertos ejercicios resueltos, la solución difiere solo unas líneas con respecto a otro ejercicio en el que se basa. Con la intención de no repetir líneas de código, estas se omiten indicándolo mediante tres puntos suspensivos (...) Los símbolos ... no pertenecen al lenguaje Java, y no pueden escribirse literalmente en un programa.

Por último, esperamos que el lector disfrute y aprenda tanto leyendo esta obra como nosotros escribiéndola.

Sevilla, enero de 2016

Los autores

Prólogo a la 1.^a edición

Estimado lector:

El libro que tienes en tus manos ha ido tomando forma durante bastante tiempo, y es fruto de años de docencia en disciplinas tan íntimamente ligadas como la algorítmica, la metodología de la programación y la enseñanza de lenguajes de programación. Durante este proceso hemos observado la evolución de nuestros alumnos; pedimos disculpas por haberlos utilizado como conejillos de indias. Pero, en nuestra defensa, podemos decir que todos ellos han resultado ilesos. El observar la trayectoria de nuestros alumnos, no solo en un curso académico, sino a través de los años, nos ha servido para intentar mejorar nuestros métodos docentes y buscar una mejor manera de enseñar (y aprender). Todo lo visto puede sintetizarse en el consejo que solemos darles: «*el mejor aprendizaje es la práctica*», lo que resume la filosofía de este libro. Para aprender a programar, aparte de conocer algunos conceptos básicos, lo mejor es practicar mucho. Quizás, la primera vez necesitemos que nos expliquen detalladamente un algoritmo. Conforme avancemos, seremos cada vez más autónomos, necesitaremos acudir menos veces a la solución propuesta y, en el último estadio, implementaremos nuestras propias soluciones. Además, está contrastado que, como en casi todos los ámbitos de la vida, en programación se aprende más de los errores propios que de los aciertos. Así que esperamos que os equivoquéis mucho.

El principal objetivo de la obra es proporcionar un material que sienta las bases y proporcione los rudimentos de la programación, a la vez que dotar al lector del conocimiento de un lenguaje de propósito general como **Java**. Todo ello acompañado de abundantes ejemplos y ejercicios resueltos. Esto lo hace adecuado tanto para un aprendizaje autodidacta, como para su uso como libro de apoyo en facultades, escuelas técnicas y ciclos superiores de programación de formación profesional.

El contenido del libro está estructurado en capítulos, en cada uno de los cuales se trata un área o técnica de programación, en lenguaje **Java**. Cada capítulo está formado por tres partes bien diferenciadas: una introducción teórica, documentada con ejemplos; una segunda parte formada por una colección de ejercicios resueltos; y, por último, algunos ejercicios propuestos que pueden resolverse utilizando todos los conceptos vistos en el capítulo. Pensamos que en los ejercicios resueltos está la verdadera fuerza del libro, ya que guían al lector con numerosos comentarios y, a menudo, plantean posibles alternativas. Según aumente su destreza programando, el lector recurrirá cada vez menos a las soluciones propuestas

e implementará sus propias soluciones, aunque siempre se aconseja examinar la solución propuesta ya que, a menudo, se usa como vehículo para mostrar la forma más conveniente de abordar determinadas tareas.

Los ejercicios resueltos se plantean con dificultad creciente y, en la resolución de muchos de ellos, se puede aprovechar el código empleado para resolver los anteriores. Recomendamos realizar todos los ejercicios, ya que en cada uno de ellos se puede encontrar algún elemento que enriquece el conocimiento y la destreza del lector.

Un programador experto seguramente encontrará soluciones más eficientes y elegantes que las planteadas aquí. Hemos elegido las mejores soluciones desde un punto de vista didáctico. Para nosotros, la mejor solución ha sido la que mejor se entiende y, por tanto, la que nos hace aprender más. Un comentario aparte merece el **Capítulo 12**, que versa sobre listas enlazadas. Las listas son una excelente herramienta para manejar distintos tipos de datos, y vienen implementadas «de serie» en casi todos los lenguajes de programación, librando al programador de esa penosa tarea. Pero aquí hemos decidido diseñar nuestras propias listas como un estupendo ejercicio práctico de diseño de una clase moderadamente compleja que, además, ayuda a entender su estructura interna y nos familiariza con su funcionamiento. Podríamos haberla diseñado con tipos genéricos, pero eso nos habría obligado a salirnos del alcance propuesto para este libro y habría aumentado su longitud.

Tenemos la obligación de avisar al lector de que la lectura de cualquier ejercicio puede ser más o menos liviana, pero su comprensión puede requerir algo más de tiempo. Aprender a programar, o lo que es lo mismo, comprender este libro, es un proceso que requiere dedicación y tiempo. Es por esto por lo que se aconseja al lector avanzar despacio pero seguro.

A lo largo del libro se han utilizado algunas convenciones que ayudarán al lector a una mejor comprensión y lectura,

- *Letra cursiva* para enfatizar las *palabras reservadas* que forman parte de **Java**.
- **Letra monoespaciada** para hacer referencia a cualquier **identificador**, **sentencia**, o **fragmento de código**.
- Por razones de compatibilidad, hemos eliminado cualquier vocal acentuada o ñe del código, utilizándose solo en los comentarios o en los literales cadena.

Por último, esperamos que el lector disfrute y aprenda tanto con esta obra, como nosotros escribiéndola.

Sevilla, diciembre de 2011

Los autores

Índice

1. Conceptos básicos	1
1.1. Algoritmo	2
1.2. Lenguajes de programación	3
1.3. ¿Cuál es el propósito de este libro?	4
1.4. NetBeans IDE	4
1.5. El programa principal	7
1.6. Palabras reservadas	7
1.7. Variables	8
1.7.1. Identificadores	8
1.8. Tipos	9
1.8.1. Rangos	10
1.9. Constantes	11
1.10. Comentarios	12
1.11. Operaciones básicas	12
1.11.1. Operador de asignación	12
1.11.2. Operadores aritméticos	13
1.11.3. Operadores relacionales	14
1.11.4. Operadores lógicos	15
1.11.5. Operadores «opera y asigna»	16
1.11.6. Operador ternario	17
1.11.7. Precedencia	17
1.12. Conversión de tipos	18
1.13. API de Java	19
1.13.1. Salida por consola	21
1.13.2. Entrada de datos	23
Ejercicios de conceptos básicos	25
Ejercicios propuestos	31
 2. Condicionales	 33
2.1. Expresiones lógicas	33
2.1.1. Operadores relacionales	33
2.1.2. Operadores lógicos	34

2.2. Condicional simple: if	36
2.3. Condicional doble: if-else	38
2.3.1. Operador ternario	39
2.3.2. Anidación de condicionales	39
2.4. Condicional múltiple: switch	40
Ejercicios de condicionales	43
Ejercicios propuestos	55
3. Bucles	57
3.1. Bucles controlados por condición	57
3.1.1. while	57
3.1.2. do-while	59
3.2. Bucles controlados por contador: for	59
3.3. Salidas anticipadas	61
3.4. Bucles anidados	62
3.4.1. Bucles independientes	63
3.4.2. Bucles dependientes	64
Ejercicios de bucles	65
Ejercicios propuestos	74
4. Funciones	75
4.1. Conceptos básicos	75
4.2. Ámbito de las variables	78
4.3. Paso de información a una función	78
4.3.1. Valores en la llamada	79
4.3.2. Parámetros de entrada	79
4.4. Valor devuelto por una función	81
4.5. Sobrecarga de funciones	82
4.6. Recursividad	83
Ejercicios de funciones	87
Ejercicios propuestos	96
5. Tablas	97
5.1. Variables escalares <i>vs</i> tablas	97
5.2. Índices	98
5.2.1. Índices fuera de rango	98
5.3. Construcción de tablas	99
5.3.1. Tamaño y tipo	99
5.3.2. Variables de tabla	99
5.3.3. Operador new	100
5.3.4. Construcción mediante asignación	100
5.4. Referencias	100
5.4.1. Recolector de basura	104
5.4.2. null	104
5.5. Clasificación de tablas	105

5.5.1.	Según el número de elementos con datos	105
5.5.2.	Según el orden	106
5.6.	Tablas como parámetros de funciones	107
5.7.	Clase Arrays	107
5.8.	Operaciones con tablas	108
5.8.1.	Obtención del tamaño	108
5.8.2.	Recorrido	108
5.8.3.	Mostrar una tabla	109
5.8.4.	Inicialización	110
5.8.5.	Comparación de dos tablas	110
5.8.6.	Búsqueda	111
5.8.7.	Inserción	113
5.8.8.	Borrado	114
5.8.9.	Ordenación	115
5.8.10.	Copia	116
5.9.	Tablas n -dimensionales	117
5.9.1.	Matrices bidimensionales	117
5.9.2.	Matrices tridimensionales	118
5.9.3.	Matrices n -dimensionales	118
	Ejercicios de tablas	120
	Ejercicios propuestos	134

6. Cadenas 137

6.1.	Tipo primitivo char	137
6.1.1.	<i>Unicode</i>	137
6.1.2.	Secuencias de escape	138
6.1.3.	Conversión char ↔ int	139
6.1.4.	Aritmética de caracteres	140
6.2.	Clase Character	141
6.2.1.	Clasificación de caracteres	141
6.2.2.	Conversión	143
6.3.	Clase String	144
6.3.1.	Inicialización de cadenas	144
6.3.2.	Comparación	145
6.3.3.	Concatenación	147
6.3.4.	Obtención de caracteres	148
6.3.5.	Longitud de una cadena	149
6.3.6.	Búsqueda	150
6.3.7.	Comprobaciones	151
6.3.8.	Conversión	153
6.4.	Cadenas y tablas de caracteres	153
	Ejercicios de cadenas	155
	Ejercicios propuestos	167

7. Clases	169
7.1. Crear una clase desde NetBeans	170
7.2. Atributos	171
7.2.1. Inicialización	171
7.3. Objetos	172
7.3.1. Referencia	172
7.3.2. Variables referencia	173
7.3.3. Operador <code>new</code>	173
7.3.4. Referencia <code>null</code>	175
7.3.5. Recolector de basura	175
7.4. Métodos	176
7.4.1. Ámbito de las variables y atributos	177
7.4.2. Ocultación de atributos	178
7.4.3. Objeto <code>this</code>	179
7.5. Atributos y métodos estáticos	179
7.6. Constructores	181
7.6.1. <code>this()</code>	182
7.7. Paquetes	183
7.7.1. Crear un paquete desde NetBeans	183
7.8. Modificadores de acceso	184
7.8.1. Modificadores de acceso para clases	185
7.8.2. Modificadores de acceso para miembros	187
7.8.3. Métodos <code>get/set</code>	189
7.9. Enumerados	190
Ejercicios de clases	192
Ejercicios propuestos	219
8. Herencia	221
8.1. Superclase	221
8.2. Modificador de acceso para herencia	222
8.3. Redefinición de miembros heredados	224
8.3.1. <code>super</code> y <code>super()</code>	225
8.3.2. Selección dinámica de métodos	227
8.4. La clase <code>Object</code>	228
8.4.1. Método <code>toString()</code>	229
8.4.2. Método <code>equals()</code>	230
8.5. Clases abstractas	231
Ejercicios de herencia	233
Ejercicios propuestos	245
9. Interfaces	247
9.1. Interfaces	247
9.2. Clases anónimas	252
9.3. Interfaz <code>Comparable</code>	253
9.4. Interfaz <code>Comparator</code>	256

Ejercicios de interfaces	258
Ejercicios propuestos	265
10.Ficheros de texto	267
10.1. Excepciones	267
10.2. Flujos de entrada	273
10.3. Flujos de salida	275
Ejercicios de ficheros de texto	276
Ejercicios propuestos	291
11.Ficheros binarios	293
11.1. Flujos de salida	293
11.2. Flujos de entrada	295
11.3. Cierre de flujos	297
Ejercicios de ficheros binarios	298
Ejercicios propuestos	311
12.Collections	313
12.1. Listas	314
12.1.1. Métodos básicos de la interfaz <code>Collection</code>	315
12.1.2. Métodos globales de la interfaz <code>Collection</code>	320
12.1.3. Métodos de tabla de la interfaz <code>Collection</code>	321
12.1.4. Métodos específicos de la interfaz <code>List</code>	322
12.2. Interfaz <code>Set</code>	325
12.2.1. Conversiones entre colecciones	328
12.2.2. Clase <code>Collections</code>	330
12.2.3. Otras utilidades	333
12.3. Interfaz <code>Map</code>	335
12.3.1. Vistas <code>Collection</code> de los mapas	337
12.3.2. Implementaciones de <code>Map</code>	339
Ejercicios de <code>Collections</code>	340
Ejercicios propuestos	352
13.Stream	355
13.1. Interfaces funcionales y expresiones lambda	355
13.2. Algunas interfaces funcionales de la API	358
13.2.1. Referencias a métodos	360
13.3. Interfaz <code>Stream</code>	361
13.3.1. Formas de crear un <code>Stream</code>	362
13.3.2. Tuberías o <i>pipelines</i>	364
Ejercicios de <code>Stream</code>	368
Ejercicios propuestos	376

A. Utilidades	377
A.1. Números aleatorios	377
A.2. Intervalos de tiempo	378
A.3. Fechas y horas	379
A.3.1. <code>LocalDate</code>	379
A.3.2. <code>LocalTime</code>	381
B. Clases envoltorio (<i>wrappers</i>)	383

Capítulo 1

Conceptos básicos

En nuestra vida cotidiana estamos en contacto con multitud de máquinas que, en la mayoría de los casos, simplifican nuestras tareas. La forma que tenemos de comunicarnos con ellas, a través de botones, ruletas o teclas, es lo que se conoce como *interfaz hombre-máquina*. Supongamos que queremos cocinar en nuestro horno. Lo ideal sería que el horno no tuviera ningún botón ni mando, solo un pequeño micrófono al que pudiéramos dirigirnos y expresarle nuestros deseos: «*esta noche me apetece cenar una pizza de espinacas con extra de queso*». Por desgracia, este tipo de interfaz todavía no se encuentra en nuestros electrodomésticos. Quizás en unos años.

Volviendo al presente, un horno sencillo donde solo podamos controlar la temperatura adecuada en cada momento y el tiempo que estaremos cocinando, presenta los siguientes mandos:

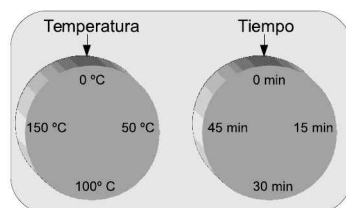


Figura 1.1. Interfaz hombre-máquina para un horno simple

Girando ambas ruedas y colocándolas en la posición deseada podemos configurar o programar el horno. Aunque internamente un horno funciona dejando pasar corriente eléctrica a través de unas resistencias que generan calor, este proceso se interrumpe y continúa, a intervalos, para controlar la temperatura deseada mediante un termostato. Además, todo está dirigido por un cronómetro que se encarga de controlar el tiempo total de funcionamiento. La interfaz, nuestras dos ruedas selectoras de tiempo y temperatura, nos abstrae del verdadero funcionamiento interno del horno. No necesitamos tener ningún conocimiento sobre resistencias, termostatos ni cronómetros. Los mandos hacen invisible el funcionamiento interno y transforman el manejo en algo mucho más sencillo.

Veamos este mismo proceso para un ordenador, una máquina mucho más compleja que un horno. El funcionamiento interno de un ordenador depende de voltajes que cam-

bian entre niveles bajos y altos: estos valores se representan también como ceros (0) y unos (1), un sistema de numeración binario, pues estos voltajes —o ceros y unos— que pasan a través de unos componentes electrónicos, son los que controlan el funcionamiento de la computadora. Cuando un videojuego o un procesador de textos se ejecuta en un ordenador, es difícil pensar que, en el fondo, no son más que el procesamiento de ceros y unos por dispositivos electrónicos. Para un humano, programar a través de ceros y unos es algo bastante complicado. Por ejemplo, indicarle a un ordenador que realice una suma se hace a través del código `0010101011011000` y una multiplicación es `0011000010101101`. Aquí se aprecia que, aparte de memorizar multitud de códigos binarios, un pequeño error puede ser un desastre y producir un resultado totalmente distinto del que pretendemos.

Para solucionar este problema, al igual que en nuestro horno, existe una interfaz hombre-máquina que nos facilita esta tarea. A este interfaz se le llama *lenguaje de programación*.

1.1. Algoritmo

Continuando con nuestra cena, para cocinarla nos basta con seguir una serie de instrucciones y seleccionar la posición adecuada de cada mando en cada momento. Dicho de otra forma, tenemos que seguir un conjunto de pasos definidos para resolver el problema: ¿cómo cocinar una pizza?

Podemos definir un *algoritmo* como un conjunto finito de instrucciones bien definidas que nos ayudan a resolver un problema. El algoritmo para preparar una pizza, que en el mundo gastronómico se conoce con el nombre de receta, es:

1. Introducir la pizza en el horno.
2. Colocar la temperatura a 150 °C.
3. Colocar el tiempo a 15 min.
4. Esperar.
5. Retirar y comer.

Estamos acostumbrados a utilizar multitud de algoritmos que son los procedimientos que realizamos de forma mecánica para solucionar un problema. Algunos ejemplos son: recetas de cocina, procesos para realizar operaciones matemáticas (sumas, multiplicaciones, etc.), pulsar los botones adecuados y en el orden correcto para que cualquier máquina haga su trabajo, etcétera.

Veamos el algoritmo para sumar dos números, utilizando a modo de ejemplo los números 2616 y 3708:

1. Colocar ambos números en dos filas haciendo coincidir las cifras del mismo orden (unidades con unidades, decenas con decenas, y así, sucesivamente) dos a dos.

$$\begin{array}{r} 2 \ 6 \ 1 \ 6 \\ + \ 3 \ 7 \ 0 \ 8 \\ \hline \end{array}$$

2. Comenzar por la derecha.
3. Hacer la suma de un solo guarismo de cada operando, anotando debajo las unidades resultantes y en la parte superior del guarismo de la izquierda las decenas, si existieran.

$$\begin{array}{r} \\ \\ + \\ \hline \end{array}$$

4. Repetir el punto 3 con el guarismo de la izquierda.

$$\begin{array}{r} \\ \\ + \\ \hline \end{array}$$

5. Terminar cuando no queden más elementos por sumar.

[illegible]

1.2. Lenguajes de programación

Un *lenguaje de programación* puede definirse como un idioma artificial diseñado para que sea fácilmente entendible por un humano e interpretable por una máquina. Consta de una serie de reglas y de un conjunto de órdenes o instrucciones. Cada una de estas instrucciones realiza una tarea determinada. A través de una secuencia de instrucciones podemos indicar a una computadora el algoritmo que debe seguir para solucionar un problema dado. A un algoritmo escrito utilizando las instrucciones de un lenguaje de programación se le denomina *programa*.

Existen multitud de lenguajes de programación, cada uno con sus ventajas e inconvenientes. Disponemos de lenguajes especializados para realizar cálculos científicos, para escribir videojuegos o para programar robots. **Fortran** es un lenguaje de programación diseñado para realizar aplicaciones científicas. Podemos utilizarlo para calcular operaciones complejas fácilmente, pero sería tremendamente laborioso utilizarlo para escribir un videojuego. Igualmente existen lenguajes de propósito general: aquellos que no están especializados en un campo concreto, pero con los que podemos realizar casi cualquier tarea con un mayor o menor esfuerzo.

Pero se nos plantea otra pequeña dificultad: un lenguaje de programación está diseñado para que una persona escriba fácilmente algoritmos, pero la circuitería de un ordenador no comprende ningún lenguaje distinto al sistema binario. La solución es utilizar una herramienta llamada *compilador*, que transforma el conjunto de órdenes o instrucciones que

escribimos utilizando cualquier lenguaje de programación en los ceros y unos que son comprensibles por la circuitería de la máquina.

Con esta solución tan elegante podremos programar una máquina tan compleja como un ordenador, casi de la misma forma en la que utilizamos un horno, abstrayéndonos de su funcionamiento interno, sin conocimientos de electrónica y sin necesidad de entender todas y cada una de sus partes.

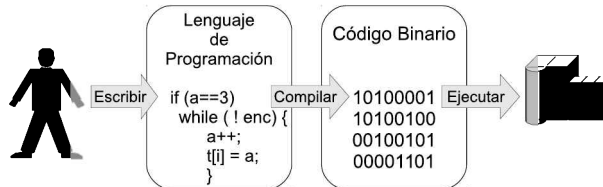


Figura 1.2. Interfaz hombre-ordenador a través de un lenguaje de programación

Entre todos los lenguajes hemos elegido Java por ser de propósito general, sencillo y didáctico, sin dejar de ser potente y escalable. Quizás, junto al lenguaje C, sea el lenguaje de programación más utilizado por empresas e instituciones científicas y académicas.

1.3. ¿Cuál es el propósito de este libro?

El objetivo principal de este libro es doble. En primer lugar, que el lector conozca y aprenda el funcionamiento de las instrucciones o sentencias que proporciona Java y, por otro lado, que sea capaz de utilizarlas para escribir algoritmos correctos que resuelvan problemas reales. Estos pueden ser tan simples como calcular la suma de dos números o tan complejos como gestionar la parte financiera de una empresa o desarrollar un videojuego.

Aquí hay que hacer hincapié en que el conocimiento de Java no implica saber programar correctamente. Dicho de otro modo, conocer el funcionamiento individual de cada instrucción no garantiza el éxito; este se consigue teniendo una visión global del problema, conociendo y aplicando técnicas algorítmicas y escribiendo las instrucciones en el orden correcto.

1.4. NetBeans IDE

Un programador dispone de multitud de herramientas para llevar a cabo su tarea. Lo más básico es un editor de texto donde escribir las instrucciones y un compilador que transforme el fichero de texto, con las sentencias de Java, en un fichero escrito en un lenguaje especial, capaz de ser interpretado por la *Máquina Virtual de Java* (JVM).

También hay entornos de programación más sofisticados que proporcionan una enorme cantidad de funcionalidades: editor de texto, ayuda, compilador, depurador y, en general, casi cualquier cosa que se nos pueda ocurrir. Estos entornos se conocen como *IDE*, las siglas en inglés de Entorno Integrado de Desarrollo, y son un conjunto de herramientas integradas orientadas al desarrollo de software.

De todos los entornos disponibles, hemos decidido utilizar NetBeans (Figura 1.3), que es gratuito y de código abierto. En la página web oficial de NetBeans (www.netbeans.org) podemos descargar la última versión. Existen otros entornos de desarrollo, cada uno con sus características. Invitamos al lector a probar y experimentar, hasta que encuentre el que mejor se adapte a sus necesidades, aunque no difieren mucho unos de otros.

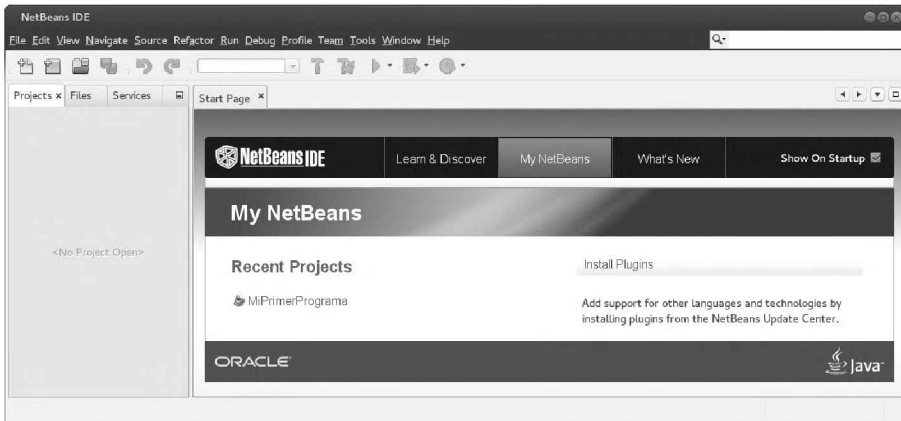


Figura 1.3. Ventana principal de NetBeans

En NetBeans, crear un proyecto significa crear un nuevo programa en el que escribiremos las sentencias en Java que necesitamos. En el menú **File/New Project...** se accede a la ventana representada en la Figura 1.4, donde podemos elegir el tipo de proyecto. NetBeans se puede utilizar para escribir programas en distintos lenguajes de programación.

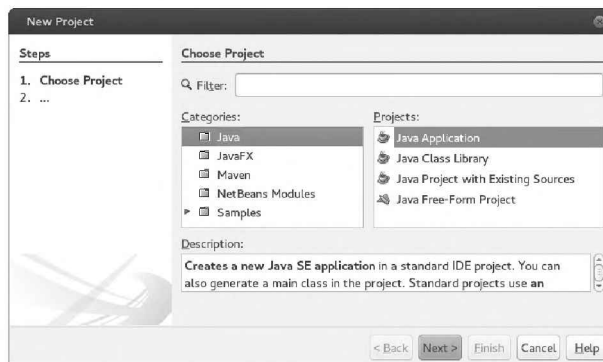


Figura 1.4. Selección del tipo de proyecto

1. Seleccionaremos el lenguaje que nos interese, en nuestro caso, Java.
2. Elegiremos **Java Application**, donde el programa que diseñemos hará que el ordenador se comporte como una consola de entrada/salida, evitando elementos de programación avanzada como los gráficos de escritorio.

Pulsando en el botón **Next** > llegamos a una ventana (Figura 1.5) donde podremos elegir el nombre y la ubicación de nuestro proyecto. Hemos elegido como nombre *MiPrimerPrograma*. Marcaremos la casilla: **Create Main Class**.

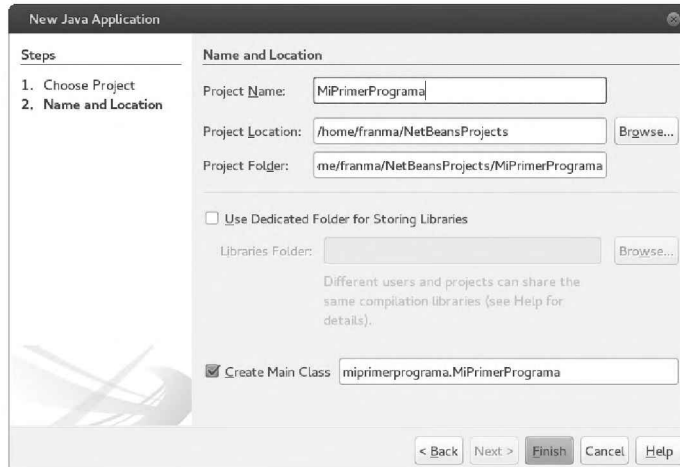


Figura 1.5. Selección del tipo de proyecto

El botón **Finish** finaliza el proceso de crear un nuevo proyecto. Ahora se muestra un editor (Figura 1.6), donde podremos insertar las instrucciones que deseemos. NetBeans simplifica el trabajo creando el código del programa principal. Nosotros insertaremos las sentencias de nuestro programa entre las llaves de la función `main`, en la parte que aparece sombreada.

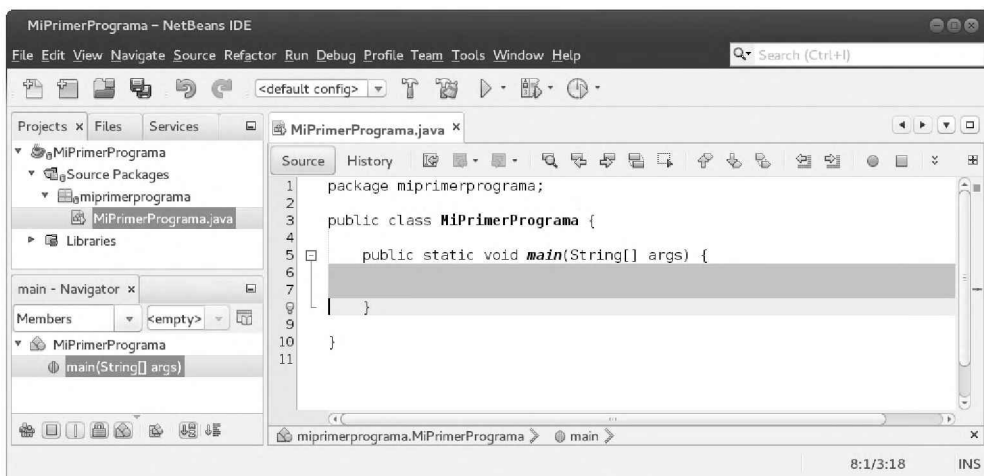


Figura 1.6. Editor de NetBeans

Una vez que hemos escrito las sentencias que necesitamos, para que comiencen a ejecutarse pulsaremos en el botón **Run Project**, representado por un triángulo verde que simula una tecla *Play* (véase Figura 1.7).

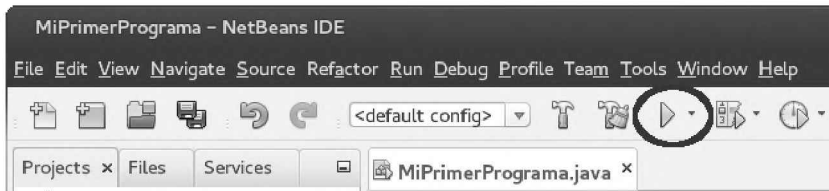


Figura 1.7. Botón ejecutar

1.5. El programa principal

Cuando se aprende a programar, durante la primera etapa, es posible que la frase: *esto requiere unos conocimientos que están fuera del alcance de un principiante* aparezca demasiadas veces. Pero aquí está plenamente justificada; más adelante veremos el concepto de clase y de función pero por ahora, para escribir los primeros programas, utilizaremos:

```
package miprimerprograma;

public class MiPrimerPrograma {

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

Al escribir un programa en Java usaremos literalmente la formula anterior, aunque todavía no la comprendamos. De hecho, ni siquiera tendremos que escribir nada, ya que NetBeans la escribirá por nosotros. Solo tenemos que sustituir *algoritmo* por el conjunto de instrucciones que necesitemos.

Hay que destacar que la primera línea de código,

```
package miprimerprograma;
```

especifica que nuestro programa se agrupará en el paquete `miprimerprograma`. El nombre del paquete dependerá del nombre del proyecto; dicho de otra forma, NetBeans escribirá un nombre distinto de paquete dependiendo del nombre que cada lector asigne a sus proyectos. Por este motivo, en la solución de los ejercicios hemos omitido la línea con la sentencia `package`.

1.6. Palabras reservadas

Como se puede apreciar en el apartado anterior, en Java existe una serie de palabras con un significado especial, como `package`, `class` o `public`. Estas se denominan palabras

reservadas y definen la gramática del lenguaje. En la Tabla 1.1, se muestra el conjunto de las palabras reservadas en Java.

Tabla 1.1. Palabras reservadas de Java

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

Al conjunto anterior hay que sumar dos palabras reservadas muy curiosas: `const` y `goto`, que no pueden ser utilizadas en el lenguaje, pero aun así, están reservadas por si en un futuro fueran necesarias. Además, existen tres valores literales: `true`, `false` y `null`, que tienen también un significado especial para el lenguaje, teniendo un estatus parecido a una palabra reservada.

Las palabras reservadas solo pueden utilizarse en determinado lugar de un programa y no pueden ser utilizadas como identificadores.

1.7. Variables

La Real Academia de la Lengua Española define variable como la *magnitud que puede tener un valor cualquiera de los comprendidos en un conjunto*. Dicho con otras palabras: una variable es una representación, mediante un identificador, de un valor. Este no tiene que ser inalterable y puede cambiar durante la ejecución de un programa. A las variables se les asignan valores concretos por medio del operador `=`, llamado *operador de asignación*. Ejemplo de ello es:

```
a = 3
```

Aquí el nombre o identificador de la variable es `a`, y el valor asignado es 3. Esto no significa que posteriormente no pueda cambiar su valor por otro. Otro ejemplo:

```
a = 10
b = a + 1
```

Utilizamos dos variables `a` y `b`. En la primera asignación damos un valor de 10 a la variable `a`, y en la segunda asignación damos a `b` el valor que tuviera `a` más 1. Como `a` vale 10, `b` tomará un valor de 10 más 1, es decir, 11.

1.7.1. Identificadores

El nombre con el que se identifica cada una de las variables debe ser único, teniendo en cuenta que Java distingue entre mayúsculas y minúsculas, es decir, el identificador `edad` es

distinto a **eDaD**. Además, no podemos utilizar como identificador ninguna palabra reservada del lenguaje. Los identificadores deben seguir las reglas:

- Comienzan siempre por una letra, un subrayado (`_`) o un dólar (`$`).
- Los siguientes caracteres pueden ser letras, dígitos, subrayado (`_`) o dólar (`$`).
- Se hace distinción entre mayúsculas y minúsculas.
- No hay una longitud máxima para el identificador.

Existe una regla de estilo que recomienda distinguir las palabras que forman un identificador escribiendo en mayúscula la primera letra de cada palabra. Esta notación hace que el aspecto del identificador se asemeja a las jorobas de un camello, de ahí su nombre: notación *Camel*. Algunos ejemplos de ella son: `edad`, `maxValor`, `numCasasLocalidad` o `notaMediaTercerTrimestre`.

1.8. Tipos

En un programa en ejecución, las variables se almacenan en la memoria del ordenador. Cada una de ellas necesita un tamaño para guardar sus valores. Un tamaño demasiado pequeño no permite guardar valores grandes o muy precisos, y se corre el riesgo de que el valor a guardar no quepa en el espacio reservado. Por el contrario, utilizar un tamaño excesivamente grande desaprovecha la memoria, haciendo un uso ineficiente de ella.

Veamos un ejemplo: la variable `nota`, que utilizaremos para guardar las calificaciones de los alumnos, almacenará valores que están comprendidos en el rango de 0 a 10. Con un tamaño en memoria para dos dígitos¹ es suficiente.

```
nota = 10
```

La forma de guardar esta información en la memoria es:

```
nota 

|   |   |
|---|---|
| 1 | 0 |
|---|---|


```

Por el contrario, si deseamos utilizar calificaciones comprendidas entre 0 y 300, los dos dígitos son insuficientes.

```
nota = 125
```

Esto supondría que en la memoria no habría espacio reservado suficiente:

```
nota 

|   |   |
|---|---|
| 1 | 2 |
|---|---|

 5
```

Lo ideal sería que cada variable reserve un espacio lo suficientemente grande para que pueda almacenar todos los valores que guardará en algún momento, pero esto no siempre es posible.

¹El tamaño de la memoria en un ordenador se mide en bytes —ocho bits (cero o uno)— y no en dígitos. Pero para comprender el concepto de tipo supondremos, por ahora, que el tamaño de la memoria se mide en dígitos.

La solución a este problema no es definir un tamaño de memoria para cada variable, sino definir unos tipos de variables, con unos tamaños y rangos de valores conocidos, y que las variables utilizadas en nuestros programa se ciñan a estos tipos.

En Java encontramos los tipos predefinidos: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` y `char`, también conocidos como *tipos primitivos*. Estos tienen un tamaño predefinido y pueden almacenar valores dentro de unos rangos (a mayor tamaño de memoria, mayor es el rango de posibles valores). Si con estos tipos primitivos no cubrimos nuestras necesidades, en capítulos posteriores veremos cómo crear otros.

Al escribir un programa, hemos de indicar a qué tipo pertenece cada variable. Este proceso recibe el nombre de *declaración de variables* y se hará forzosamente antes de su primer uso. Veamos como ejemplo la forma de declarar la variable `importe` de tipo `int`:

```
int importe;
```

Todas las declaraciones de variables terminan en punto y coma (;), aunque es posible declarar a la vez varias del mismo tipo, separándolas por comas (,):

```
int importe, total, suma;
```

Existe la posibilidad de asignar un valor —inicializar— una variable en el momento de declararla,

```
int importe = 100;
```

que declara la variable `importe` de tipo `int` y le asigna un valor de 100.

1.8.1. Rangos

La Tabla 1.2 describe el tamaño —espacio que ocupa en memoria— y el rango de valores que puede almacenar cada tipo.

Tabla 1.2. Tipos primitivos en Java

Tipo	Uso	Tamaño	Rango
byte	entero corto	8 bits	de -128 a 127
short	entero	16 bits	de -32 768 a 32 767
int	entero	32 bits	de -2 147 483 648 a 2 147 483 647
long	entero largo	64 bits	$\pm 9\,223\,372\,036\,854\,775\,808$
float	real precisión sencilla	32 bits	de -10^{32} a 10^{32}
double	real precisión doble	64 bits	de -10^{300} a 10^{300}
boolean	lógico	1 bit	<i>true</i> o <i>false</i>
char	texto	16 bits	cualquier carácter

El desbordamiento de memoria puede ocurrir cuando un dato ocupa más espacio del asignado. El espacio extra se tomaría de la memoria adyacente, ocupándola. Y es aquí donde aparece el verdadero problema: desconocemos qué es lo que había almacenado en la porción extra de memoria que hemos sobreescrito. Quizás tengamos la suerte de que esté vacío, o por el contrario, podríamos estar destruyendo algún dato crucial. En Java no

existe el desbordamiento de memoria, al disponer el lenguaje de un fuerte control de tipos que impide que se puedan realizar operaciones con desbordamiento. Sin embargo, sí existen lenguajes donde el control de tipos es menos exhaustivo o incluso inexistente, y donde sí podemos encontrarnos con una situación de desbordamiento de memoria.

Por una parte, Java impide que asignemos a una variable un valor fuera del rango permitido por el tipo al que pertenece,

```
byte a = 300;
```

que produce un error del compilador, ya que el tipo `byte` posee un rango comprendido entre `-128` y `127`.

Por otra parte, para evitar el desbordamiento como resultado de un cálculo, los rangos en Java funcionan de forma circular; cuando se sobrepasa el valor máximo, se vuelve al valor mínimo del rango, y viceversa. Para el caso de una variable de tipo `byte`, la forma de contar sería: `0, 1, 2, ..., 126, 127, -128, -127, ..., -2, -1, 0, 1, ...`. Expresado de otra forma, el valor máximo de un tipo más 1 no es un valor fuera de rango, sino el valor mínimo permitido para ese tipo. Por ejemplo, después de las sentencias,

```
byte a = 127;
a = a + 1;
```

`a` tendrá el valor `-128`, que es el siguiente a `127`.

En el Apartado 1.13.1 veremos a fondo el funcionamiento de `System.out.println()`, pero nos adelantamos para que el lector pueda visualizar el valor de las variables usadas en los ejemplos. La forma de mostrar en pantalla el valor de la variable `a` es escribiendo,

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

1.9. Constantes

Las *constantes* son un caso especial de variables, en el que una vez que se les asigna su primer valor, este permanece inmutable durante el resto del programa. Cualquier dato que no cambie es candidato a guardarse en una constante. Ejemplo de constante son el número π , e o el IVA aplicable.

La declaración de constantes es similar a la de variables, pero añadiendo la palabra reservada `final`:

```
final tipo2 nombreConstante3;
```

La mayoría de los programadores suele escribir sus códigos siguiendo una guía de estilos. Es habitual que los identificadores de las constantes se escriban en mayúscula. Nada nos impide escribirlas de otra forma, pero se hace así para distinguirlas, de un solo vistazo, de las variables. Un ejemplo de la declaración de constantes,

```
final byte MAYORIA_EDAD = 18;
final double PI = 3.141592;
```

²En lugar de *tipo* se escribirá cualquiera de los tipos primitivos de Java: `int`, `char`, `byte`, etc.

³*nombreConstante* puede sustituirse por cualquier nombre que nos guste para una constante. Por ejemplo: `PI`, `MAXIMO`, `IVA`, etc.

1.10. Comentarios

Un programa no solo está formado por instrucciones del lenguaje; también es posible incluir notas o comentarios. El objetivo de estos es doble: describir la funcionalidad del código (qué hace), y facilitar la comprensión de la solución implementada (cómo lo hace).

Se considera una buena práctica escribir códigos bien documentados. Están especialmente indicados para facilitar el mantenimiento (modificación futura) de los programas: un código con el que trabajemos habitualmente y que conocemos con gran exactitud, puede convertirse en un galimatías tras un tiempo sin trabajar con él; por otra parte, cuando se trabaja en colaboración con otros programadores, los comentarios que acompañan al código ayudan al resto del equipo.

Java dispone de tres tipos de comentarios:

Comentario multilínea: cualquier texto incluido entre los caracteres `/*` (apertura de comentario) y `*/` (cierre de comentario) será interpretado como un comentario, y puede extenderse a través de varias líneas.

Comentario hasta final de línea: todo lo que sigue a los caracteres `//` hasta el final de la línea se considera un comentario.

Comentario de documentación: similar al comentario multilínea, con la diferencia que, para iniciarlo, se utilizan los caracteres `/**`. Existen herramientas que generan documentación automática a partir de este tipo de comentarios.

Veamos algunos ejemplos:

```
/* esto es un comentario  
   que se extiende  
   durante varias líneas */  
  
int a; //declaramos la variable "a" como un entero  
  
/** este comentario aparecerá en caso de utilizar una herramienta  
    de generación automática de documentación */
```

1.11. Operaciones básicas

Java dispone de multitud de operadores con los que se pueden crear expresiones utilizando como operandos variables, constantes, números y otras expresiones.

1.11.1. Operador de asignación

Modifica el valor de una variable. La sintaxis es:

```
variable = expresión;
```

Tabla 1.3. Operador de asignación

Símbolo	Descripción
=	Asignación

A la variable se le asigna como valor el resultado de la expresión. Una expresión no es más que una serie de operaciones. Si en el momento de la asignación la variable tuviera un valor anterior, este se pierde. Un ejemplo:

```
int total, a; //declaramos dos variables enteras

total = 123; //la variable total toma un valor de 123.
total = 0; //ahora toma un valor de 0. El valor 123 se pierde.
a = 3; //la variable a toma el valor 3
```

En estas asignaciones la **expresión** asignada es un valor explícito y no una expresión que necesite ser evaluada. Estos valores explícitos se llaman literales. Por ejemplo, 123 es un literal entero, mientras que 2.5 es un literal `double`. En cambio,

```
total = 2 * a; //total toma como valor el resultado de:
               //2 por el valor de la variable a (que es 3). Es decir 6.
total = total - 5; //a total se le asigna el valor de:
                  //total (que es 6) menos 5. Es decir 1.
```

Ahora a `total` se le asigna dos expresiones que necesitan ser evaluadas antes de la asignación. La primera es una multiplicación —operador `*`— y la segunda es una resta.

Desde que se declara una variable hasta que se le asigna el primer valor, ¿cuánto vale la variable? Java soluciona este problema dejando provisionalmente las variables sin valor alguno e impidiendo realizar operaciones con ellas hasta que realicemos la primera asignación. En otros lenguajes de programación hay que tener mucho cuidado, ya que la política de variables sin asignar cambia. El lenguaje `C` asigna a la variable un valor al azar.

1.11.2. Operadores aritméticos

El operador `-` (menos unario) sirve para cambiar el signo de la expresión que le sigue (Tabla 1.4).

```
a = 1;
b = -a; // b vale -1
```

El operador `%` devuelve el resto de dividir el primer operando entre el segundo. Por ejemplo `7%3` (se lee resto de 7 módulo 3) vale 1, ya que al dividir 7 entre 3 el resto es 1.

Los operadores `++` y `--`, se utilizan para incrementar o decrementar una variable en 1. El siguiente código,

```
a ++;
b --;
```

es equivalente a

```
a = a + 1;
b = b - 1;
```

Tabla 1.4. Operadores aritméticos

Símbolo	Descripción
+	Suma
+	Más unario: positivo
-	Resta
-	Menos unario: negativo
*	Multiplicación
/	División
%	Resto módulo
++	Incremento: +1
--	Decremento: -1

En un programa el incremento o decremento de una variable es algo tan usual, que estos operadores están pensados con el único propósito de ahorrar trabajo al programador a la hora de teclear, y de paso, hacer el código más compacto, legible y eficiente. Ambos operadores pueden utilizarse de forma prefija (++a;) o postfija (a++;), y su comportamiento es distinto. Cuando se utiliza como prefijo, el operador tiene precedencia sobre el resto. Y usado como posfijo, se realiza antes cualquier otra operación, dejando el incremento para el final.

```
int a, b, c; //declaramos la variables de tipo entero

a = 1; //a la variable a le asignamos 1
b = a ++; //primero asignamos el valor de a a b, y después incrementamos a
c = ++ a; //primero incrementamos a, y después asignamos su valor a c
```

Después de estas líneas de código, a vale 3, b vale 1 y c vale 3.

Al asignar b, lo primero que se hace es copiar el valor actual de a y a continuación, se incrementa a. El orden de las operaciones es asignar (=) e incrementar (++). El valor final de c es 3, debido a que la primera operación que se hace es incrementar a, y después asignamos el valor incrementado a la variable c.

1.11.3. Operadores relacionales

Son aquellos que producen un resultado lógico o booleano a partir de las relaciones de expresiones numéricas (Tabla 1.5). El resultando solo permite dos posibles valores: verdadero o falso. En Java estos valores se representan mediante los literales **true** y **false**. Al principio, es usual confundir el operador de asignación (=) con el operador de comparación (==), y creemos estar comparando, cuando en realidad estamos asignando, o viceversa. Comparemos de distintas formas los números 3 y 5:

3 < 5 ¿es 3 menor que 5? Es cierto; 3 es un número más pequeño que 5. Por tanto, la expresión devuelve **true**.

3 == 5 ¿es 3 igual que 5? Falso; ambos números son distintos, es decir, no son iguales. La expresión devuelve **false**.

3 <= 5 ¿es 3 menor o igual que 5? Cierto. La expresión devuelve **true**.

3 <= 3 ¿es 3 menor o igual que 3? Es cierto.

3 != 4 ¿es 3 distinto de 4? Cierto; ya que 3 es distinto a 4.

Tabla 1.5. Operadores relacionales

Símbolo	Descripción
==	Igual que
!=	Distinto que
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

1.11.4. Operadores lógicos

Permiten operar a partir de expresiones lógicas, formando expresiones más complejas, que devuelven, a su vez, un valor lógico (Tabla 1.6). Existen los operadores *and* (conjunción *Y*), *or* (disyunción *O*) y *not* (negación).

Tabla 1.6. Operadores lógicos

Símbolo	Descripción
&&	Operador and: Y
	Operador or: O
!	Operador not: Negación

La expresión formada a partir de otras dos unidas por el operador *and* será **true** cuando ambas expresiones utilizadas se evalúen como ciertas. Y en caso contrario, es decir, si alguna o las dos expresiones se evalúa como falsa, la expresión resultante también será falsa.

exprA && exprB

La expresión anterior será cierta solo cuando **exprA** y **exprB** sean ciertas. Veamos las siguientes expresiones:

3 <= 5 && 2 == 2 ¿es 3 menor o igual que 5 y a la vez, es 2 igual a 2? Cierto, ya que tanto la expresión **3 <= 5** como **2 == 2** son ciertas.

3 <= 5 && 2 > 10 ¿es 3 menor o igual que 5 y a la vez, es 2 mayor que 10? Falso, ya que al menos una expresión utilizada es falsa (**2 > 10**).

El operador *or* —o— será cierto cuando alguna de las expresiones que lo forman sean cierta. En caso contrario será falso.

exprA || exprB

La expresión se evaluará cierto cuando, o bien, `exprA` sea cierta o bien cuando `exprB` sea cierta, o ambas sean ciertas.

`1 != 2 || 5 < 3` ¿es cierto que 1 sea distinto de 2 o que 5 sea menor que 3? La primera expresión es cierta: 1 es distinto de 2, mientras que la segunda expresión es falsa. Por tanto, la expresión completa se evalúa como verdadera.

El operador not —negación—, es un operador unario que cambia los valores booleanos de cierto a falso y viceversa.

`!(1 < 2)` la expresión se evalúa como la negación —lo contrario— de 1 menor que 2, que es verdadera. Por tanto, la expresión completa se evalúa falsa.

1.11.5. Operadores «opera y asigna»

Por simplicidad existen otros operadores de asignación llamados *opera y asigna* (Tabla 1.7), que realizan la operación indicada tomando como operandos el valor de la variable a la izquierda y el valor a la derecha del `=`. El resultado se asigna a la misma variable utilizada como primer operando.

Tabla 1.7. Operadores: opera y asigna

Símbolo	Descripción
<code>+ =</code>	Suma y asigna
<code>- =</code>	Resta y asigna
<code>* =</code>	Multiplica y asigna
<code>/ =</code>	Divide y asigna
<code>% =</code>	Módulo y asigna

Todos tienen el mismo funcionamiento. Utilizan la misma variable para operar con su valor y asignarle el resultado. Veamos a modo de ejemplo:

```
var += 3;
```

Lo que es equivalente a:

```
var = var + 3;
```

De igual forma,

```
x *= 2;
```

es equivalente a

```
x = x * 2;
```

1.11.6. Operador ternario

Este operador devuelve un valor que se selecciona de entre dos posibles (Tabla 1.8). La selección dependerá de la evaluación de una expresión relacional o lógica, que como hemos visto puede tomar dos valores: verdadero o falso.

Tabla 1.8. Operador ternario

Símbolo	Descripción
<code>? :</code>	Operador ternario

El operador tiene la sintaxis,

`expr ? valor1 : valor2`

La evaluación de la expresión decidirá cual de los dos posibles valores se devuelve. En el caso de que la expresión resulte cierta, se utiliza `valor1`, y cuando la expresión resulte falsa, se utiliza `valor2`. Veamos un ejemplo:

```
int a, b;
a = 3 < 5 ? 1 : -1; // 3 < 5 es cierto: a toma el valor 1
b = a == 7 ? 10 : 20; // a (que vale 1) == 7 es falso: b toma el valor 20
```

1.11.7. Precedencia

La Tabla 1.9 muestra los operadores ordenados, según su precedencia, de mayor a menor. La precedencia establece qué operaciones se realizan antes. A igualdad de precedencia las operaciones se realizan en el mismo orden en el que se escriben: de izquierda a derecha. Sea,

`2 + 3 * 4` el operador `*` tiene una precedencia mayor que el operador `+`, lo que significa que la multiplicación se realizará antes que la suma. Primero se hace la operación `3 * 4` (que es 12) y a continuación se realiza la suma `2 + 12`, que es 14.

`3 <= 5 && 2 == 2` el operador con mayor precedencia es `<=`, que será la primera operación que se realice, siendo cierta. Queda,

`true && 2 == 2`

A continuación se realizará la comparación, que también resulta cierta,

`true && true`

Y el operador con menor precedencia es `and` lógico, que se realiza en último lugar, resultando la expresión cierta.

La precedencia puede romperse utilizando paréntesis; por ejemplo, en la expresión:

`(2 + 3) * 4`

el uso de paréntesis obliga a que la primera operación que se realice sea la suma.

Tabla 1.9. Precedencia de operadores

Descripción	Operador
Postfijos	expr++ expr--
Unarios prefijos	++expr --expr +expr -expr !expr
Aritméticos	* / %
Aritméticos	+ -
Relacionales	< <= > >=
Comparación	== !=
AND lógico	&&
OR lógico	
Ternario	? :
Asignación	= += -= *= /= %= & = ^=

1.12. Conversión de tipos

Como hemos visto, todas las variables en Java tienen asociado un tipo. Cuando asignamos un valor a una variable, ambos deben ser del mismo tipo. A una variable de tipo `int` se le puede asignar un valor `int` y a una variable `double` se le puede asignar un valor `double`.

```
int a = 2;
double x = 2.3;
```

Cada tipo se caracteriza por ocupar un tamaño en memoria, donde se almacenan los valores correspondientes.

Si escribimos,

```
int a = 2.6; //trata de asignar un valor real a una variable entera
```

El compilador nos avisará de que estamos cometiendo un error y no nos dejará compilar. La razón de este control de tipos tan estricto es evitar errores durante la ejecución del programa, ya que es evidente que una variable de un tipo no puede almacenar valores con un tamaño superior. Por ejemplo, un valor `double` ocupa en la memoria 64 bits, mientras que una variable `int` utiliza 32 bits para almacenar un valor. Simplemente un valor `double` no cabe en una variable `int`.

Sin embargo, un valor `int` puede ser guardado sin problemas en una variable `double`. ¿Por qué no permitir una asignación como esta?

```
int a = 3;
double x = a;
```

Java permite esta asignación sin violar la norma de que a una variable `double` se le asigne un valor `double`. Para ello, Java convierte de forma automática el valor entero 3 en el valor `double` 3.0 antes de asignarlo a la variable `x`. Esto es posible porque la variable `double` es de mayor tamaño que el valor `int`, es decir, tiene suficiente espacio para guardarlo. Por tanto, este tipo de conversiones y asignaciones automáticas será posible cuando la variable sea de mayor tamaño que el valor asignado. Se habla entonces de conversiones de ensanchamiento.

Nos permitirá, por ejemplo, guardar valores `byte`, `short`, `int`, `long` o `float` en una variable `double`⁴.

Muy distinto es que intentemos asignar un valor `double` a una variable `int`, que no tiene sitio suficiente para guardarlo. Lo normal es que se trate de un error del programador. En este caso Java no hará ninguna conversión automática. Se limitará a darnos un error de compilación. Sin embargo, a veces es interesante guardar la parte entera de un número con decimales en una variable entera. Evidentemente, esto supone una pérdida de información, ya que los decimales desaparecerán. Para ello deberemos colocar un *cast* o molde delante del valor que queremos asignar,

```
int a = (int) 2.6; //(int) indica el tipo al que se convertirá el valor
```

El *cast* es lo que va entre paréntesis. Lo que hace es eliminar (truncar) la parte decimal de 2.6 y convertirlo en el entero 2, que podrá ser asignado a la variable `a` sin problemas. Este tipo de conversión se llama de *estrechamiento*, ya que fuerza la asignación de un tipo de dato en una variable de menor tamaño, eso sí, con pérdida de información.

Nada impide, aunque no es necesario, colocar un *cast* en una conversión de *ensanchamiento*. Esto a veces hace que el programa gane en legibilidad:

```
double x = (double) 3;
```

1.13. API de Java

Una de las grandes ventajas de los lenguajes de programación modernos es que disponen de una amplia biblioteca de herramientas que realizan tareas complejas de forma transparente al programador que las utiliza, facilitando su tarea.

En el caso de que un lenguaje de programación no disponga de alguna herramienta específica es necesario que sea el propio programador quien la diseñe, con los inconvenientes que esto conlleva. Es indudable que el hecho de disponer de herramientas prediseñadas facilita la labor de programar: por un lado se ahorra tiempo y trabajo, al no tener que implementarla; y por otro aporta un extra de seguridad al tener la certeza de que estas herramientas han sido diseñadas y comprobadas por programadores expertos.

Ilustraremos esta idea con un ejemplo: supongamos que deseamos colgar un bonito cuadro en el salón de nuestra casa. Para ello es primordial hacer un taladro en la pared⁵. Si no disponemos de un taladro eléctrico tendremos que construirlo, lo que nos obliga a tener conocimientos de mecánica y electricidad entre otras cosas, a la vez que es una tarea que ocupa nuestro tiempo. Es más, una vez construido tendremos dudas sobre su calidad; si no lo hemos hecho bien es posible que nos de un calambrazo al usarlo. Es mucho más cómodo disponer de la máquina para taladrar y solo tener que utilizarla. Las herramientas que acompañan a un lenguaje de programación —al igual que el taladro eléctrico— están a nuestra disposición para facilitar las tareas cotidianas y liberarnos del trabajo de construirlas.

⁴Caso especial son los valores de tipo `char` que, con un tamaño de 16 bits, pueden ser convertidos a su valor entero en el código *UNICODE*, como se verá más adelante. Esto permite asignarlos a una variable `int`,

```
int c = 'a'; //que equivale a int c = 97;
```

ya que 97 es el código asignado al carácter 'a'.

⁵Existen varias maneras de colgar un cuadro, pero supondremos que la única forma de hacerlo es mediante un taladro.

A estas herramientas, en Java, se les denomina *clases* y facilitan multitud de tareas. Algunos ejemplos de las funcionalidades que nos brindan son:

- **Lectura de datos:** leen información desde el teclado, desde un fichero o desde otros dispositivos.
- **Cálculos complejos:** realizan operaciones matemáticas como raíces cuadradas, logaritmos, cálculos trigonométricos, etc.
- **Manejo de errores:** controlan la situación cuando se produce un error de algún tipo.
- **Escritura de datos:** escriben información relevante en dispositivos de almacenamiento, impresoras, monitores, etc.

Estos son solo algunos ejemplos, pero la cantidad de clases que se distribuyen con Java es enorme y cubren las necesidades típicas de un programador. A toda esta biblioteca de clases se le denomina *API*; que son las siglas en inglés de *Interfaz de Programación de Aplicaciones*. El número de clases es tal que para facilitar su organización se agrupan según su funcionalidad. A una agrupación de clases se le denomina *paquete*. Los paquetes pueden agruparse, a su vez, en otros paquetes. Por ejemplo, la clase `Math`, que proporciona herramientas para realizar cálculos matemáticos, se engloba dentro del paquete `lang`, que engloba clases que son fundamentales para el lenguaje, y que a su vez se encuentra dentro del paquete `java`, que es el que contiene al resto de los paquetes.

Cada clase se identifica mediante su nombre completo —o nombre cualificado— que incluye la estructura de paquetes junto al nombre de la clase. Por ejemplo, el nombre cualificado para la clase `Math` es: `java.lang.Math`.

A su vez, una clase proporciona una o varias funcionalidades, que se denominan *métodos*. Si consideramos el taladro eléctrico como una clase, nos proporciona dos funcionalidades —dos métodos—: taladrar y lijar. `Math`, entre otros, dispone de los métodos `sqrt()`, que calcula una raíz cuadrada, o de `abs()`, que calcula el valor absoluto de un número.

Podemos utilizar cualquier método de una clase de la API, para ello, tendremos que escribir el nombre del método junto al nombre cualificado de la clase a la que pertenece. Por ejemplo, veamos cómo obtener la hora actual del sistema. Para ello utilizaremos el método `now()` de la clase `LocalTime`. Esta clase se ubica en el paquete `java.time`,

```
System.out.println(java.time.LocalTime.now()); //nombre cualificado
```

Tener que escribir continuamente el nombre cualificado de una clase —por ejemplo `java.time.LocalTime`— puede llegar a ser engorroso. Una alternativa es declarar que vamos a utilizar una clase concreta, mediante la palabra reservada `import`. De la forma:

```
import java.time.LocalTime;
```

Que se interpreta como: voy a necesitar —importar— en mi programa la clase `LocalTime`, que se encuentra dentro del paquete `time` que a su vez se encuentra dentro del paquete `java`. La sentencia `import` se coloca justo debajo de la declaración del paquete, como en la Figura 1.10. A partir de ahora solo es necesario escribir el nombre de la clase junto al método que deseamos utilizar,

```
System.out.println(LocalTime.now()); //nombre corto
```

Es posible importar tantas clases como necesitemos y el hecho de importar una clase no nos obliga a utilizarla; tan solo acorta su escritura en la expresión donde la utilizemos. En ocasiones tener que importar una a una las clases de un paquete puede ser algo tedioso. Es posible importar todas las clases de un paquete mediante un asterisco:

```
import java.time.*; //importa todas las clases del paquete java.time
```

Con respecto al mecanismo de importación, el paquete `java.lang` es una excepción. Al albergar clases fundamentales para Java —sin las cuales sería prácticamente imposible programar— se necesitan sus clases en prácticamente cualquier programa. Por este motivo, es el propio compilador el que importa de forma automática todas las clases de `java.lang`,

```
import java.lang.*; //se importa de forma automática
```

Es decir, podemos utilizar cualquier clase del paquete `java.lang` mediante su nombre corto sin tener que preocuparnos de su importación.

Por otra parte, cada clase que compone la *API* puede utilizarse de dos formas:

- De forma estática: se utiliza directamente el método. Por ejemplo, la clase `Math` se utiliza de forma estática, veamos como calcular la raíz cuadrada de 16:

```
raiz = Math.sqrt(16); //que resulta 4.0
```

- De forma no estática: esta manera de utilizar las clases, requiere del operador `new` que se verá en profundidad en capítulos posteriores. Un ejemplo de clase que se utiliza de esta forma es `Scanner`, que permite que el usuario introduzca datos en una aplicación. Por ahora, utilizaremos la clase `Scanner` como una fórmula literal (véase el apartado 1.13.2).

Hasta aquí hemos visto las clases como un conjunto de herramientas, pero en el Capítulo 7 las estudiaremos a fondo, veremos cómo implementar nuestras propias clases y todo lo relacionado con ellas. Por ahora, nos limitaremos a utilizar algunas de ellas sin más.

1.13.1. Salida por consola

Una de las operaciones más básica que proporciona la API es aquella que permite mostrar mensajes en el monitor, con idea de aportar información al usuario que utiliza el programa. Cuando los mensajes se muestran de forma simple: en modo texto y sin interfaz gráfica, se habla de *salida por consola*. Java dispone para ello de la clase `System` con los métodos:

`System.out.print("Mensaje");` que muestra literalmente el mensaje en el monitor.

`System.out.println("Mensaje");` igual que el anterior pero, tras el mensaje, inserta un retorno de carro (nueva línea).

```
System.out.print("Hola mundo.");
```



Figura 1.8. Salida por consola

Un ejemplo de como `System.out.print()` muestra la salida por consola se puede apreciar en la Figura 1.8.

El uso de la clase `System` es tan básico que no es necesario importarla, esto lo hace Java por defecto. Incluso el entorno que vamos a utilizar para escribir programas —*NetBeans*— tiene un truco para `System.out.println`: no es necesario escribir el método completamente; basta con escribir `sout` y pulsar la tecla tabulador. *NetBeans* lo escribirá por nosotros.

Para combinar la salida de mensajes literales de texto y el valor de las variables utilizaremos `+`, que nos permite unir todos los elementos que deseemos para formar el mensaje de salida.

```
int edad = 8;  
System.out.print("Su edad es de " + edad + " años.");
```

Se obtiene el mensaje en el monitor:

```
Su edad es de 8 años.
```

Lo que está entre comillas se muestra literalmente, mientras que `edad`, al no estar entrecomillado, se evalúa, mostrando su valor: 8.

Hay que tener en cuenta que `System.out.print` no inserta ningún retorno de carro al final del mensaje. El siguiente mensaje irá a continuación, sin dejar ningún tipo de separación. Un ejemplo de ello:

```
System.out.println("Hola."); //inserta un retorno de carro  
System.out.print("Encantando de conocerte."); //sin retorno de carro  
System.out.print("Adiós."); //sin retorno de carro
```

El resultado será:

```
Hola.  
Encantando de conocerte.Además.
```

Los dos últimos mensajes se encuentran unidos, ya que no existe ningún retorno de carro después de "Encantando de conocerte". Donde queramos insertarlo, usaremos

```
System.out.println()
```

o el carácter especial `'\n'`, que equivale a un retorno de carro.

```
System.out.println("Hola.");  
System.out.print("Encantando \nde \nconocerte."); //dos retornos de carro  
System.out.print("Adiós.");
```


Aparece en pantalla:

```
Hola
Encantado
de
conocer te. Adiós.
```

1.13.2. Entrada de datos

Otra operación muy utilizada, disponible en la API —mediante la clase **Scanner**—, consiste en recabar información del usuario a través del teclado. Cuando se hace de forma simple, en modo texto, sin ratón ni interfaz gráfica, se dice que obtenemos *datos por consola*.

Scanner es una clase de la API que se utiliza de forma no estática, es decir, necesita del operador **new**. Y la forma de trabajar con ella es siempre la misma: en primer lugar tendremos que crear un nuevo escáner,

```
Scanner sc = new Scanner(System.in);
```

System.in indica que vamos a leer del teclado. Una vez que hemos creado nuestro escáner, que hemos llamado **sc**, ya solo queda utilizarlo. Para ello disponemos de los métodos:

- **sc.nextInt()**: que lee un número entero (**int**) por teclado.
- **sc.nextDouble()**: lee un número real (**double**).
- **sc.nextLine()**: lee una cadena de caracteres hasta que se pulsa **Intro**.

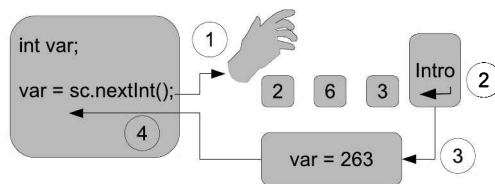


Figura 1.9. Entrada por consola

Las sentencias para introducir datos por teclado funcionan de la siguiente forma:

1. Se detiene la ejecución del programa y se espera a que el usuario teclee.
2. Recoge toda la secuencia tecleada hasta que se pulsa la tecla *Intro*.
3. Todo el contenido tecleado es interpretado y se asigna a una variable.
4. El programa dispone del dato introducido por el usuario en la variable.

Veamos un ejemplo completo de la lectura por consola de un número real:

```

Scanner sc = new Scanner(System.in); //creamos el nuevo escáner

double numero; //declaramos la variable numero
numero = sc.nextDouble(); //leemos por consola un valor double
//ahora disponemos del valor introducido, a través de la variable numero
System.out.println("Ha escrito: " + numero);

```

Este fragmento de código se utilizará como una fórmula literal cada vez que necesitemos introducir información por teclado. Según deseemos leer un entero, un real o una cadena de caracteres solo será necesario modificar el nombre y tipo de la variable a leer y el método de `sc`: `nextInt()`, `nextDouble()` o `nextLine()`.

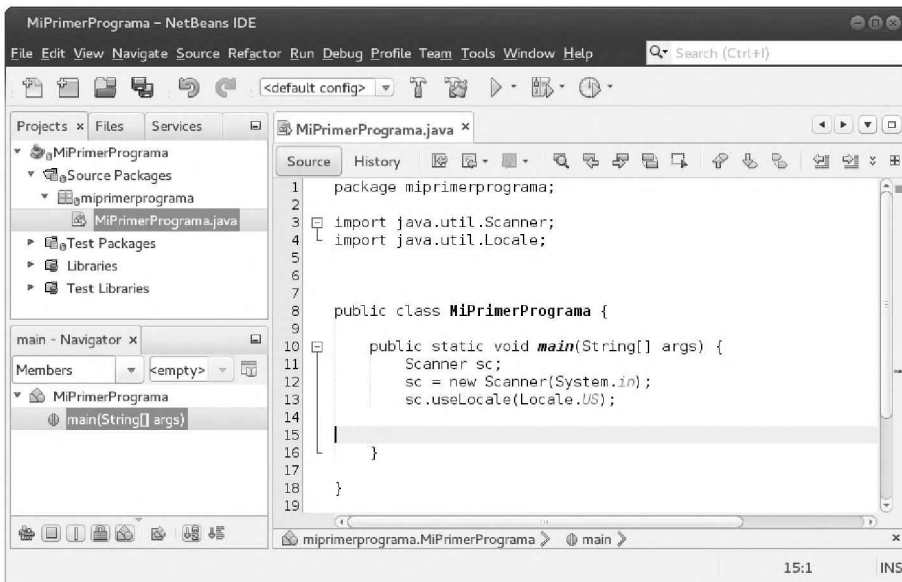


Figura 1.10. Programa principal con varios import

Para utilizar la clase `Scanner`, como hemos hecho en el ejemplo anterior, sin tener que escribir su nombre cualificado, la importaremos de la forma:

```
import java.util.Scanner;
```

Mientras `java.lang` contiene clases fundamentales a la hora de programar, es decir, es prácticamente imposible escribir un programa sin utilizar ninguna de ellas; el paquete `java.util` contiene clases muy útiles, pero no imprescindibles.

Una vez creado `sc` podemos utilizarlo para leer tantas veces como necesitemos,

```

Scanner sc = new Scanner(System.in);
edad = sc.nextInt(); //leemos un entero
precio = sc.nextDouble(); //leemos un real

```

Hay que tener en cuenta que, al utilizar `sc.nextDouble()`, que lee un número real por teclado, tenemos que introducir los números con coma decimal (,) —por ejemplo: 12,3—

en lugar de punto (.) —12.3—. Si queremos introducir los decimales con punto, debemos añadir la línea:

```
sc.useLocale(Locale.US);
```

justo después de crear el escáner. Para ello, antes debemos importar la clase `Locale`,

```
import java.util.Locale;
```

Para escribir nuestros programas utilizaremos siempre la estructura que se representa en la Figura 1.10. Finalmente, lo único que nos queda es seguir aprendiendo a programar con Java. Esto requiere practicar mucho sin dejar de disfrutar.

Ejercicios de conceptos básicos

- 1.1. Diseñar un programa que pida un número al usuario —por teclado— y a continuación lo muestre.

```
import java.util.Scanner;

/*
 * En este ejercicio se pide un número y después se muestra tal cual.
 * En este caso no procesamos el dato de entrada.
 * Esto no es un caso típico, pero nos sirve para ir mostrando las distintas
 * herramientas de las que disponemos.
 */
public class Main {

    public static void main(String[] args) {
        int num; //en la variable num guardaremos el número que se introduzca

        System.out.print("Escriba un número: "); //salida por consola: mensaje
        Scanner sc = new Scanner(System.in);

        num = sc.nextInt(); // entrada por consola
        //ahora mostraremos el valor de la variable num
        System.out.println("Valor introducido: " + num); //salida: mensaje + variable
        //utilizando + podemos concatenar en la salida por consola tantos
        //mensajes y variables como necesitemos
    }
}
```

- 1.2. Pedir al usuario su edad y mostrar la que tendrá el próximo año.

```
import java.util.Scanner;

/*
 * En el ejercicio realizamos las tres fases típicas de cualquier aplicación:
 * - Entrada de datos: pedimos la edad
 * - Procesado: en este caso incrementar la edad en 1
 * - Salida de datos: mostrar los resultados
 */
public class Main {

    public static void main(String[] args) {
        int edad; //aquí guardaremos la edad del usuario
        Scanner sc = new Scanner(System.in);
```

```
System.out.print("Escriba su edad: ");
edad = sc.nextInt();
edad = edad + 1; //el año que viene tendrá un año más
//la línea anterior puede sustuirse por: edad++;
//ahora mostraremos el valor de la variable edad
System.out.println("El próximo año su edad será: " + edad + " años");
}
}
```

- 1.3. Escribir una aplicación que pida el año actual y el de nacimiento del usuario. Debe calcular su edad, suponiendo que en el año en curso el usuario ya ha cumplido años.

```
import java.util.Scanner;

/*
 * La edad puede calcularse como la diferencia entre el año actual y el de
 * nacimiento. Esto puede contener un error, en el caso de que en la fecha
 * actual aun no se haya celebrado el cumpleaños del año en curso.
 * Supondremos que el cumpleaños del usuario ya ha tenido lugar este año.
 */
public class Main {

    public static void main(String[] args) {
        int aActual;    //año en curso (actual)
        int aNacimiento; //año de nacimiento
        int edad;
        Scanner sc = new Scanner(System.in);

        //leemos los datos
        System.out.print("Año de nacimiento: ");
        aNacimiento = sc.nextInt();
        System.out.print("Año actual: ");
        aActual = sc.nextInt();

        edad = aActual - aNacimiento; //calculamos la edad

        System.out.println("Su edad es: " + edad + " años.");
    }
}
```

- 1.4. El tipo `short` permite almacenar valores comprendidos entre -32 768 y 32 767. Se pide escribir un programa que compruebe que el rango de valores de un tipo se comporta de forma cíclica, es decir, el valor siguiente al máximo es el valor mínimo y viceversa.

```
/*
 * Veremos como Java evita que una operación provoque un desbordamiento. */
public class Main {

    public static void main(String[] args) {
        short num;
        num = 32767; //valor máximo dentro del rango de short
        System.out.println("Valor máximo para el tipo short: " + num);
        num++; //incrementamos en 1. Para evitar salirse del rango, la
        //variable num tomará el valor mínimo para el tipo short
        System.out.println("Valor mínimo para el tipo short: " + num);
    }
}
```

- 1.5. Necesitamos una aplicación que calcule la media aritmética de dos notas enteras. Hay que tener en cuenta que la media puede contener decimales.

```
import java.util.Scanner;

/*
 * Pediremos dos notas enteras y calcularemos la media. Como la media puede
 * tener decimales utilizaremos una variable de tipo real.
 */
public class Main {

    public static void main(String[] args) {
        int nota1, nota2; //variables enteras para las notas
        double media; //la media puede contener decimales: usamos double
        Scanner sc = new Scanner(System.in);

        //leemos las notas
        System.out.print("Nota 1: ");
        nota1 = sc.nextInt();
        System.out.print("Nota 2: ");
        nota2 = sc.nextInt();

        //calculamos la media
        media = (nota1 + nota2) / 2.0;
        //en la operación (nota1+nota2)/2.0, el punto decimal del 2 hace que
        //no sea una división entera. Con ello el resultado sufre una
        //conversión automática a real en doble precisión y conserva
        //la parte decimal

        System.out.println("La media es: " + media);
    }
}
```

- 1.6. Modificar el ejercicio anterior para que muestre la parte entera de la media de tres notas decimales.

```
import java.util.Scanner;

/*
 * Pediremos tres notas enteras y calcularemos la media. Como la media puede
 * tener decimales utilizaremos una variable real. */
public class Main {

    public static void main(String[] args) {
        int nota1, nota2, nota3, media; //variables para las notas y la media
        Scanner sc = new Scanner(System.in);

        System.out.print("Nota 1: "); //leemos las notas
        nota1 = sc.nextInt();
        System.out.print("Nota 2: ");
        nota2 = sc.nextInt();
        System.out.print("Nota 3: ");
        nota3 = sc.nextInt();

        //calculamos la media
        media = (int) ((nota1 + nota2 + nota3) / 3.0); //convertimos un valor
        //double en un valor int, truncando la parte decimal.
        //Por tanto, hay pérdida de información.

        System.out.println("La media es: " + media); //resultados
    }
}
```

- 1.7. Sería interesante disponer de un programa que pida como entrada un número decimal y lo muestre redondeado al entero más próximo.

```
import java.util.Locale;
import java.util.Scanner;

/*
 * Redondear un número decimal significa aproximarle al entero más cercano.
 * Para ello, lo que haremos será sumar 0.5 y truncar (eliminar los decimales)
 * el resultado. Así los números:
 *     2.3 se redondea a 2
 *     4.8 se redondea a 5
 */
public class Main {

    public static void main(String[] args) {
        double n; //aquí guardamos el número decimal introducido por el usuario
        int redondeo; //utilizamos esta variable para truncar los decimales
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US); //en lugar de coma utiliza punto para los decimales

        System.out.print("Escriba un número decimal (con punto): ");
        n = sc.nextDouble();
        //ahora redondearemos n
        redondeo = (int) (n + 0.5); //convertimos un real en un entero.
        //Esta es una conversión por estrechamiento, por lo tanto estamos
        //obligados a aplicar un cast (int). En caso de no hacerlo el
        //compilador generará un error.
        System.out.println(n + " redondeado es: " + redondeo);
    }
}
```

- 1.8. Un frutero necesita calcular los beneficios anuales que obtiene de la venta de manzanas y peras. Por este motivo, es necesario diseñar una aplicación que solicite las ventas (en kilos) de cada trimestre para cada fruta. La aplicación mostrará el importe total sabiendo que el precio del kilo de manzanas está fijado en 2.35 euros y el kilo de peras está fijado en 1.95 euros.

```
import java.util.Locale;
import java.util.Scanner;

/*
 * Los datos de entrada que necesitamos son:
 *     - cantidad vendida en el trimestre 1 (de peras y manzanas)
 *     - cantidad vendida en el trimestre 2 (idem)
 *     - cantidad vendida en el trimestre 3 (idem)
 *     - cantidad vendida en el trimestre 4 (idem)
 */
public class Main {

    public static void main(String[] args) {
        final double PRECIO_MANZANAS = 2.35; //valores constantes, ya que no tienen que
        final double PRECIO_PERAS = 1.95;    //variar a lo largo del programa.
        //los identificadores de constantes los escribimos en mayúsculas

        int vManza1t, vManza2t, vManza3t, vManza4t; //ventas (en kilos) por trimestre
        int vPeras1t, vPeras2t, vPeras3t, vPeras4t; //igual para las peras

        double impTotal; //importe total
        Scanner sc = new Scanner(System.in);
        sc = sc.useLocale(Locale.US);
    }
}
```

```

//pedimos los datos
System.out.println("Para las manzanas: ");
System.out.print("Ventas (en kilos) del primer trimestre: ");
vManza1t = sc.nextInt();
System.out.print("Ventas (en kilos) del segundo trimestre: ");
vManza2t = sc.nextInt();
System.out.print("Ventas (en kilos) del tercer trimestre: ");
vManza3t = sc.nextInt();
System.out.print("Ventas (en kilos) del cuarto trimestre: ");
vManza4t = sc.nextInt();
System.out.println("Para las peras: ");
System.out.print("Ventas (en kilos) del primer trimestre: ");
vPeras1t = sc.nextInt();
System.out.print("Ventas (en kilos) del segundo trimestre: ");
vPeras2t = sc.nextInt();
System.out.print("Ventas (en kilos) del tercer trimestre: ");
vPeras3t = sc.nextInt();
System.out.print("Ventas (en kilos) del cuarto trimestre: ");
vPeras4t = sc.nextInt();

//calculamos el importe total obtenido
impTotal = (vManza1t + vManza2t + vManza3t + vManza4t) * PRECIO_MANZANAS;
impTotal += (vPeras1t + vPeras2t + vPeras3t + vPeras4t) * PRECIO_PERAS;

System.out.println("El importe total es de: " + impTotal + " euros");
}
}

```

1.9. Los precios de la fruta no suelen ser constantes; varían según el mercado. Se pide modificar el ejercicio anterior para que los precios no estén fijados y sea la aplicación quien los pida al usuario.

```

import java.util.Locale;
import java.util.Scanner;

/*
 * Los datos de entrada que necesitamos son:
 *   - precios por kilos de manzanas y peras
 *   - cantidad vendida en cada trimestre (de peras y manzanas)
 */
public class Main {

    public static void main(String[] args) {
        double precioManzanas, precioPeras;

        int vManza1t, vManza2t, vManza3t, vManza4t; //ventas (kilos) por trimestre
        int vPeras1t, vPeras2t, vPeras3t, vPeras4t; //ventas (kilos) por trimestre

        double importeTotal;
        Scanner sc = new Scanner(System.in);
        sc = sc.useLocale(Locale.US);

        //pedimos los datos
        System.out.print("Precio del kilo de manzanas: ");
        precioManzanas = sc.nextDouble();

        System.out.print("Precio del kilo de peras: ");
        precioPeras = sc.nextDouble();

        System.out.print("Ventas (en kilos) del primer trimestre: ");
        vManza1t = sc.nextInt();
        System.out.print("Ventas (en kilos) del segundo trimestre: ");
        vManza2t = sc.nextInt();
    }
}

```

```
System.out.print("Ventas (en kilos) del tercer trimestre: ");
vManza3t = sc.nextInt();
System.out.print("Ventas (en kilos) del cuarto trimestre: ");
vManza4t = sc.nextInt();

System.out.println("Para las peras: ");
System.out.print("Ventas (en kilos) del primer trimestre: ");
vPeras1t = sc.nextInt();
System.out.print("Ventas (en kilos) del segundo trimestre: ");
vPeras2t = sc.nextInt();
System.out.print("Ventas (en kilos) del tercer trimestre: ");
vPeras3t = sc.nextInt();
System.out.print("Ventas (en kilos) del cuarto trimestre: ");
vPeras4t = sc.nextInt();

//calculamos el importe total obtenido
importeTotal = (vManza1t + vManza2t + vManza3t + vManza4t) * precioManzanas;
importeTotal += (vPeras1t + vPeras2t + vPeras3t + vPeras4t) * precioPeras;

System.out.println("El importe total es de " + importeTotal + " euros");
}
}
```

- 1.10. Diseñar una aplicación que calcule la longitud y el área de una circunferencia. Para ello, el usuario debe introducir el radio (que puede contener decimales). Recordamos:

$$\begin{aligned} \text{longitud} &= 2\pi \cdot \text{radio} \\ \text{área} &= \pi \cdot \text{radio}^2 \end{aligned}$$

Solución a)

```
import java.util.Locale;
import java.util.Scanner;

/*
 * Para calcular la longitud y el área fijaremos el valor de pi como
 * una constante.
 */
public class Main {

    public static void main(String[] args) {
        final double PI = 3.14; //pi como constante
        double radio; //el radio puede contener decimales
        double area, longitud; //hay que tener cuidado en no utilizar el
                                //identificador long: ya que es una palabra reservada
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US);
        // pedimos el radio
        System.out.print("Escriba el radio: ");
        radio = sc.nextDouble();

        longitud = 2 * PI * radio;
        area = PI * radio * radio; //radio*radio es el radio al cuadrado
        //para calcular el cuadrado también podemos utilizar:
        // Math.pow(radio, 2) que eleva radio al cuadrado (es decir, a 2)

        System.out.println("La longitud del círculo es: " + longitud);
        System.out.println("El área de la circunferencia es: " + area);
    }
}
```


Solución b)

```
import java.util.Locale;
import java.util.Scanner;

/*
 * Para calcular la longitud y el área utilizaremos el valor de pi que nos
 * brinda Math. La principal ventaja es que la precisión es mucho mayor.
 * Y utilizaremos un método de la API que eleva una base a un exponente para el cuadrado.
 */
public class Main {

    public static void main(String[] args) {
        double radio; //el radio puede contener decimales
        double area, longitud;

        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US);
        // pedimos los datos
        System.out.print("Escriba el radio: ");
        radio = sc.nextDouble();

        longitud = 2 * Math.PI * radio; //la clase Math pertenece al paquete
        //java.lang, que se importa por defecto
        area = Math.PI * Math.pow(radio, 2); //Math.pow(base, exponente) eleva la base
        //al exponente utilizado. Math.pow(radio, 2) eleva el radio a 2 (al cuadrado)

        System.out.println("La longitud de la circunferencia es: " + longitud);
        System.out.println("El área del círculo es: " + area);
    }
}
```

Ejercicios propuestos

1.1. Un economista nos ha encargado un programa para realizar cálculos con el IVA. La aplicación debe solicitar la base imponible y el IVA a aplicar. Debemos mostrar en pantalla el importe correspondiente al IVA y el total.

1.2. Escribir un programa que tome como entrada un número entero y nos indique qué cantidad hay que sumarle para que el resultado sea múltiplo de 7. Un ejemplo:

- A 2 hay que sumarle 5 para que el resultado ($2 + 5 = 7$) sea múltiplo de 7.
- A 13 hay que sumarle 1 para que el resultado ($13 + 1 = 14$) sea múltiplo de 7.

Si proporcionamos el número 2 o el 13, la salida de la aplicación debe ser 5 o 1, respectivamente.

1.3. Modificar el ejercicio anterior para que, indicando dos números **n** y **m**, nos diga qué cantidad hay que sumarle a **n** para que sea múltiplo de **m**.

1.4. Crear un programa que pida la base y la altura de un triángulo y muestre su área.

$$\text{área} = \frac{\text{base} \cdot \text{altura}}{2}$$

1.5. Dado un polinomio de segundo grado

$$y = ax^2 + bx + c,$$

crear un programa que pida los coeficientes a , b y c , así como el valor de x , y calcule el valor correspondiente de y .

1.6. Diseñar una aplicación que solicite al usuario que introduzca una cantidad de segundos. La aplicación debe mostrar cuántas horas, minutos y segundos hay en el número de segundos introducidos por el usuario.

1.7. Solicitar al usuario tres distancias:

- la primera, medida en milímetros;
- la segunda, medida en centímetros;
- y la última, medida en metros.

Diseñar un programa que muestre la suma de las tres longitudes introducidas (medida en centímetros).

1.8. Un biólogo está realizando un estudio de distintas especies de invertebrados y necesita una aplicación que le ayude a contabilizar el número de patas que tienen en total todos los animales capturados durante una jornada de trabajo. Para ello, nos ha solicitado que escribamos una aplicación a la que hay que proporcionar:

- el número de hormigas capturadas (6 patas),
- el número de arañas capturadas (8 patas),
- el número de cochinillas capturadas (14 patas).

La aplicación debe mostrar el número total de patas que poseen todos los animales.

Capítulo 2

Condicionales

Un programa no tiene por que ejecutar siempre la misma secuencia de instrucciones. Puede darse el caso en que, dependiendo del valor de alguna expresión o de alguna condición, interese ejecutar o evitar un conjunto de sentencias. Esta funcionalidad la brindan las sentencias `if`, `if-else` y `switch`.

2.1. Expresiones lógicas

En primer lugar, hablaremos un poco sobre los condicionales. Una *condición* no es más que el resultado de la evaluación de una expresión relacional y/o lógica. El valor de una condición siempre es de tipo booleano: verdadero o falso. En Java existen dos valores literales que representan estos valores: `true` y `false`.

La diferencia entre un operador relacional y uno lógico es que, mientras el primero utiliza como operandos expresiones numéricas, el segundo utiliza expresiones booleanas. Pero ambos generan valores booleanos.

2.1.1. Operadores relacionales

Los *operadores relacionales* son aquellos que comparan expresiones numéricas para generar valores booleanos. Recordemos que solo existen dos posibles valores: verdadero o falso. Los operadores relacionales disponibles son (Tabla 2.1):

Tabla 2.1. Operadores relacionales

Símbolo	Descripción
<code>==</code>	Igual que
<code>!=</code>	Distinto de
<code><</code>	Menor que
<code><=</code>	Menor o igual que
<code>></code>	Mayor que
<code>>=</code>	Mayor o igual que

Veamos algunos ejemplos de expresiones relacionales,

$a + b \leq 18$ será cierto si el valor de a más el valor de b es menor o igual que dieciocho.

Veamos dos posibles casos,

a) Con $a = 10$ y $b = 2 \Rightarrow 10 + 2 = 12 \leq 18$ la expresión es **true**.

b) Con $a = 20$ y $b = 1 \Rightarrow 20 + 1 = 21 \not\leq 18$ la expresión es **false**.

$2 * 5 == 10$ es siempre **true**, ya que dos por cinco siempre son diez.

$1 \neq 2$ siempre es **true** también, ya que 1 siempre es distinto de 2.

2.1.2. Operadores lógicos

Al utilizar los operadores lógicos podemos construir condiciones más complejas, ya que estos operadores utilizan y generan valores booleanos. Definamos el comportamiento de los operadores (Tabla 2.2).

Tabla 2.2. Operadores lógicos

Símbolo	Descripción
&&	Operador Y
	Operador O
!	Operador Negación

Operador &&: será cierto si ambos operandos son ciertos (Tabla 2.3).

Tabla 2.3. Tabla de verdad de &&

a	b	a && b
falso	falso	falso
cierto	falso	falso
falso	cierto	falso
cierto	cierto	cierto

Operador ||: es cierto si cualquiera de los operandos es cierto, como muestra la Tabla 2.4.

Tabla 2.4. Tabla de verdad de ||

a	b	a b
falso	falso	falso
cierto	falso	cierto
falso	cierto	cierto
cierto	cierto	cierto

Operador !: niega —cambia— el valor al que se aplica, convirtiendo **true** en **false** y viceversa, como se aprecia en la Tabla 2.5.

Tabla 2.5. Tabla de verdad de !

a	!a
falso	cierto
cierto	falso

Veamos algunos ejemplos. Vamos a suponer las siguientes expresiones:

$$\begin{aligned} a + b <= 18 &\Rightarrow \text{cierto} \\ a == 4 &\Rightarrow \text{falso} \end{aligned}$$

entonces:

$$!(a + b <= 18)$$

$$! \underbrace{(a + b <= 18)}_{\text{cierto}} \Rightarrow ! \text{cierto} = \text{falso}$$

resulta falso.

$$(a + b <= 18) \ \&\& \ (a == 4)$$

$$\underbrace{(a + b <= 18)}_{\text{cierto}} \ \&\& \ \underbrace{(a == 4)}_{\text{falso}} \Rightarrow \text{cierto} \ \&\& \ \text{falso} = \text{falso}$$

resulta falso, ya que el operador **&&** devuelve falso si cualquiera de los operandos también lo hace.

$$!(a + b <= 18) \ || \ (a == 4)$$

$$! \underbrace{(a + b <= 18)}_{\text{cierto}} \ || \ \underbrace{(a == 4)}_{\text{cierto}} \Rightarrow ! \text{cierto} \ || \ \text{falso} \Rightarrow \text{falso} \ || \ \text{falso} = \text{falso}$$

también es falso, ya que ambos operadores los son.

$$(a + b <= 18) \ || \ (\text{cualquier condición})$$

$$\underbrace{(a + b <= 18)}_{\text{cierto}} \ || \ \underbrace{(\text{cualquier condición})}_x \Rightarrow \text{cierto} \ || \ x = \text{cierto}$$

resulta cierto, debido a que el operador **||**, para devolver cierto, solo requiere que uno de los operandos sea cierto.

2.2. Condicional simple: if

La instrucción `if` proporciona un control sobre un conjunto de instrucciones que pueden ejecutarse o no, dependiendo de la evaluación de una condición. Los dos posibles valores (`true` o `false`) de esta determinan si el bloque de instrucciones de `if` se ejecuta (condición `true`) o no (condición `false`). La sintaxis es la siguiente:

```
if (condición) {  
    bloque de instrucciones  
    ...  
}
```

La Figura 2.1 nos muestra el flujo de control de un condicional simple, que funciona de la forma siguiente:

1. Se ejecutan las instrucciones anteriores a `if`.
2. Cuando le llega el turno a `if`, lo primero que ocurre es que se evalúa la condición. Del resultado de esta se obtienen dos posibles valores: `true` o `false`.
3. En caso de que la evaluación de la condición resulte `false`, no se ejecuta el bloque de instrucciones y se salta a la siguiente instrucción después de la estructura `if`.
4. Si, por el contrario, la evaluación de la condición es `true`, se ejecutará el bloque de instrucciones que contiene `if`. Este bloque de instrucciones puede albergar cualquier tipo de sentencia, incluido otro `if`.

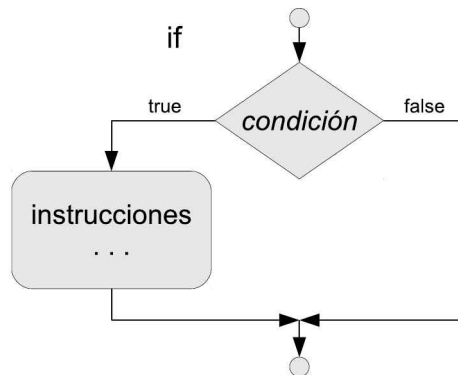


Figura 2.1. Funcionamiento de la instrucción `if`

Un bloque de instrucciones es un conjunto de sentencias delimitadas mediante llaves (`{}`). Dentro de un bloque de instrucciones es posible utilizar cualquier número de sentencias, e incluso definir otros bloques de instrucciones. También existe la posibilidad de declarar variables, que solo podrán ser utilizadas dentro del bloque donde se declaran. El lugar o bloque donde es posible utilizar una variable se denomina *ámbito de la variable*. Hasta ahora

hemos visto dos ámbitos de variables: las variables declaradas en `main`, que se denominan *variables locales*; y las variables declaradas en un bloque de instrucción, denominadas *variables de bloque*. Ambas son variables con idéntico funcionamiento; la distinción entre variables locales y de bloque reside en su ámbito,

Veamos un ejemplo:

```
a = 3;

if (a + 1 < 10) {
    a = 0;
    System.out.println("Hola");
}

System.out.println("El valor de a es: " + a);
```

Al evaluar la condición

`a + 1 < 10`

resulta

`3 + 1 < 10`

que da verdadero (`true`), ya que 4 es menor que 10. Al ser cierta la evaluación de la condición, se ejecutará el bloque de instrucciones de `if`:

- `a = 0`; se asigna a la variable `a` un valor cero.
- Se muestra en pantalla el mensaje "Hola".

Una vez terminado el bloque `if`, se continua con la siguiente instrucción, que muestra el mensaje:

El valor de a es 0

Veamos otro ejemplo similar con distinto resultado:

```
a = 9;

if (a + 1 < 10) {
    a = 0;
    System.out.println("Hola");
}

System.out.println ("El valor de a es: " + a);
```

En este caso, la condición resulta ser falsa, ya que diez no es menor que diez ($10 \not< 10$); en todo caso, son iguales. El bloque de instrucciones de `if` se ignora, ejecutando directamente la siguiente instrucción, que mostraría el mensaje:

El valor de a es 9

2.3. Condicional doble: if-else

Existe otra versión de la sentencia `if`, denominada **if-else**, donde se especifican dos bloques de instrucciones. El primero (*bloque true*) se ejecutará cuando la condición resulte verdadera y el segundo (*bloque false*) se ejecutará cuando la condición resulte falsa. Ambos bloques son mutuamente excluyentes, es decir, en cada ejecución de la instrucción `if-else` solo se ejecutará uno de ellos. La sintaxis es:

```
if (condición) {  
    bloque true    //se ejecuta cuando la condición es cierta  
} else {  
    bloque false   //se ejecuta cuando la condición es falsa  
}
```

La Figura 2.2 describe los posibles flujos de control de un programa que utilice un condicional doble.

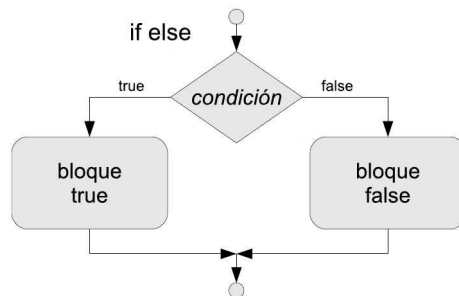


Figura 2.2. Funcionamiento de la instrucción `if-else`

Veamos un ejemplo:

```
if (a < 0) {  
    System.out.println("Valor negativo");  
} else {  
    System.out.println("Valor positivo");  
}
```

Para simplificar este ejemplo estamos considerando el número 0 como positivo. Supongamos que al evaluar la condición resulta cierta, lo que significa que la variable `a` tiene un valor menor que 0. Entonces se ejecuta el primer bloque de instrucciones mostrando el mensaje:

Valor negativo

Si al evaluar la condición (`a < 0`) resulta falsa, lo que significa que `a` tiene un valor igual o mayor a cero, se ignorará el primer bloque y se ejecutará el segundo, mostrando:

Valor positivo

2.3.1. Operador ternario

El operador ternario permite seleccionar un valor de entre dos posibles, dependiendo de la evaluación de una expresión, según sea **true** o **false**. La sentencia,

```
variable = expr ? valor1 : valor2
```

es equivalente a utilizar un condicional doble de la forma,

```
if (expr) {
    variable = valor1;
} else {
    variable = valor2;
}
```

Es recomendable, utilizar el operador ternario, por economía y legibilidad del código, en lugar de un **if-else**, cuando sea posible.

2.3.2. Anidación de condicionales

Cuando debemos realizar múltiples comprobaciones, podemos anidar tantos **if** o **if-else** como necesitemos, unos dentro de otros. La anidación de condicionales hace que las comprobaciones sean excluyentes, y resulta un código más eficiente. Veamos un ejemplo,

```
if (a - 2 == 1) {
    System.out.println("Hola ");
} else {
    if (a - 2 == 5) {
        System.out.println("Me ");
    } else {
        if (a - 2 == 8) {
            System.out.println("Alegro ");
        } else {
            if (a - 2 == 9) {
                System.out.println("De ");
            } else {
                if (a - 2 == 11) {
                    System.out.println("Conocerte.");
                } else {
                    System.out.println("Sin coincidencia");
                }
            }
        }
    }
}
```

Podemos aprovechar una cualidad que poseen tanto **if** como **if-else**, y es que cuando el bloque de instrucción del condicional está formado por una única sentencia, no es necesario utilizar llaves (**{ }**), aunque lo habitual es continuar utilizando llaves para delimitar el bloque de instrucción de **if** y eliminarlas de la construcción **else if**, ganando así en legibilidad. De esta manera, el ejemplo anterior podría reescribirse,

```
if (a - 2 == 1) {
    System.out.println("Hola ");
} else if (a - 2 == 5) {
    System.out.println("Me ");
} else if (a - 2 == 8) {
    System.out.println("Alegro ");
} else if (a - 2 == 9) {
    System.out.println("De ");
} else if (a - 2 == 11) {
    System.out.println("Conocerte.");
} else {
    System.out.println("Sin coincidencia");
}
```

2.4. Condicional múltiple: switch

En ocasiones, el hecho de utilizar muchos `if` o `if-else` anidados suele producir un código poco legible y difícil de mantener. Para estos casos Java dispone de la sentencia `switch`, cuya sintaxis es,

```
switch (expresión) {
    case expresión 1:
        conjunto instrucción 1
    case expresión 2:
        conjunto instrucción 2
    . . .
    case expresión N:
        conjunto instrucción N
    default:
        conjunto instrucción default
}
```

La evaluación de *expresión* debe dar un resultado entero, convertible en entero o un valor de tipo `String`¹. La cláusula `default` es opcional. Disponemos de la Figura 2.3 para describir el comportamiento de un condicional múltiple.

La dinámica del `switch` es la siguiente:

1. Evalúa *expresión* y obtiene su valor.
2. Comprueba si el valor obtenido coincide con el valor de la expresión del primer `case`, es decir, compara el valor de la expresión principal con el valor de *expresión 1*.
3. Si no coincide, sigue comprobando las expresiones de los siguientes `case`.
4. Si el valor de la expresión utilizado en alguna cláusula `case` coincide con el valor de la expresión principal, ejecuta el conjunto de instrucciones correspondiente y los sucesivos conjuntos de instrucciones de las instrucciones `case` que le siguen.

¹El tipo `String` se verá en el Capítulo 6.

- Si no existe coincidencia alguna, en caso de incluir el caso **default**, se ejecutará su conjunto de instrucciones.

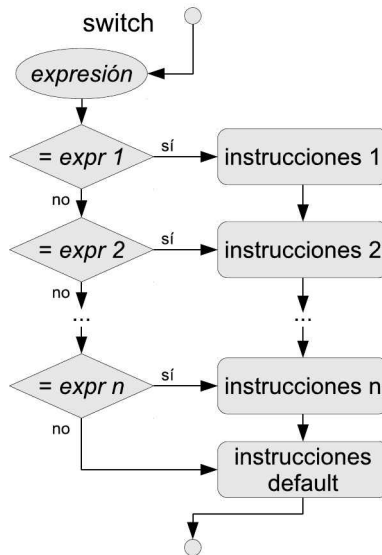


Figura 2.3. Funcionamiento de la instrucción **switch**

Veamos un ejemplo sencillo, donde las expresiones de los casos son simples números:

```

a = 10;

switch (a-2) {
    case 1:
        System.out.print("Hola ");
    case 5:
        System.out.print("Me ");
    case 8:
        System.out.print("Alegro ");
    case 9:
        System.out.print("De ");
    case 11:
        System.out.print("Conocerte. ");
    default:
        System.out.print("Sin coincidencia");
}
  
```

Veamos cómo se ejecuta **switch**:

1. Evaluamos la expresión principal, en este caso **a**, que vale 10, menos 2 resulta 8.
2. Se comprueba uno a uno el valor de cada **case**. En el ejemplo, el tercer **case** lleva asociado el valor 8, que coincide con el resultado de la evaluación de la expresión principal.

- Se ejecuta el conjunto de instrucciones desde el tercer **case** hasta el último, incluyendo la cláusula **default**.

El resultado final es:

Alegro De Conocerte. Sin coincidencia

Como puede verse, dependiendo del orden en el que coloquemos las cláusulas **case** se ejecutará un conjunto de instrucciones u otro. Java dispone de la sentencia **break** que, colocada al final de cada bloque de sentencias, impide que la ejecución continúe en el bloque siguiente. Esto nos permite hacer que solo se ejecute un bloque. De esta forma, es más sencillo escribir el **switch** sin tener que preocuparse del orden en el que se colocan los **case**. La Figura 2.4 describe los flujos de control si utilizamos **break**.

Se puede utilizar un sistema mixto, donde algunos bloques de instrucciones acaben en **break** y otros no, según nuestra conveniencia.

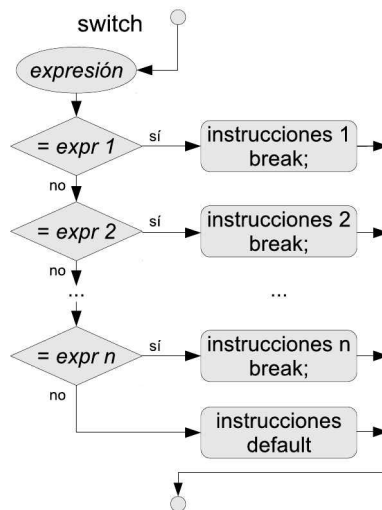


Figura 2.4. Funcionamiento de la instrucción **switch** con **break**

A continuación se muestra un ejemplo de código que usa **break** en todos los bloques:

```

a = 1;
switch (a*2) {
    case 1:
        System.out.println("Hola");
        break;
    case 2:
        System.out.println("Paco");
        break;
    case 3:
        System.out.println("Adiós");

```

```

        break;
    default:
        System.out.println("Sin coincidencia");
        break;
}

```

Nótese que el último **break** no es necesario pero, aparte de homogeneizar el código, facilita una futura modificación del programa por si decidimos añadir otros **case** al final.

En este ejemplo, el valor de la expresión principal del **switch** coincide con el segundo **case**. Se ejecuta el bloque de instrucciones asociado al segundo **case**, pero **break** impide que se ejecuten los siguientes. La salida por pantalla sería:

Paco

Veamos otro ejemplo:

```

a = 1;

switch (a*2) {
    case 1:
        System.out.println("Hola");
        break;
    case 2:
        System.out.println("Paco");
    case 3:
        System.out.println("Adiós");
        break;
    default:
        System.out.println("Sin coincidencia");
}

```

Este ejemplo es similar al anterior, pero hemos eliminado algunos **break**. Se ejecutará el bloque del segundo **case**, junto a los bloques de los siguientes **case**. La ejecución continuará hasta encontrar el **break** del tercer **case**. La salida que se obtiene es:

Paco
Adiós

Ejercicios de condicionales

2.1. Diseñar una aplicación que solicite al usuario un número e indique si es par o impar.

```

import java.util.Scanner;

/*
 * Vamos a introducir por teclado un número (entero). Para distinguir si es par o
 * impar comprobamos el resto de dividir por 2. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int num; //número introducido por el usuario
    }
}

```

```
        System.out.print("Introduzca un número: ");
        num = sc.nextInt();

        if (num % 2 == 0) { //si num es par
            System.out.println("Es par");
        } else { //es impar
            System.out.println("Es impar");
        }
    }
}
```

2.2. Pedir dos números enteros y decir si son iguales o no.

```
import java.util.Scanner;

/*
 * Leemos dos números enteros, que tendremos que comparar para decidir si son
 * iguales o distintos */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número: ");
        int n1 = sc.nextInt(); //primer número
        System.out.print("Introduzca otro número: ");
        int n2 = sc.nextInt();

        if (n1 == n2) { //si n1 es igual que n2
            System.out.println("Ambos números son iguales");
        } else {
            System.out.println("Los números son distintos");
        }
    }
}
```

2.3. Solicitar dos números distintos y mostrar cuál es el mayor.

```
import java.util.Scanner;

/*
 * Leemos dos números (en este caso enteros), que compararemos con el operador > */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número: ");
        int n1 = sc.nextInt();
        System.out.print("Introduzca otro número: ");
        int n2 = sc.nextInt();

        // el caso donde ambos números son iguales no se contempla e imprimiría
        // en pantalla que n2 es mayor que n1
        if (n1 > n2) {
            System.out.println(n1 + " es mayor que " + n2);
        } else {
            System.out.println(n2 + " es mayor que " + n1);
        }
    }
}
```

- 2.4. Realizar de nuevo el ejercicio anterior considerando el caso de que los números introducidos sean iguales.

```
import java.util.Scanner;

/*
 * En esta versión contemplamos la posibilidad que ambos números sean iguales.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número: ");
        int n1 = sc.nextInt();
        System.out.print("Introduzca otro número: ");
        int n2 = sc.nextInt();

        if (n1 == n2) {
            System.out.println("Son iguales");
        } else {
            //si no son iguales podemos decidir cuál es el mayor
            if (n1 > n2) {
                System.out.println(n1 + " es mayor que " + n2);
            } else {
                System.out.println(n2 + " es mayor que " + n1);
            }
        }
    }
}
```

- 2.5. Implementar un programa que pida por teclado un número decimal e indique si es un número *casi-cero*, que son aquellos, positivos o negativos, que se acercan a 0 por menos de 1 unidad, aunque curiosamente el 0 no se considera un número casi-cero. Ejemplos de números casi-cero son el 0.3, el -0.99 o el 0.123. Y números que no se consideran casi-ceros son: el 12.3, el 0 o el -1.

```
import java.util.Locale;
import java.util.Scanner;

/*
 * Un número casi-cero es el que se encuentra en el rango (-1, 1), donde se excluye el
 * -1, el 0 y el 1. Para comprobar si un número es casi-cero tendremos que utilizar
 * una condición con una expresión lógica.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US); //para utilizar punto (.) con los decimales

        System.out.print("Introduzca un número real positivo o negativo: ");
        double num = sc.nextDouble();
        //un casi-cero cumple: que es mayor que -1, que es menor que 1 y que no es 0
        if (-1 < num && num < 1 && num != 0) {
            System.out.println("Es un número casi-cero");
        } else {
            System.out.println("No es un casi-cero");
        }
    }
}
```

2.6. Pedir dos números y mostrarlos ordenados de forma decreciente.

```
import java.util.Scanner;

/*
 * Para ordenar dos números tendremos que compararlos. Es posible realizar este programa
 * utilizando if-else, pero en este caso vamos a hacerlo con el operador ternario. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n1, n2; //números introducidos por el usuario
        int mayor, menor; //variables que contendrán el mayor y el menor de entre n1 y n2

        System.out.print("Introduzca un número: ");
        n1 = sc.nextInt();
        System.out.print("Introduzca otro: ");
        n2 = sc.nextInt();

        mayor = n1 > n2 ? n1 : n2; //si n1 es mayor que n2, entonces mayor = n1, si no = n2
        menor = n1 < n2 ? n1 : n2; //si n1 es menor que n2, entonces menor = n1, si no = n2
        System.out.println(mayor + ", " + menor);
    }
}
```

2.7. Pedir tres números y mostrarlos ordenados de mayor a menor.

```
import java.util.Scanner;

/*
 * Supondremos que todos los números introducidos por teclado son distintos. Para
 * el caso de números iguales solo hay que utilizar el operador >=
 * Vamos a plantear tantos condicionales como casos existen con tres números.
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a, b, c; //número a ordenar

        System.out.print("Introduzca primer número: ");
        a = sc.nextInt();
        System.out.print("Introduzca segundo número: ");
        b = sc.nextInt();
        System.out.print("Introduzca tercer número: ");
        c = sc.nextInt();

        if (a > b && b > c) {
            System.out.println(a + ", " + b + ", " + c);
        } else if (a > c && c > b) {
            System.out.println(a + ", " + c + ", " + b);
        } else if (b > a && a > c) {
            System.out.println(b + ", " + a + ", " + c);
        } else if (b > c && c > a) {
            System.out.println(b + ", " + c + ", " + a);
        } else if (c > a && a > b) {
            System.out.println(c + ", " + a + ", " + b);
        } else if (c > b && b > a) {
            System.out.println(c + ", " + b + ", " + a);
        }
    }
}
```


- 2.8. Pedir los coeficientes de una ecuación de 2.º grado, y mostrar sus soluciones reales. Si no existen, debe indicarlo. Recordemos que las soluciones de una ecuación de segundo grado, $ax^2 + bx + c = 0$ son,

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

```
import java.util.Locale;
import java.util.Scanner;
/*
 * Para calcular las soluciones de una ecuación de segundo grado solo hay que aplicar
 * una sencilla fórmula. El único inconveniente es que hay que comprobar que no existan
 * divisiones por 0 o que no calculemos la raíz cuadrada de un número negativo.
 * Estos errores producen una parada de la ejecución del programa.
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US);
        double a, b, c; // coeficientes ax^2 + bx + c = 0
        double x1, x2, d; // soluciones y discriminante.

        System.out.print("Introduzca primer coeficiente (a): ");
        a = sc.nextDouble();
        System.out.print("Introduzca segundo coeficiente (b): ");
        b = sc.nextDouble();
        System.out.print("Introduzca tercer coeficiente (c): ");
        c = sc.nextDouble();

        // calculamos el discriminante
        d = (b * b - 4 * a * c);
        if (d < 0) { //hay que calcular la raíz cuadrada de d (d no puede ser negativo)
            System.out.println("No existen soluciones reales");
        } else {
            // si a=0 nos encontraríamos una división por cero. Y en este caso, ni siquiera
            //sería una ecuación de 2º grado
            if (a == 0) { //si a es igual a 0
                System.out.println("No es una ecuación de segundo grado");
            } else {
                x1 = (-b + Math.sqrt(d)) / (2 * a); //Math.sqrt() calcula la raíz cuadrada
                x2 = (-b - Math.sqrt(d)) / (2 * a);

                System.out.println("Solución 1: " + x1);
                System.out.println("Solución 2: " + x2);
            }
        }
    }
}
```

- 2.9. Escribir una aplicación que indique cuántas cifras tiene un número entero introducido por teclado, que estará comprendido entre 0 y 99.999.

```
import java.util.Scanner;
/*
 * Sabemos que los números comprendidos entre 0 y 9, inclusive, tienen una cifra.
 * Los números comprendidos entre 10 y 99, inclusive, tienen 2 cifras.
 * Los números comprendidos entre 100 y 999, inclusive, tienen 3 cifras.
 * Etc. */
```

```
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número entre 0 y 99.999: ");
        int num = sc.nextInt();

        if (num < 10) {
            System.out.println("Tiene 1 cifra");
        } else if (num < 100) {
            System.out.println("Tiene 2 cifras");
        } else if (num < 1000) {
            System.out.println("Tiene 3 cifras");
        } else if (num < 10000) {
            System.out.println("Tiene 4 cifras");
        } else if (num < 100000) {
            System.out.println("Tiene 5 cifras");
        }
    }
}
```

2.10. Pedir un número entre 0 y 9.999, y decir si es capicúa.

```
import java.util.Scanner;
/* Un número capicúa se lee igual de derecha a izquierda, que de izquierda a derecha.
 * Para comparar los guarismos de un número lo descompondremos en: decenas de millar(dm),
 * unidades de millar (um), centenas (c), decenas (d) y unidades (u).
 * Un número capicúa tiene las decenas de millar iguales que las unidades, y las unidades
 * de millar iguales a las decenas. Las centenas no son necesarias compararias con nada.
 * Pero hay que tener en cuenta que el número 121 (que es capicúa) al descomponerlo
 * resulta el número: 00121, por tanto, tendremos que tratar con especial atención
 * los números que, al descomponerlos, tienen ceros a la izquierda. Así el 121 o el 33
 * serán reconocidos como capicuas.
 * La idea es: sin tener en cuenta los ceros por la izquierda, comparar los guarismos
 * dos a dos por los extremos. Por ej: 121, se descompone 00121, detectar los ceros y
 * comparar los guarismos necesarios en cada caso.
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int dm, um, c, d, u; //almacenarán cada guarismo del número
        boolean capicua = false; //indica si el número es capicua. Suponemos que no lo es

        System.out.print("Introduzca un número entre 0 y 99.999: ");
        int num = sc.nextInt();

        // descomponemos el número dividiendo entre 10 y mediante el resto:
        u = num % 10; // unidades. Ej: 12345 % 10 = 5
        num = num / 10; //eliminamos las unidades del número. Ej: 12345 / 10 = 1234

        d = num % 10; // decenas. Ej: 1234 % 10 = 4
        num = num / 10; //volvemos a eliminar el último guarismo. Ej: 1234 / 10 = 123

        c = num % 10; // centenas. Ej: 123 % 10 = 3
        num = num / 10; //eliminamos el último guarismo. Ej: 123 / 10 = 12

        um = num % 10; // unidades de millar. Ej 12 % 10 = 2
        num = num / 10; //eliminamos el últimoguarismo. Ej: 12 / 10 = 1

        dm = num; // decenas de millar. El número solo contiene las decenas de millar
```

```
// si el número tiene 5 cifras (dm, um, c, d, u)
if (dm != 0 && dm == u && um == d) {
    capicua = true;
}
// si el número tiene 4 cifras (0, um, c, d, u)
if (dm == 0 && um != 0 && um == u && c == d) {
    capicua = true;
}
// si el número tiene 3 cifras (0, 0, c, d, u)
if (dm == 0 && um == 0 && c != 0 && c == u) {
    capicua = true;
}
// si el número tiene 2 cifras (0, 0, 0, d, u)
if (dm == 0 && um == 0 && c == 0 && d != 0 && d == u) {
    capicua = true;
}

// se entiende que un número de una cifra no es capicúa

if (capicua) {
    System.out.println("El número es capicúa");
} else {
    System.out.println("El número NO es capicúa");
}
}
}
```

2.11. Pedir una nota de 0 a 10 y mostrarla de la forma: Insuficiente (de 0 a 4), Suficiente (5), Bien (6), Notable (7 y 8) y Sobresaliente (9 y 10).

Solución a)

```
import java.util.Scanner;
/*
 * Hay alumnos que no están muy de acuerdo con esta clasificación de las notas, y toda
 * calificación mayor a 3 debe ser Notable. :-)
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int nota;

        System.out.print("Introduzca una nota: ");
        nota = sc.nextInt();

        if (0 <= nota && nota < 5) { //se podría utilizar 0<=nota && nota<=4
            System.out.println("Insuficiente");
        } else if (nota == 5) {
            System.out.println("Suficiente");
        } else if (nota == 6) {
            System.out.println("Bien");
        } else if (nota == 7 || nota == 8) { //si nota es 7 u 8
            System.out.println("Notable");
        } else if (nota == 9 || nota == 10) { //si nota es 9 o 10
            System.out.println("Sobresaliente");
        } else {
            System.out.println("Error: nota no válida");
        }
    }
}
```

Solución b)

```
import java.util.Scanner;
/*
 * Vamos a resolver el ejercicio utilizando una estructura switch en lugar de if's
 * anidados, aprovechando la propiedad de que en un switch, se ejecuta un case y
 * los siguientes hasta que encontremos un break o se llegue al final del bloque. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca una nota: ");
        int nota = sc.nextInt();

        switch (nota) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
                System.out.println("Insuficiente");
                break;
            case 5:
                System.out.println("Suficiente");
                break;
            case 6:
                System.out.println("Bien");
                break;
            case 7:
            case 8:
                System.out.println("Notable");
                break;
            case 9:
            case 10:
                System.out.println("Sobresaliente");
                break;
            default:
                System.out.println("Error: nota no válida");
                break; //el último break no es necesario, pero es buena costumbre ponerlo
        }
    }
}
```

2.12. Pedir el día, mes y año de una fecha e indicar si la fecha es correcta. Recordamos que existen meses con 28, 30 y 31 días. No consideraremos los años bisiestos.

```
import java.util.Scanner;

/*
 * Hay que tener en cuenta que no todos los meses tienen el mismo número de días. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int dia, mes, año;
        boolean fechaCorrecta; //bandera que indica si la fecha es correcta.
        System.out.print("Introduzca día: ");
        dia = sc.nextInt();
        System.out.print("Introduzca mes: ");
        mes = sc.nextInt();
        System.out.print("Introduzca año: ");
        año = sc.nextInt();
```

```

if (año == 0) { // el único año que no existe es el 0
    fechaCorrecta = false;
} else {
    // primero comprobaremos febrero (mes = 2)
    if (mes == 2 && (1 <= día && día <= 28)) {
        fechaCorrecta = true;
    } else // veremos si es un mes de 30 días
    if ((mes == 4 || mes == 6 || mes == 9 || mes == 11)
        && (1 <= día && día <= 30)) {
        fechaCorrecta = true;
    } else // comprobaremos si es un mes de 31 días
    if ((mes == 1 || mes == 3 || mes == 5 || mes == 7 || mes == 8
        || mes == 10 || mes == 12)
        && (1 <= día && día <= 31)) {
        fechaCorrecta = true;
    } else { //en cualquier otro caso
        fechaCorrecta = false;
    }
}
}
if (fechaCorrecta) {
    System.out.println("Fecha correcta: " + día + "/" + mes + "/" + año);
} else {
    System.out.println("Fecha incorrecta");
}
}
}

```

2.13. Escribir un programa que pida una hora de la forma: hora, minutos y segundos. El programa debe mostrar la hora un segundo más tarde. Un ejemplo:

hora actual [10:41:59] → hora actual +1 segundo [10:42:00]

```

import java.util.Scanner;

/*
 * Supondremos que la hora introducida por el usuario es correcta. La idea del algoritmo
 * es incrementar los segundos en 1. Esto puede provocar que salgan del rango 0..59, en
 * este caso, pondremos los segundos a 0 e incrementamos los minutos. Igualmente tenemos
 * que comprobar que los minutos no se salgan de rango. E igual para las horas.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int h, m, s; // horas, minutos y segundos

        System.out.print("Introduzca hora: ");
        h = sc.nextInt();
        System.out.print("Introduzca minutos: ");
        m = sc.nextInt();
        System.out.print("Introduzca segundos: ");
        s = sc.nextInt();

        s++; // incrementamos los segundos

        if (s > 59) { //si los segundos superan 59
            s = 0; //los reiniciamos a 0
            m++; //e incrementamos los minutos

            if (m > 59) { //si los minutos superan 59,
                m = 0; //los reiniciamos
                h++; //e incrementamos la hora
            }
        }
    }
}

```

```
        if (h > 23) { //si la hora supera 23
            h = 0; //reiniciamos la hora a 0
        }
    }
}
System.out.println("Hora + 1 segundo: " + h + ":" + m + ":" + s);
}
}
```

2.14. Crear una aplicación que solicite al usuario una fecha (día, mes y año) y muestre la fecha correspondiente al día siguiente.

```
import java.util.Scanner;
/*
 * Este ejercicio es similar al anterior, en el que incrementábamos la hora. En este caso
 * la dificultad es que no todos los meses tienen el mismo número de días. Por eso, lo
 * primero que haremos es ver cuantos días tiene el mes de la fecha.
 * No tendremos en cuenta los años bisiestos y suponemos correcta la fecha introducida.*/
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int diasDelMes=0; // Aquí guardaremos el número de días que tiene el mes

        System.out.print("Introduzca día: ");
        int dia = sc.nextInt();
        System.out.print("Introduzca mes: ");
        int mes = sc.nextInt();
        System.out.print("Introduzca año: ");
        int año = sc.nextInt();

        // suponemos que la fecha introducida es correcta
        if (mes == 2) { //febrero tiene 28 días
            diasDelMes = 28;
        }
        if (mes == 4 || mes == 6 || mes == 9 || mes == 11) { //estos meses tienen 30 días
            diasDelMes = 30;
        }
        if (mes == 1 || mes == 3 || mes == 5 || mes == 7 || mes == 8
            || mes == 10 || mes == 12) { //y estos meses tienen 31 días
            diasDelMes = 31;
        }

        dia++; // incrementamos el día

        if (dia > diasDelMes) { //si día supera el número de días del mes
            dia = 1; //reiniciamos día a 1
            mes++; //e incrementamos el mes

            if (mes > 12) { // si mes supera 12
                mes = 1; //lo reiniciamos a 1
                año++; //e incrementamos el año
            }
        }
        // hay que tener en cuenta que el año pasó del -1 al +1. El año 0 nunca existió.
        // Para evitar que el año pase del -1 al 0
        if (año == 0) {
            año = 1;
        }

        System.out.println(dia + "/" + mes + "/" + año);
    }
}
```

- 2.15.** Idear un programa que solicite al usuario un número comprendido entre 1 y 7, correspondiente a un día de la semana. Se debe mostrar el nombre del día de la semana al que corresponde. Por ejemplo, el número 1 corresponde a «lunes» y el 6 a «sábado».

```
import java.util.Scanner;
/*
 * Utilizaremos switch, para distinguir las distintas alternativas.
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número de 1 a 7: ");
        int dia = sc.nextInt();

        switch (dia) {
            case 1:
                System.out.println("lunes");
                break;
            case 2:
                System.out.println("martes");
                break;
            case 3:
                System.out.println("miércoles");
                break;
            case 4:
                System.out.println("jueves");
                break;
            case 5:
                System.out.println("viernes");
                break;
            case 6:
                System.out.println("sábado");
                break;
            case 7:
                System.out.println("domingo");
                break;
        }
    }
}
```

- 2.16.** Solicitar un número comprendido entre 1 y 99. El programa debe mostrarlo escrito. Por ejemplo, para 56 mostrar: «cincuenta y seis».

```
import java.util.Scanner;
/*
 * Descompondremos el número en unidades y decenas. Mostraremos la decenas y a continua-
 * ción las unidades. Por ejemplo, 56: decenas 5, mostramos "cincuenta", y las unidades
 * 6, mostramos "y seis". El problema es que el 11 se muestra como "dieci y uno".
 * Habrá que considerar los números que no siguen la regla general: 11, 12, 13, 14...
 * Estos casos excepcionales (del 11 al 15) se considerarán aparte, pero por no alargar
 * la solución del ejercicio no se consideran otras excepciones: 22 (veintidos)...
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca un número (0 a 99): ");
        int num = sc.nextInt();
```

```
if (10 <= num && num <= 15) {//vemos los casos excepcionales
    switch (num) {
        case 10:
            System.out.println("diez");
            break;
        case 11:
            System.out.println("once");
            break;
        case 12:
            System.out.println("doce");
            break;
        case 13:
            System.out.println("trece");
            break;
        case 14:
            System.out.println("catorce");
            break;
        case 15:
            System.out.println("quince");
            break;
    }
} else { //estamos en un caso general
    int unidades = num % 10; // descomponemos el número
    int decenas = num / 10;

    switch (decenas) {
        case 0:
            System.out.print(""); //no mostramos nada
            break;
        case 1:
            System.out.print("dieci");
            break;
        case 2:
            System.out.print("veinte");
            break;
        case 3:
            System.out.print("treinta");
            break;
        case 4:
            System.out.print("cuarenta");
            break;
        case 5:
            System.out.print("cincuenta");
            break;
        case 6:
            System.out.print("sesenta");
            break;
        case 7:
            System.out.print("setenta");
            break;
        case 8:
            System.out.print("ochenta");
            break;
        case 9:
            System.out.print("noventa");
            break;
    }
    if (decenas != 0 && decenas != 1 // "y" se muestra si las decenas no son 0 ni 1
        && unidades != 0) { //tampoco se muestra cuando las unidades son 0
        System.out.print(" y ");
    }

    switch (unidades) {
        case 0:
            if (decenas == 0) { //para el caso 0
                System.out.println("cero"); // ya mostramos como decenas "cero"
            }
    }
}
```



```

        break;
    case 1:
        System.out.println("uno");
        break;
    case 2:
        System.out.println("dos");
        break;
    case 3:
        System.out.println("tres");
        break;
    case 4:
        System.out.println("cuatro");
        break;
    case 5:
        System.out.println("cinco");
        break;
    case 6:
        System.out.println("seis");
        break;
    case 7:
        System.out.println("siete");
        break;
    case 8:
        System.out.println("ocho");
        break;
    case 9:
        System.out.println("nueve");
        break;
    }
}
}
}

```

Ejercicios propuestos

2.1. Sabiendo que para calcular la letra de un documento nacional de identidad el algoritmo es el siguiente:

- Obtener el módulo 23 del número del DNI.
- Según el módulo obtenido y la siguiente tabla, se asigna la letra correspondiente al DNI:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	→
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	→

→	15	16	17	18	19	20	21	22
→	S	Q	V	H	L	C	K	E

Diseñar una aplicación en la que, dado un número de DNI, calcule la letra que le corresponde.

2.2. En una granja se compra diariamente una cantidad (`comidaDiaria`) de comida para los animales. El número de animales a alimentar (todos de la misma especie) es `numAnimales`, y sabemos que cada animal come una media de `kilosPorAnimal`.

Diseñar un programa que solicite al usuario los valores anteriores y determine si disponemos de alimento suficiente para cada animal. En caso negativo, ha de calcular cuál es la ración que corresponde a cada uno de los animales.

Nota: Evitar que la aplicación realice divisiones por cero.

Capítulo 3

Bucles

Un bucle es un tipo de estructura que contiene un bloque de instrucciones que se ejecuta repetidas veces; cada ejecución o repetición del bucle se llama *iteración*. El uso de bucles simplifica la escritura de programas, minimizando el código duplicado. Cualquier fragmento que se necesite ejecutar varias veces seguidas es susceptible de incluirse en un bucle. Java dispone de los bucles: **while**, **do-while** y **for**.

3.1. Bucles controlados por condición

El control del número de iteraciones se lleva a cabo mediante una condición. Si la evaluación de la condición es cierta el bucle realizará una nueva iteración.

3.1.1. **while**

Al igual que el comportamiento de **if**, **while** depende de la evaluación de una condición. El bucle **while** solamente decide si realizar una nueva iteración basándose en el valor de la condición. Su sintaxis es:

```
while (condición) {  
    bloque de instrucciones  
    ...  
}
```

El comportamiento de este bucle (*véase* Figura 3.1) es:

1. Se evalúa *condición*.
2. Si la evaluación resulta **false**: terminamos la ejecución del bucle.
3. Si, por el contrario, la condición es **true**: se ejecuta el bloque de instrucciones.
4. Tras ejecutarse el bloque de instrucciones, se vuelve al primer punto.

Un bucle **while** puede realizar cualquier número de iteraciones, desde cero, cuando la primera evaluación de la condición resulta falsa, a infinitas, en el caso que la condición

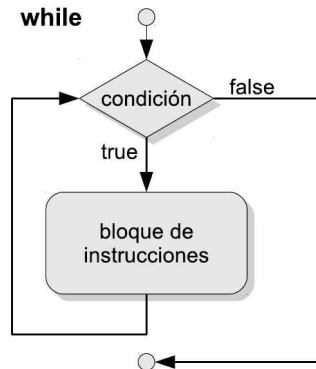


Figura 3.1. Bucle while

sea cierta y las variables que la controlan nunca tomen valores que la hagan falsa. Esto es lo que se conoce como *bucle infinito*.

Veamos dos ejemplos:

```

edad = 10;
while (edad > 18) {
    ...
}

```

En este caso, independientemente del bloque de instrucciones asociado a la estructura **while**, no se llega a entrar nunca en el bucle, debido a que la condición exige que **edad > 18** y, con un valor de **edad = 10**, no se cumple. Se realizan cero iteraciones. En cambio,

```

edad = 20;
while (edad >= 18) {
    System.out.println ("Es usted mayor de edad.");
}

```

Hagamos una traza (seguimiento) del fragmento de código anterior:

1. Se asigna **edad = 20**.
2. Evaluamos la condición del **while**: ¿es cierto que $edad \geq 18$? Es decir: ¿ $20 \geq 18$?
3. Cierto. Mostramos el mensaje: **Es usted mayor de edad**.
4. Evaluamos la condición del **while**: ¿es cierto que $edad \geq 18$? Es decir: ¿ $20 \geq 18$?
5. Cierto. Mostramos el mensaje: **Es usted mayor de edad**.
6. Continuamos indefinidamente.

Dentro del bloque de instrucciones no hay nada que modifique la variable **edad**, lo que hace que la condición permanezca idéntica, evaluándose siempre **true** y haciendo que el bucle sea infinito.

3.1.2. do-while

Disponemos de un segundo bucle controlado por una condición: el bucle **do-while**, muy similar al **while**, con la diferencia de que primero se ejecuta el bloque de instrucciones y después se evalúa la condición para decidir si se realiza una nueva iteración. Su sintaxis es:

```
do {  
    bloque de instrucciones  
    ...  
} while (condición);
```

La Figura 3.2 describe su comportamiento.

1. Se ejecuta el bloque de instrucciones.
2. Se evalúa *condición*.
3. Según el valor obtenido, se termina el bucle o comenzamos en el primer punto.

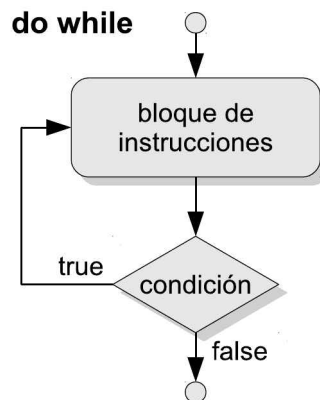


Figura 3.2. Bucle do-while

Debemos recordar que es el único bucle que finaliza en punto y coma (;). Mientras el bucle **while** se puede ejecutar de 0 a infinitas veces, el **do-while** lo hace de 1 a infinitas veces. De hecho, la única diferencia con el bucle **while** es que **do-while** se ejecuta, al menos, una vez.

3.2. Bucles controlados por contador: for

El bucle **for** permite controlar el número de iteraciones mediante una variable contador. La sintaxis de la estructura **for** es:

```
for (inicialización; condición; incremento) {  
    bloque de instrucciones  
    ...  
}
```

Donde *inicialización* es una o más asignaciones de variable, *condición* es una expresión booleana que controla las iteraciones del bucle e *incremento* es el aumento a aplicar a las variables que controlan la condición.

El funcionamiento de `for` se describe en la Figura 3.3.

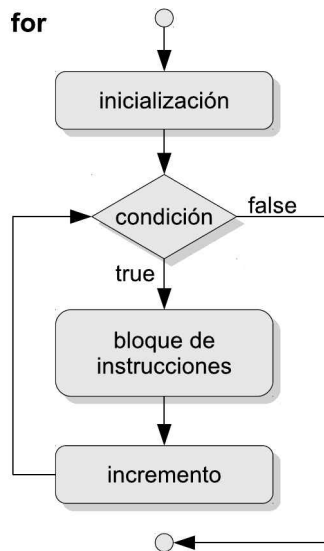


Figura 3.3. Bucle `for`

1. Se ejecuta la inicialización; esto se hace una sola vez al principio.
2. Se evalúa la condición: si resulta **false**, salimos del bucle y continuamos con el resto del programa; en caso de que la evaluación sea **true**, se ejecuta todo el bloque de instrucciones.
3. Cuando termina de ejecutarse el bloque de instrucciones se ejecuta el incremento.
4. Se vuelve de nuevo al segundo punto.

Aunque `for` está controlado por una condición que, en principio, puede ser cualquier expresión booleana, la posibilidad de configurar la inicialización y el incremento permiten determinar, de antemano, el número de iteraciones que va a tener el bucle controlado mediante variables contador. Veamos un ejemplo,

```
for (i = 1; i <= 2; i++) {  
    System.out.println("La i vale " + i);  
}
```

Un traza de la ejecución del bucle anterior:

1. Primero se ejecuta la inicialización: `i=1`;
2. Evaluamos la condición: ¿es cierto que $i \leq 2$? Es decir: $1 \leq 2$?
3. Cierto. Ejecutamos el bloque de instrucción: `System.out.println(...)`.
4. Obtenemos el mensaje: La i vale 1.
5. Terminado el bloque de instrucciones ejecutamos el incremento: `i++`; $\Rightarrow$ i vale 2.
6. Evaluamos la condición: ¿es cierto que $i \leq 2$? Es decir: $2 \leq 2$?
7. Cierto. Ejecutamos `System.out.println(...)`.
8. Obtenemos el mensaje: La i vale 2.
9. Ejecutamos el incremento: `i++`; $\Rightarrow$ la i vale 3.
10. Evaluamos la condición: ¿es cierto que $i \leq 2$? Es decir: $3 \leq 2$?
11. Falso. El bucle termina y continúa la ejecución de las sentencias que siguen a la estructura `for`.

3.3. Salidas anticipadas

Dependiendo de la lógica a implementar en un programa, puede ser interesante terminar un bucle con anterioridad y no esperar que finalice por la condición (realizando todas las iteraciones). Para poder hacer esto disponemos de:

- **break**, que finaliza completamente el bucle, y
- **continue**, detiene la iteración actual, y continua con la siguiente.

Cualquier programa puede escribirse sin utilizar **break** ni **continue**, incluso algunos autores recomiendan evitarlos, ya que rompen la secuencia natural de las instrucciones. Veamos un ejemplo:

```
i = 1;
while (i <= 10) {
    System.out.println("La i vale" + i);
    if (i == 2) {
        break;
    }
    i++;
}
```

En un primer vistazo da la impresión de que el bucle ejecutará 10 iteraciones, pero cuando está realizando la segunda (*i* vale 2), la condición de `if` se evalúa como cierta y entra en juego `break`, que interrumpe completamente el bucle, sin que se ejecute las sentencias restantes de la iteración, ni el resto de las iteraciones, con lo que solo se realizan dos iteraciones y se obtiene:

```
La i vale 1
La i vale 2
```

Veamos otro ejemplo:

```
i = 1;
while (i <= 10) {
    i++;
    if (i % 2 == 0) { //si i es par
        continue;
    }
    System.out.println("La i vale " + i);
}
```

Cuando la condición `i % 2 == 0` sea cierta, es decir, cuando *i* es par, la sentencia `continue` detiene la iteración actual y continua con la siguiente, saltándose el resto del bloque de instrucciones. `System.out.println` solo llegará a ejecutarse cuando *i* sea impar, o dicho de otro modo: en iteraciones alternas. Se obtiene la salida por consola:

```
La i vale 1
La i vale 3
La i vale 5
La i vale 7
La i vale 9
```

3.4. Bucles anidados

En el uso de los bucles es muy frecuente la anidación, que consiste en incluir un bucle dentro de otro, como describe la Figura 3.4.

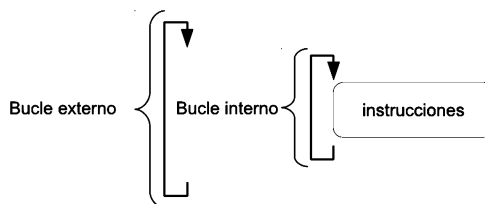


Figura 3.4. Bucles anidados

Al hacer esto se multiplica el número de veces que se ejecuta el bloque de instrucciones de los bucles internos. Los bucles anidados pueden encontrarse relacionados cuando los

valores de las variables de los bucles más externos intervienen en el control de la iteración de un bucle interno, o independientes, cuando no existe relación alguna entre ellos, siendo el número de iteraciones de un bucle ajeno a los valores de las variables utilizadas en otro bucle, ya sea externo o interno.

3.4.1. Bucles independientes

Cuando los bucles anidados no dependen, en absoluto, unos de otros para determinar el número de iteraciones, se denominan *bucles anidados independientes*. Veamos un ejemplo sencillo: la anidación de dos bucles `for`:

```
for (i = 1; i <= 4; i++) {
    for (j = 1; j <= 3; j++) {
        System.out.println("Ejecutando...");
    }
}
```

El bucle externo, controlado por la variable `i`, realizará cuatro iteraciones, donde `i` toma los valores 1, 2, 3 y 4. El bucle interno, controlado por `j`, realizará tres iteraciones, tomando `j` los valores 1, 2 y 3.

¿Cuántas veces se ejecutará la sentencia `System.out.println("Ejecutando...");`? Mientras el bucle externo está en su primera iteración (`i` vale 1), el bucle interno realiza sus 3 iteraciones; `j` toma los valores: 1, 2 y 3. Durante la segunda iteración del bucle externo (`i` vale 2), el bucle interno realiza otras tres iteraciones, tomando de nuevo `j` los valores: 1, 2 y 3. Para las siguientes dos iteraciones del bucle externo, donde `i` toma los valores 3 y 4 consecutivamente, el bucle interno realizará de nuevo tres iteraciones en ambos casos.

Tabla 3.1. Valores de las variables que controlan el bucle

Bucle externo (i)	Bucle interno (j)
i = 1	j = 1
	j = 2
	j = 3
i = 2	j = 1
	j = 2
	j = 3
i = 3	j = 1
	j = 2
	j = 3
i = 4	j = 1
	j = 2
	j = 3

El bucle externo realiza 4 vueltas, y el interno, 3 por cada vuelta del externo; lo que resulta $4 \times 3 = 12$ ejecuciones del bloque de instrucciones más interno. Obtendremos 12 veces el mensaje:

```
Ejecutándose...
```

Un nuevo ejemplo es:

```
for (i = 1; i <= 2; i++) {
    System.out.println("Bucle externo, i=" + i);
    for (j = 1; j <= 3; j++) {
        System.out.println("...Bucle medio, j=" + j);
        for (k = 1; k <= 4; k++) {
            System.out.println(".....Bucle interno, k=" + k);
        }
    }
    System.out.println("");
}
```

La salida que se obtiene es:

Bucle externo, i=1	Bucle externo, i=2
...Bucle medio, j=1	...Bucle medio, j=1
.....Bucle interno, k=1	Bucle interno, k=1
.....Bucle interno, k=2	Bucle interno, k=2
.....Bucle interno, k=3	Bucle interno, k=3
.....Bucle interno, k=4	Bucle interno, k=4
...Bucle medio, j=2	...Bucle medio, j=2
.....Bucle interno, k=1	Bucle interno, k=1
.....Bucle interno, k=2	Bucle interno, k=2
.....Bucle interno, k=3	Bucle interno, k=3
.....Bucle interno, k=4	Bucle interno, k=4
...Bucle medio, j=3	...Bucle medio, j=3
.....Bucle interno, k=1	Bucle interno, k=1
.....Bucle interno, k=2	Bucle interno, k=2
.....Bucle interno, k=3	Bucle interno, k=3
.....Bucle interno, k=4	Bucle interno, k=4

En el caso anterior: el bucle *i* se ejecuta 2 veces, el bucle *j* lo hace 3 veces y el bucle *k* realiza 4 iteraciones, resultando $2 \times 3 \times 4 = 24$ iteraciones para el bloque de instrucciones del bucle más interno. El conjunto de instrucciones del bucle medio se ejecuta un total de $2 \times 3 = 6$ veces. Y el bloque de instrucciones del bucle externo se ejecuta solo 2 veces.

Anidar bucles es una herramienta que facilita el procesado, por ejemplo, de tablas multidimensionales. Se utiliza cada nivel de anidación para manejar los índices de la dimensión correspondiente. Sin embargo, el uso descuidado de bucles anidados puede convertir un algoritmo en algo ineficiente, disparando el número de instrucciones ejecutadas.

3.4.2. Bucles dependientes

Puede darse el caso que el número de iteraciones de un bucle interno no sea independiente de la ejecución de los bucles exteriores, y dependa de sus variables de control. Decimos entonces que son *bucles anidados dependientes*. Veamos el siguiente fragmento de código, a modo de ejemplo, donde las variables utilizadas en el bucle externo (*i*) se toma

como base para comparar con los valores de la variable (j) que controla el bucle más interno. En algunas ocasiones, la dependencia de los bucles no se aprecia de forma tan clara como en el ejemplo:

```
for (i = 1; i <= 3; i++) {
    System.out.println("Bucle externo, i=" + i);

    j = 1;
    while (j <= i) {
        System.out.println("...Bucle interno, j=" + j);
        j++;
    }
}
```

que proporciona la salida:

```
Bucle externo, i=1
...Bucle interno, j=1
Bucle externo, i=2
...Bucle interno, j=1
...Bucle interno, j=2
Bucle externo, i=3
...Bucle interno, j=1
...Bucle interno, j=2
...Bucle interno, j=3
```

- Durante la primera iteración del bucle i, el bucle interno realiza una sola iteración.
- En la segunda iteración del bucle externo con i que vale 2, el bucle interno realiza dos iteraciones.
- En la última vuelta, cuando i vale 3, el bucle interno se ejecuta tres veces.

La variable i controla el número de iteraciones del bucle interno y resulta un total de $1 + 2 + 3 = 6$ iteraciones. Los posibles cambios en el número de iteraciones de estos bucles hacen que a priori no sea tan fácil conocer el número total de iteraciones.

Ejercicios de bucles

- 3.1.** Diseñar un programa que muestre, para cada número introducido por teclado, si es par, si es positivo y su cuadrado. El proceso se repetirá hasta que el número introducido por teclado sea 0.

```
import java.util.Scanner;
/*
 * A priori no tenemos la certeza de cuántas veces se ejecutará el informe, por lo tanto,
 * los únicos bucles que se adaptan son while o do-while. Recordamos que la diferencia
 * entre ellos es que uno se ejecuta 0 o más veces, y el otro al menos 1 vez.
 * En este caso, si se introduce un 0 como primer número, el informe no se ejecuta.
 * Utilizaremos un bucle while. */
```

```
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        boolean esPar, esPositivo; //indicadores para el informe
        System.out.print("Introduzca número: ");
        int num = sc.nextInt(); //leemos el número

        while (num != 0) { //repetimos mientras el número leído no sea 0
            //si divido un número entre 2 y obtengo como resto 0, significa que es par
            //el operador % (resto módulo) calcula el resto. Así sabremos la paridad
            esPar = num % 2 == 0 ? true : false; // si el resto es 0, será par
            esPositivo = num >= 0 ? true : false; //consideramos el 0 positivo

            System.out.println("Es par?: " + esPar + "\nEs positivo?: " + esPositivo);
            System.out.println("Cuadrado: " + num * num);
            System.out.print("Introduzca otro número: ");

            num = sc.nextInt(); // volvemos a leer num
        }
    }
}
```

- 3.2.** Un centro educativo nos ha pedido que diseñemos una aplicación para calcular algunos datos estadísticos de las edades de los alumnos. Se introducirán datos hasta que uno de ellas sea negativo. La aplicación mostrará la suma de todas las edades, la media, de cuántos alumnos hemos introducido las edades y cuántos alumnos son mayores de edad. Implementar la aplicación requerida.

```
import java.util.Scanner;
/*
 * Como no sabemos cuántas edades se van a utilizar como datos, utilizaremos un bucle
 * while. En cada iteración acumularemos la edad, incrementaremos un contador para llevar
 * la cuenta de las edades introducidas y dependiendo si el alumno es mayor de edad,
 * incrementaremos el contador de alumnos mayores de edad.
 * Cuando salgamos del bucle mostraremos los datos y calcularemos la media. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int sumaEdades = 0; //acumulará la suma de todas las edades
        int contadorAlumnos = 0, //contador de alumnos (o de edades introducidas)
            contadorMayorEdad = 0; //contador del número de alumnos mayores de edad
        double media; //media de las edades

        System.out.print("Introduzca edad: ");
        int edad = sc.nextInt(); //leemos la edad

        while (edad >= 0) { //repetimos mientras la edad no sea negativa
            sumaEdades += edad; //acumulamos la edad introducida
            contadorAlumnos++; //incrementamos, se ha introducido la edad de un alumno más
            if (edad >= 18) { //si la edad introducida corresponde a un mayor de edad
                contadorMayorEdad++; //incrementamos, ahora hay un mayor de edad más
            }
            System.out.print("Introduzca edad: ");
            edad = sc.nextInt(); // volvemos a leer
        }
        media = (double) sumaEdades / contadorAlumnos; // con el cast la división es real

        //mostramos el informe estadístico
        System.out.println("Suma de todas las edades: " + sumaEdades);
        System.out.println("Media: " + media);
    }
}
```

```

        System.out.println("Número total de alumnos: " + contadorAlumnos);
        System.out.println("Mayores de edad: " + contadorMayorEdad);
    }
}

```

- 3.3.** Realizar el juego *el número secreto*, que consiste en acertar un número desconocido (generado aleatoriamente entre 1 y 100). Para ello se leen por teclado una serie de números, para los que se indica: «mayor» o «menor», según sea mayor o menor con respecto al número secreto. El proceso termina cuando el usuario acierta o cuando se rinde (introduciendo un -1).

```

import java.util.Scanner;
/*
 * La aplicación generará un número aleatorio entre 1 y 100. A continuación el jugador
 * irá probando suerte con la ayuda de las indicaciones que la propia aplicación le
 * ofrece. El juego termina cuando acierta o cuando se rinde (introduciendo un -1). */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int numSecreto = (int) (Math.random() * 100 + 1); //número aleatorio entre 1 y 100

        System.out.print("Introduzca un número entre 1 y 100: ");
        int num = sc.nextInt();

        while (numSecreto != num && // mientras no acertemos (ambos números son distintos)
            num != -1) { // y no introduzcamos un -1
            if (numSecreto < num) { //si el número secreto es menor que el introducido
                System.out.println("Menor");
            } else { //en otro caso, será mayor
                System.out.println("Mayor");
            }
            System.out.print("Introduzca otro número: ");
            num = sc.nextInt();
        }
        //salimos del bucle por: el jugador acierta el número o se rinde
        if (numSecreto == num) {
            System.out.println("Enhorabuena...");
        } else {
            System.out.println("Otra vez será...");
        }
    }
}

```

- 3.4.** Escribir una aplicación para aprender a contar, que pedirá un número n y mostrará todos los números del 1 a n .

```

import java.util.Scanner;
/*
 * Sabemos con certeza el número de iteraciones del bucle: n, por lo que utilizaremos
 * un bucle for que recorrerá todos los números de 1 a n.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número: ");
        int n = sc.nextInt();
    }
}

```

```
        for (int i = 1; i <= n; i++) {  
            //i tomará los valores de 1 a n, en cada iteración la  
            //variable i es una variable del bloque de instrucciones del for, es decir,  
            //solo se puede utilizar en dicho bloque (su ámbito es el bloque)  
            //intentar utilizar la variable i fuera del bloque genera un error  
            System.out.println(i); //mostramos i  
        }  
    }  
}
```

- 3.5.** Desarrollar un programa que solicite los valores mínimo y máximo de un rango. A continuación solicitará por teclado un número que debe estar dentro del rango. Si el valor introducido no pertenece al rango, la aplicación volverá a pedir otro valor, y así repetidas veces, hasta que el valor se encuentre dentro del rango.

```
import java.util.Scanner;  
/*  
 * Primero solicitamos los valores extremos del rango. A continuación, utilizaremos un  
 * bucle do-while para pedir un número y verificar que pertenece al rango. Una propiedad  
 * del bucle do-while es que su bloque de instrucciones se ejecuta al menos una vez.  
 */  
public class Main {  
  
    public static void main(String[] args) {  
        int n, min, max; //variables  
        Scanner sc = new Scanner(System.in);  
  
        System.out.print("Valor mínimo del rango: ");  
        min = sc.nextInt(); //supondremos que min será menor que max  
        System.out.print("Valor máximo del rango: ");  
        max = sc.nextInt();  
  
        do {  
            System.out.print("Escriba un número: ");  
            n = sc.nextInt();  
        } while (!(min <= n && n <= max)); //mientras n no pertenezca al rango  
  
        System.out.println(n + " pertenece al rango"); //al salir del bucle, tenemos la  
        //certeza que n pertenece al rango min..max  
    }  
}
```

- 3.6.** Escribir todos los múltiplos de 7 menores que 100.

```
/*  
 * Vamos a utilizar un bucle for, inicializando la i a 0, e iterando hasta que el valor  
 * supere 100. Los múltiplos de 7, se caracterizan por que se diferencian en 7.  
 */  
public class Main {  
  
    public static void main(String[] args) {  
        for (int i = 0; i < 100; i += 7) {  
            System.out.println(i);  
        }  
        // cuando el bloque de instrucciones de for, while o do-while está  
        // formado por una sola instrucción, no precisa de llaves { }  
        // aunque, por claridad en el código se aconseja ponerlas  
    }  
}
```

3.7. Diseñar un programa que muestre el producto de los 10 primeros números impares.

```
/*
 * Para calcular los 10 primeros números impares utilizamos un bucle for que:
 * - comience en 1
 * - en cada vuelta se incremente en 2; así obtenemos:
 *   1, 3, 5, 7, 9, 11, 13, 15, 17 y 19
 * - la i del bucle for debe ser <= 19 (o <20 o <10*2) */
public class Main {

    public static void main(String[] args) {
        double producto = 1; // guardará la multiplicación de los 10 primeros números
        // impares. Es muy importante acordarse de inicializarlo a 1. Ya que si lo hacemos
        // a 0, el producto siempre valdrá 0 (por estar multiplicando).
        for (int i = 1; i < 20; i += 2) {
            producto = producto * i; //equivalente a: producto *= i
        }
        System.out.println("El producto de los 10 primeros impares es: " + producto);
    }
}
```

3.8. Pedir un número y calcular su factorial. Por ejemplo, el factorial de 5 se denota 5! y es igual a $5 \times 4 \times 3 \times 2 \times 1 = 120$.

```
import java.util.Scanner;
/*
 * El factorial de n se define como el producto de todos los enteros entre 1 y n.
 * Por ejemplo: el factorial de 10 es: 10*9*8*7*6*5*4*3*2*1 = 3628800 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        //podríamos declarar long factorial, pero con una variable de tipo long se puede
        //calcular hasta el factorial de 25. Mejor utilizamos un double
        double factorial;
        int num;

        System.out.print("Introduzca un número: ");
        num = sc.nextInt();

        factorial = 1; // es importante inicializarlo a 1, ya que multiplicará
        for (int i = num; i > 0; i--) {
            factorial = factorial * i;
        }
        System.out.println("El factorial de " + num + " es: " + factorial);
    }
}
```

3.9. Un centro de investigación de la flora urbana necesita una aplicación que muestre cuál es el árbol más alto. Para ello se introducirá por teclado la altura (en centímetros) de cada árbol (terminando cuando se utilice -1 como altura). Los árboles se identifican mediante etiquetas con números únicos correlativos, comenzando en 0. Se pide diseñar una aplicación que resuelva el problema planteado.

```
import java.util.Scanner;
/*
 * Introducimos la altura de cada árbol dentro de un bucle y guardaremos la mayor y el
 * número de etiqueta del árbol al que corresponde.
```

```
* En la búsqueda del máximo (o mínimo) se nos plantea un problema: con qué valor
* inicializamos el máximo. Hemos de inicializar el máximo con un valor menor o igual a
* algunos de los valores con los que trabajaremos.
* Un ejemplo: si se desea calcular el máximo de {-2, -7 y -4}, está claro que el
* máximo es -2, pero si inicializamos de forma arbitraria el máximo a 0, al ser 0 mayor
* que cualquier valor del conjunto, el algoritmo dirá que el máximo es 0. Algo erróneo.
* En este caso, como trabajamos con alturas (todos positivos), podríamos inicializar
* sin problema a 0, ya que 0 es menor que cualquier positivo. Sin embargo, en el caso
* general, la mejor opción es inicializar el máximo al primer valor leído. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int etiquetaArbolMasAlto, //número identificador del árbol más alto
            alturaArbolMasAlto; //altura del árbol más alto

        //como no sabemos cuántos árboles hay, utilizaremos un while
        int etiqueta = 0; //número identificador del árbol del que se piden los datos
        int altura; //altura del árbol del que se piden los datos

        System.out.print("Altura del árbol (" + etiqueta + "): ");
        altura = sc.nextInt();
        alturaArbolMasAlto = altura; //el primer árbol sera, por ahora, el más alto
        etiquetaArbolMasAlto = 0; //el árbol con etiqueta 0 es, por ahora, el más alto
        while (altura != -1) {
            if (altura > alturaArbolMasAlto) { //hemos encontrado un árbol más alto
                alturaArbolMasAlto = altura;
                etiquetaArbolMasAlto = etiqueta;
            }
            etiqueta++; //incrementamos la etiqueta, para solicitar la altura del siguiente
            System.out.print("Altura del árbol (" + etiqueta + "): ");
            altura = sc.nextInt();
        }

        if (alturaArbolMasAlto == -1) {
            System.out.println("No hay ningún árbol");
        } else {
            System.out.println("El árbol más alto mide: " + alturaArbolMasAlto);
            System.out.println("Corresponde al árbol con etiqueta:" + etiquetaArbolMasAlto);
        }
    }
}
```

- 3.10.** Se desea implementar una aplicación que pida al usuario que introduzca un número comprendido entre 1 y 10. Debemos mostrar la tabla de multiplicar de dicho número. El código tendrá que asegurarse de que el número introducido se encuentra entre el 1 y el 10.

```
import java.util.Scanner;
/*
 * Las tablas de multiplicar nos traen recuerdos de nuestros tiempos de escolares, cuando
 * intentábamos aprenderlas (recitándolas una y otra vez). */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int num; //del que mostraremos la tabla de multiplicar

        //nos aseguramos de que el número está entre 1 y 10
        do {
            System.out.print("Introduzca un número (de 1 a 10): ");
            num = sc.nextInt();
        } while (!(1 <= num && num <= 10));
    }
}
```



```

        System.out.println("\n\nTabla del " + num);
        for (int i = 1; i <= 10; i++) {
            System.out.println(num + " x " + i + " = " + num * i);
        }
    }
}

```

3.11. Diseñar una aplicación que muestre las tablas de multiplicar del 1 al 10.

```

/*
 * Ya tenemos un algoritmo (en el ejercicio anterior) para realizar la tabla de multi-
 * plicar de un número dado. La idea es aprovecharlo, y ejecutar el código repetidas
 * veces para mostrar las tablas de multiplicar del 1 al 10. */
public class Main {

    public static void main(String[] args) {
        for (int tabla = 1; tabla <= 10; tabla++) {
            System.out.println("\n\nTabla del " + tabla);

            // por cada iteración del bucle exterior, el interior se ejecuta 10 veces
            for (int i = 1; i <= 10; i++) {
                System.out.println(tabla + " x " + i + " = " + tabla * i);
            }
        }
    }
}

```

3.12. Pedir 5 calificaciones de alumnos y decir al final si hay algún suspenso.

```

import java.util.Scanner;

/*
 * Utilizamos una bandera para controlar si entre los alumnos existe al menos uno con una
 * asignatura suspensa (nota menor que 5). Una bandera es una variable, normalmente
 * booleana, que indica, mediante sus valores, alguna situación o estado. En este caso:
 * suspenso = false, significa que no existe ninguna nota suspensa
 * suspenso = true, significa que existe, al menos, un alumno suspenso
 * Hay que tener cuidado, cuando se activa una bandera, en no volver a desactivarla, ya
 * que entonces no refleja lo que intentamos evaluar, sino la última situación ocurrida.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        boolean suspensos = false; // suponemos que en principio no hay ningún suspenso

        for (int i = 0; i < 5; i++) {
            System.out.print("Introduzca nota (de 0 a 10): ");
            int notas = sc.nextInt();

            if (notas < 5) { //si la nota corresponde a un suspenso
                suspensos = true; //activamos la bandera a cierto
            }
        }

        if (suspensos) {
            System.out.println("Hay alumnos suspensos");
        } else {
            System.out.println("No hay suspensos");
        }
    }
}

```

- 3.13.** Dadas 6 notas, escribir la cantidad de alumnos aprobados, condicionados (=4) y suspensos.

```
import java.util.Scanner;

/*
 * Utilizaremos contadores que se incrementan cuando nos encontramos en una situación
 * concreta: la nota está aprobada, está condicionada o está suspensa.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int aprobados = 0, suspensos = 0, condicionados = 0; //contadores

        for (int i = 1; i <= 6; i++) {
            System.out.println("Nota del alumno número " + i + ": ");
            int nota = sc.nextInt();

            if (nota == 4) { //comprobaremos en que caso nos encontramos
                condicionados++;
            } else if (nota >= 5) {
                aprobados++;
            } else if (nota < 4) { //este if es redundante, al ser el único caso posible
                suspensos++;      //y podríamos poner else {...}
            }
        }
        System.out.println("Aprobados: " + aprobados); //mostramos el informe
        System.out.println("Suspensos: " + suspensos);
        System.out.println("Condicionados: " + condicionados);
    }
}
```

- 3.14.** Pedir por consola un número n y dibujar un triángulo rectángulo de n elementos de lado, utilizando para ello asteriscos (*). Por ejemplo, para $n = 4$,

```
* * * *
* * *
* *
*

import java.util.Scanner;

/*
 * Utilizaremos un bucle para mostrar cada fila, y dentro de este, otro para escribir
 * cada * (columna). El bucle que escribe cada columna (dentro de la fila) dependerá
 * de los valores de la fila, así conseguimos el efecto "triángulo". */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Escriba n: ");
        int n = sc.nextInt();

        for (int fila = 1; fila <= n; fila++) {
            for (int col = fila; col <= n; col++) { //el número de * coincide con: n-fila+1
                System.out.print("* "); //no println, para que no cambie de línea
            }
            System.out.println(""); //tras cada fila metemos una nueva línea
        }
    }
}
```

- 3.15.** Realizar un programa que nos pida un número n , y nos diga cuántos números hay entre 1 y n que sean primos. Un número primo es aquél que solo es divisible por 1 y por él mismo. Veamos un ejemplo para $n = 8$:

comprobamos todos los números del 1 al 8	{	1 → primo
		2 → primo
		3 → primo
		4 → no primo
		5 → primo
		6 → no primo
		7 → primo
		8 → no primo

Resultando un total de 5 números primos.

```
import java.util.Scanner;
/*
 * Para un número dado (i), calcularemos si es primo: para ello comprobaremos que no es
 * divisible por ningún número perteneciente al rango 2..i-1.
 * El algoritmo anterior se repetirá para cada número entre 1 y n. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int j, contadorPrimos = 0;
        boolean primo; //bandera que indica si un número es primo

        System.out.print("Escriba n: ");
        int n = sc.nextInt();

        // vamos procesando todos los números entre 1 y n
        for (int i = 1; i <= n; i++) { //vamos a comprobar si i es primo
            primo = true; // suponemos inicialmente que i es primo
            j = 2; // j contendrá los valores del rango 2..i-1

            // el bucle while puede iterar menos vueltas. Consultar algoritmos para primos.
            //Saldremos del while cuando tengamos la certeza de que i no es primo o cuando
            //terminemos de comprobar todos los números, en cuyo caso i es primo
            while (j <= i - 1 && primo == true) {
                if (i % j == 0) { //si i es divisible por j
                    primo = false; //si es dividible, no puede ser primo
                }
                j++;
            }

            if (primo) { // si es primo
                contadorPrimos++; // incrementamos el contador de primos
                System.out.println(i + (" es primo")); // y mostramos
            }
        }
        System.out.println("De 1 a " + n + ", hay " + contadorPrimos + " números primos");
    }
}
```

Ejercicios propuestos

- 3.1.** Implementar un programa que pida al usuario un número de tres cifras y lo muestre guarismo a guarismo. Por ejemplo, para el número 123, debe mostrar primero el 1, a continuación el 2 y por último el 3.
- 3.2.** Escribir un programa que incremente la hora de un reloj. Se pedirán por teclado la hora, los minutos y segundos, así como cuántos segundos se desea incrementar la hora introducida. La aplicación mostrará la nueva hora. Por ejemplo, si las 13:59:51 se incrementan en 10 segundos resultan las 14:00:01.
- 3.3.** Implementar la aplicación *eco*, que pide al usuario un número y muestre en pantalla la salida:

```
Eco...  
Eco...  
Eco...
```

que muestre "Eco..." tantas veces como indique el número introducido. La salida anterior sería para el número 3.

- 3.4.** Calcular la raíz cuadrada de un número natural mediante aproximaciones. En el caso de que no sea exacta, mostraremos el resto. Por ejemplo, para calcular $\sqrt{23}$, probamos $1^2 = 1$, $2^2 = 4$, $3^2 = 9$, $4^2 = 16$, $5^2 = 25$ (nos pasamos) resultando 4 la raíz cuadrada de 23 con un resto ($23 - 16$) de 7.
- 3.5.** Diseñar una aplicación que dibuje el triángulo de Pascal, para n filas. Numerando las filas y elementos desde 0, la fórmula para obtener el m -ésimo elemento de la n -ésima fila es:

$$E(n, m) = \frac{n!}{m! \times (n - m)!}$$

donde $n!$ es el factorial de n .

Un ejemplo de triángulo de Pascal con 5 filas ($n=4$)

```
1  
1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1
```

Capítulo 4

Funciones

Conforme aumenta el tamaño de un programa —medido en líneas de código fuente— es habitual tener que utilizar, en distintas partes, la misma funcionalidad, cosa que implica copiar una y otra vez, donde sea necesario, el mismo fragmento de código. Esto genera dos problemas:

- Duplicidad del código: aumenta el tamaño del programa y lo hace menos legible.
- Dificultad en el mantenimiento: cualquier modificación necesaria dentro del fragmento de código repetido tendría que realizarse en todos y cada uno de los lugares donde se encuentra.

Podríamos pensar en utilizar un bucle, pero si el código se repite en lugares separados del programa, esto no es posible.

4.1. Conceptos básicos

La solución es utilizar una función, que no es más que etiquetar un fragmento de código con un nombre y sustituir en el programa los fragmentos de código repetidos por el nombre que le hemos asignado. Esta idea puede verse en el siguiente ejemplo,

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

    System.out.println("Voy a saludar tres veces:"); //fragmento repetido
    for(int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }

    ... //más código
    System.out.println("Voy a saludar tres veces:"); //fragmento repetido
    for(int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }
}
```

```
... //otro código
System.out.println("Voy a saludar tres veces:"); //fragmento repetido
for(int i = 0; i < 3; i++) {
    System.out.println("Hola.");
}
... //resto del código
}
```

Cada fragmento de código repetido a lo largo del programa, con la misma funcionalidad —en nuestro ejemplo mostrar una serie de mensajes por consola—, puede sustituirse por una función, quedando,

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

    tresSaludos(); //sustitución por una función
    ... //más código

    tresSaludos(); //sustitución por una función
    ... //otro código

    tresSaludos(); //sustitución por una función
    ...//resto del código
}
```

Para poder utilizar la función hace falta definirla; esto puede hacerse antes o después del `main()`,

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

static void tresSaludos() {
    System.out.println("Voy a saludar tres veces:");
    for(int i = 0; i < 3; i++) {
        System.out.println("Hola.");
    }
}
```

En general, la sintaxis para definir una función es:

```
static tipo nombreFunción() {
    cuerpo de la función
}
```

Por ahora, definiremos las funciones `static`¹ y utilizaremos como tipo de la función `void`, que indica que la función no devuelve nada². Habitualmente, los nombres de funciones siguen el estilo *Camel*: los nombres comienzan en minúscula, distinguiendo en los nombres

¹En el Capítulo 7, donde veremos las clases, se explicará en profundidad el concepto de función o método estático.

²En la Sección 4.4 se verá con detalle el tipo devuelto por una función.

compuestos cada palabra mediante la mayúscula inicial, lo que recuerda las jorobas de un camello; algunos ejemplos son:

`suma(), tresSaludos(), calculaRaizCuadrada(), muestraTodosDatosCliente()...`

Y *cuerpo de la función* es el bloque (entre llaves) de instrucciones que implementa la función.

Definimos algunos conceptos necesarios para seguir trabajando con funciones:

Llamada a la función: el nombre de la función, seguido de () —paréntesis— se convierte en una nueva instrucción que podemos utilizar para invocarla.

Prototipo de la función: es la declaración de la función, donde se especifica su nombre, el tipo que devuelve y los parámetros que utiliza. En nuestro ejemplo, el prototipo de la función `tresSaludos()` es,

```
static void tresSaludos()
```

Cuerpo de la función: el bloque de código que ejecuta la función cada vez que se invoca.

Definición de una función: está formada por el prototipo más el cuerpo de la función.

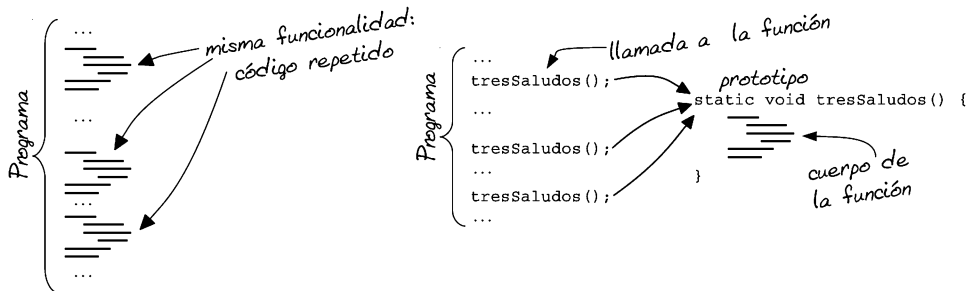


Figura 4.1. Representación de un programa con y sin funciones

De forma esquemática, la Figura 4.1 representa un programa en el que no se utilizan funciones (a la izquierda) y el mismo programa en el que se ha utilizado una función. Se puede apreciar que, en el segundo caso, el cuerpo de la función solo está escrito una vez. Con esto evitamos:

- La duplicidad del código: ya que el código se escribe una única vez, en la definición de la función.
- La dificultad en el mantenimiento: ahora, las modificaciones, en el caso de que sean necesarias, solo se realizan en un lugar: en la definición de la función.

El comportamiento de una llamada a una función, (véase la Figura 4.2), es:

1. Las instrucciones del programa se ejecutan hasta que encuentra la llamada a la función, en nuestro caso, `tresSaludos()`;

2. La ejecución salta a la definición de la función.
3. Se ejecuta el cuerpo de la función.
4. Cuando la ejecución del cuerpo termina, retornamos al punto del programa desde donde se invocó la función.
5. El programa continúa su ejecución.

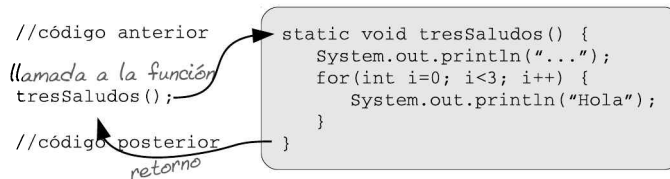


Figura 4.2. Llamada a la función

Tanto en la llamada a la función como en el retorno es posible incluir información útil. Estos flujos de información se verán a lo largo del capítulo.

Hemos estado utilizando funciones sin percatarnos desde que comenzamos a programar. Por ejemplo, `System.out.println()`, que permite mostrar un texto en pantalla. Si no dispusiéramos de ella, cada vez que necesitáramos mostrar un mensaje nos veríamos obligados a escribir las diferentes líneas de código que la componen.

4.2. Ámbito de las variables

En el cuerpo de una función podemos declarar variables, que se conocen como *variables locales*. El ámbito de estas, es decir, donde pueden utilizarse, es la propia función donde se declaran, no pudiéndose utilizar fuera de ella. Nada impide que dentro del cuerpo de una función se utilicen sentencias (`if`, `if-else`, etc.) con sus respectivos bloques de instrucciones, donde a su vez, se pueden volver a declarar nuevas variables que se conocen como *variables de bloques*, siempre y cuando su nombre no coincida con una variable declarada antes, ya que esto producirá un error.

En el código de la Figura 4.3, se definen dos funciones `func1()` y `func2()` y se representa gráficamente el ámbito de las distintas variables declaradas.

4.3. Paso de información a una función

En ocasiones, una función necesita conocer información externa para poder llevar a cabo su tarea, lo que permite flexibilizar su comportamiento. Veamos un ejemplo: la función `tresSaludos()` es conveniente cuando queremos saludar *exactamente* tres veces. Si deseamos saludar un número distinto de veces, estaríamos obligados a implementar las funciones: `unSaludo()`, `dosSaludos`, `cuatroSaludos()`, etc. Es mucho más práctico implementar

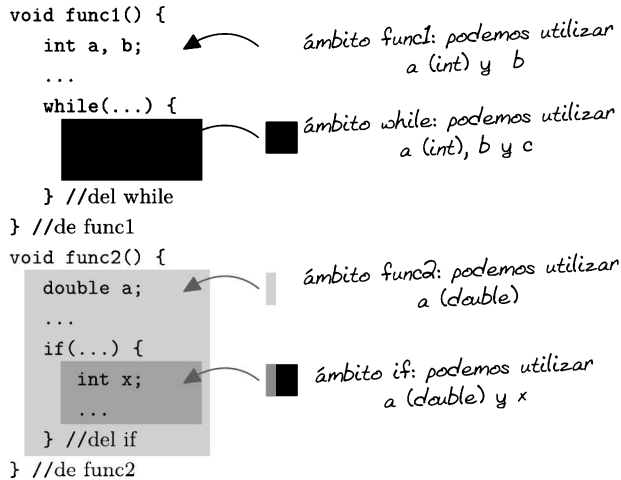


Figura 4.3. Ámbitos de distintas variables

la función `variosSaludos()` a la que se le pasa el número de veces que deseamos saludar. De esta manera, si ejecutamos `variosSaludos(7)` saludará siete veces y si ejecutamos `variosSaludos(2)` lo hará en dos ocasiones.

```

static void variosSaludos(int veces) {
    for(int i = 0; i < veces; i++) {
        System.out.println("Hola.");
    }
}

```

4.3.1. Valores en la llamada

En la llamada a una función se pueden pasar valores que provienen de literales, expresiones o variables. Por ejemplo, a la función `variosSaludos()` se le pasa un entero (número de veces que deseamos saludar),

```

variosSaludos(2); //llamada con un literal
int n = 3;
variosSaludos(2*n); //llamada con una expresión

```

4.3.2. Parámetros de entrada

Una función puede definirse para recibir tantos datos como necesite. Por ejemplo, una función que realiza la suma, puede definirse para que se le pasen dos valores a sumar; y a otra que indica si una fecha es correcta se le pasarán tres valores: el año, el mes y el día de la fecha.

Para una función que calcula y muestra la suma de dos número, la llamada sería,

```
int a = 3;
suma(a, 2); //muestra la suma de a (que vale 3) más 2
```

Cada dato utilizado en la llamada a una función será asignado a un parámetro de entrada, especificado en la definición de la función con la sintáxis,

```
tipo nombreFuncion(tipo1 parametro1, tipo2 parametro2... ) {
    cuerpo de la función
}
```

El primer parámetro de entrada lo hemos llamado **parametro1** y se le puede asignar un valor del tipo *tipo1*, etc. El número de parámetros definidos en la función determina el número de valores que hay que utilizar en cada llamada.

Los parámetros no son más que variables locales de una función que han sido inicializados. ¿De dónde toman los parámetros su valor? De la llamada a la función. De esta forma, en distintas llamadas podemos asignar distintos valores, lo que hace más versátil el uso de las funciones. Solo hay que tener en cuenta que el valor asignado a cada parámetro tiene que corresponder con su tipo. Veamos la función **variosSaludos()**:

```
static void variosSaludos(int veces) {
    int i;
    //disponemos de las variables locales: i y veces
    // el valor de veces se determina en la llamada

    for(i = 0; i < veces; i++) {
        System.out.println("Hola.");
    }
}
```

Si invocamos la función con,

```
variosSaludos(7); //7 se asigna al primer parámetro: veces
```

Suponiendo que hubieramos implementado la función **compruebaFecha()**, a la que se le pasa el día, mes y año de una fecha, y muestra en pantalla si la fecha es correcta o incorrecta, el prototipo de la función es,

```
static void compruebaFecha(int dia, int mes, int año)
```

Un ejemplo de llamada a la función sería,

```
compruebaFecha(a, 4, 2*b+1);
```

El mecanismo de paso de parámetro se representa en la Figura 4.4.

En Java los parámetros toman su valor como una copia del valor de la expresión o variable utilizada en la llamada; este mecanismo de paso de parámetros se denomina *paso de parámetros por valor*, o *por copia*.

En la Figura 4.5 se aprecia cómo se realiza la copia de los valores de las variables utilizadas en la llamada a las variables utilizadas como parámetros.

Veámoslo detenidamente: tras ejecutar la llamada a la función, se salta al cuerpo de la función, copiando el valor del primer parámetro (el valor de **a**, que es 20) a la primera

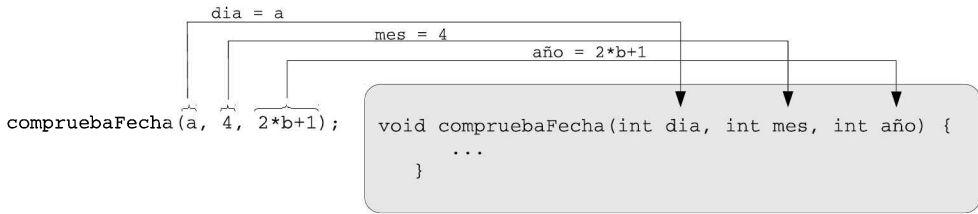


Figura 4.4. Paso de parámetros a una función

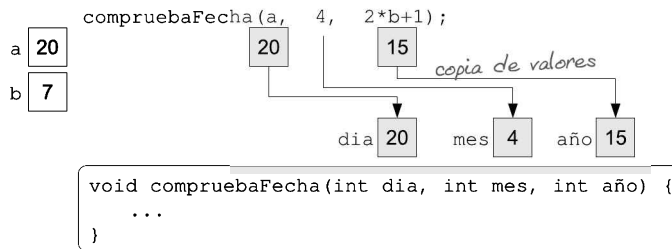


Figura 4.5. Mecanismo de copia de valores a los parámetros

variable utilizada como parámetro (*dia*), el segundo valor utilizado en la llamada (4), al segundo parámetros de entrada (*mes*), etc. El proceso se repite para cada pareja valor-parámetro.

Hay que destacar que, dentro del cuerpo de la función, cualquier cambio en un parámetro de entrada no repercute en la variable o expresión utilizada en la llamada, ya que lo que se modifica es una copia y no el dato original. Veamos un ejemplo,

```
int a = 1, b = 2, c = 3;
compruebaFecha(a, b, c); //llamada
...
//definición de la función
static void compruebaFecha(int dia, int mes, int año) {
    //dia tiene un valor asignado en la llamada (1)
    dia = 23; //modificamos dia, pero la variable a sigue valiendo 1
    ...
}
```

4.4. Valor devuelto por una función

Hemos visto que es posible pasar información hacia la función a través de los parámetros de entrada. También es posible que el paso de información sea en sentido contrario, es decir, desde el cuerpo de la función hacia el código donde se hace la llamada. Con esto conseguimos que la llamada a una función se convierta en un valor cualquiera. Este puede

ser utilizado desde el lugar donde se invoca. Supongamos que disponemos de una función que realiza la suma de dos enteros:

```
int a = suma(2, 3);
int b = suma(7, 1) * 5;
```

En la primera instrucción, `suma(2, 3)` se sustituye por 5, que se asigna a la variable `a`. En la segunda instrucción, `suma(7, 1)` se sustituye por el valor 8, que se multiplica por 5, resultando 40, que se asigna a la variable `b`.

Hasta ahora hemos utilizado siempre `void` como tipo de una función, lo cual indica que la función no devuelve nada, o dicho de otra forma: la llamada a la función no se sustituye por ningún valor. Es posible utilizar cualquier tipo para especificar que la llamada a la función se sustituirá por un valor del tipo indicado. ¿Cómo damos ese valor a la llamada de la función? Para ello disponemos de la instrucción `return` que finaliza la ejecución de la función y devuelve el valor indicado a la llamada. De forma general,

```
tipo nombreFunción(parámetros) {
    ...

    return (valor);
}
```

Donde el tipo de `valor` debe coincidir con *tipo*. La instrucción `return` se utiliza en funciones con un tipo distinto a `void`.

Veamos cómo se implementa la función que realiza la suma de dos números enteros:

```
static int suma(int x, int y) { //cada llamada se sustituye por un int
    int resultado;
    resultado = x + y;

    return (resultado); //sustituye la llamada por el valor especificado
}
```

Debe existir una concordancia entre el tipo de la función y el tipo del valor devuelto. Es importante recordar que la última instrucción de la función debe ser `return`, que fuerza su fin. En caso de existir instrucciones posteriores, no se ejecutarían. Nada impide utilizar varios `return` en una misma función, pero es una práctica desaconsejable, ya que una función debe tener un único punto de entrada y un único punto de salida³, y el uso de varios `return` rompe esta norma. En los ejercicios resueltos utilizamos un único `return` en cada función.

4.5. Sobrecarga de funciones

Java permite tener dos o más funciones con el mismo nombre dentro del mismo programa. Esto es lo que se conoce como *sobrecarga de funciones*. La forma de distinguir entre funciones sobrecargadas es mediante su listas de parámetros, que deben ser distintas, ya sean en número o en tipo.

³Por cuestiones de legibilidad del código y facilidad en el mantenimiento y en la depuración.

La funciones sobrecargadas pueden devolver distintos tipos, aunque estos no sirven para distinguir una función sobrecargada de otra.

Supongamos que queremos diseñar una función para calcular la suma de dos enteros, pero también es útil hacer una suma ponderada, donde cada sumando tiene un peso distinto. Veamos las dos funciones sobrecargadas:

```
//función sobrecarga
static int suma(int a, int b) {
    int suma;
    suma = a + b;
    return(suma);
}

//función sobrecargada
static double suma(int a, double pesoA, int b, double pesoB) {
    double suma;
    suma = a * pesoA / (pesoA + pesoB) + b * pesoB / (pesoA + pesoB);
    return(suma);
}
```

A partir de la definición de las funciones, ambas están disponibles, y cumplen con la única restricción de las funciones sobrecargadas: que se puedan distinguir mediante los parámetros.

Si llamamos a la función `suma` de la forma, `suma(2, 3)`, se ejecutará la primera versión, y devolverá 5. En cambio, si se llama con `suma(2, 0.5, 3, 0.75)`, se ejecutará la segunda versión, y devolverá 3.25.

Es muy común encontrar en la API funciones (métodos) sobrecargadas, ya que permiten agrupar distintas funciones, cuyo uso es similar, bajo el mismo nombre. Por ejemplo, la función que quizás más hemos utilizado hasta ahora, `System.out.println`, se encuentra sobrecargada para poder mostrar en pantalla cualquier tipo de dato.

4.6. Recursividad

Una función puede ser invocada desde cualquier lugar: desde el programa principal, desde otra función, e incluso desde dentro de su propio cuerpo de instrucciones. En este último caso, cuando una función se invoca a sí misma, diremos que es una *función recursiva*.

```
static int funciónRecursiva() {
    ...
    funciónRecursiva(); //llamada recursiva
    ...
}
```

Este es el esquema general de una función recursiva. Si observamos con atención, se plantea un problema: dentro de `funciónRecursiva()` se invoca a `funciónRecursiva()`, donde, a su vez, se volverá a llamar a `funciónRecursiva()`, y así sucesivamente. Esto nos lleva a un ciclo infinito de llamadas a la función. Para evitarlo, hemos de habilitar un mecanismo que detenga, en algún momento, la serie de llamadas recursivas. Una sentencia

if que, utilizando una condición, llamada *caso base*, impida que se continúe con una nueva llamada recursiva. Veamos el esquema general⁴:

```
int funciónRecursiva(datos) {
    int resultado;
    if (caso base) {
        resultado = valorBase;
    } else {
        resultado = funciónRecursiva(nuevos datos); //llamada recursiva
        . . .
    }
    return (resultado);
}
```

Solo cuando la condición del caso base sea *false*, se hará una nueva llamada recursiva. Cuando el caso base sea *true* se romperá la cadena de llamadas. La idea principal de la recursividad es solucionar un problema reduciendo su tamaño. Este proceso continúa hasta que tenga un tamaño tan pequeño, que su solución sea trivial.

Para conseguir cada vez problemas más pequeños, los datos de entrada deben tender hacia el caso base. Conceptualmente *nuevos datos* deben ser *de menor tamaño* que *datos*; así garantizamos que en algún momento los datos utilizados en la función alcanzan el caso base, cortando la serie de llamadas recursivas.

Veamos un ejemplo: supongamos que deseamos limpiar y ordenar una casa; quizás la tarea de limpiar todo a la vez sea demasiado compleja. Podemos utilizar una forma recursiva de solucionar el problema: dividimos la casa en zonas, por ejemplo, por estancias o habitaciones, e iremos limpiando y ordenando cada una de las habitaciones. A su vez, las habitaciones se pueden dividir en zonas, que a su vez se dividen en subzonas. Y así, sucesivamente, hasta que el área donde trabajar sea lo suficientemente manejable.

Veamos otro ejemplo más práctico. Supongamos que deseamos calcular el factorial de un número; sabemos que:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots 2 \times 1$$

por ejemplo:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Podemos calcular el factorial de cualquier número directamente realizando la multiplicación anterior mediante un bucle, pero existe una solución recursiva. Veamos de nuevo la definición de factorial:

$$\begin{aligned} n! &= n \times \underbrace{(n - 1) \times (n - 2) \cdots \times 2 \times 1}_{(n - 1)!} \Rightarrow \\ n! &= n \times (n - 1)! \end{aligned}$$

Para calcular el factorial de un número, estamos utilizando el factorial de un número más pequeño. Para romper la cadena, hemos de buscar un caso base, es decir, un valor

⁴Para la función recursiva `funciónRecursiva()` se ha elegido el tipo `int`. Pero se podría haber utilizado cualquier otro.

para el que calcular el factorial sea algo trivial y no necesitemos volver a utilizar el método recursivo.

El caso base del factorial es:

$$0! = 1$$

Los datos de entrada van siendo menores, y tienden hacia el caso base: para calcular el factorial de n , utilizamos $(n - 1)$ que, a su vez, usará $(n - 2)$, y así, sucesivamente, hasta llegar a 0, cuyo factorial vale 1. Este será el caso base.

Con toda la información de la que disponemos podemos escribir una función que calcule el factorial de un número de forma recursiva:

```
int factorial(int n) {
    int resultado;
    if (n == 0) { //si n es 0
        resultado = 1; //caso base
    } else {
        resultado = n * factorial(n - 1); //llamada recursiva
    }
    return(resultado);
}
```

Hagamos una traza —ejecución paso a paso de las instrucciones de un programa— de la función `factorial(3)`:

```
int factorial(3) {
    int resultado;
    if (3 == 0) { //falso
        . . .
    } else {
        resultado = 3 * factorial(2);
    }
}
```

La ejecución de `factorial(3)` queda a la espera de que se ejecute `factorial(2)`. En este instante existen dos funciones `factorial()` en memoria. Veamos qué ocurre en la llamada a `factorial(2)`:

```
int factorial(2) {
    int resultado;
    if (2 == 0) { //falso
        . . .
    } else {
        resultado = 2 * factorial(1);
    }
}
```

Ahora también `factorial(2)` se queda esperando a la ejecución de `factorial(1)`. En este momento existen en memoria las funciones `factorial(3)` y `factorial(2)` esperando a que termine la ejecución de `factorial(1)`. Veamos cómo se ejecuta la nueva llamada:

```
int factorial(1) {
    int resultado;
    if (1 == 0) { //falso
        . . .
    } else {
        resultado = 1 * factorial(0);
    }
}
```

De forma análoga a las anteriores, se detiene la ejecución de la llamada a la función `factorial(1)` para que comience a ejecutarse una nueva instancia de la función recursiva; es el último caso `factorial(0)`. Hay que destacar que en este momento de la ejecución, están a la espera de que finalicen las respectivas llamadas recursivas varias instancias, o copias, de la función `factorial()`. Veamos cómo se ejecuta `factorial(0)`:

```
int factorial(0) {
    int resultado;
    if (0 == 0) { //cierto
        resultado = 1;
    } else {
        . . .
    }
    return(1);
}
```

La última instancia de la función termina de ejecutarse, devolviendo el valor 1, y permitiendo que la llamada anterior (`factorial(1)`) prosiga su ejecución

```
int factorial(1) {
    . . .
    resultado = 1 * 1; //1
}
return(1);
}
```

De nuevo la función que se ejecuta actualmente, `factorial(1)`, termina, devolviendo el valor 1 y permite que la instancia de la función que esperaba su finalización continúe. Veamos cómo prosigue `factorial(2)` desde el punto en que se quedó esperando:

```
int factorial(2) {
    . . .
    resultado = 2 * 1; //2
}
return(2);
}
```

Termina devolviendo el control para que siga su ejecución `factorial(3)`:

```
int factorial(3) {
    . . .
    resultado = 3 * 2; //6
}
return(6);
}
```

Finaliza la primera instancia de la función que se invocó, devolviendo el control al programa desde donde se llamase, resultando:

```
factorial(3) = 6;
```


Ejercicios de funciones

- 4.1. Diseñar la función `eco()` a la que se le pasa como parámetro un número n , y muestra por pantalla n veces el mensaje:

Eco...

```
import java.util.Scanner;
/*
 * Las implementaciones de funciones irán acompañadas de una función main que sirva de
 * prueba. En este ejercicio, el prototipo de la función es: void eco(int n). */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número: ");
        int n = sc.nextInt();
        System.out.println("--Antes de llamar a la función--");
        eco(n); //invocamos la función
        System.out.println("--Después de llamar a la función--");
    }

    //La función lo único que hace es iterar en un bucle y mostrar un mensaje
    static void eco(int a) { //el parámetro no tiene por qué llamarse como en la llamada
        for (int i = 0; i < a; i++) {
            System.out.println("Eco...");
        }
    }
}
```

- 4.2. Escribir una función a la que se le pasen dos enteros y muestre todos los números comprendidos entre ellos.

```
import java.util.Scanner;
/*
 * Tenemos que saber si los números están en orden creciente (3, 7) o decreciente (7, 3).
 */
public class Main {

    //la función ordena los valores pasados y los utiliza como valores de un bucle for
    static void mostrar(int a, int b) {
        int mayor = a > b ? a : b; //asignamos a mayor el mayor entre a y b
        int menor = a < b ? a : b; //y en menor el más pequeño entre a y b

        for (int i = menor; i <= mayor; i++) { //siempre iremos del menor al mayor
            System.out.println(i);
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca primer número: ");
        int a = sc.nextInt();
        System.out.print("Introduzca segundo número: ");
        int b = sc.nextInt();

        mostrar(a, b);
    }
}
```

- 4.3. Realizar una función que calcule y muestre el área o el volumen de un cilindro, según se especifique. Para distinguir un caso de otro se le pasará un número 1 (para área) o 2 (para el volumen). Además, hemos de pasarle a la función el radio de la base y la altura.

$$\text{área} = 2\pi \cdot \text{radio} \cdot (\text{altura} + \text{radio})$$
$$\text{volumen} = \pi \cdot \text{radio}^2 \cdot \text{altura}$$

```
import java.util.Scanner;
/*
 * Recordemos que el área de un cilindro es 2*PI*radio*(altura+radio) y
 * la fórmula para el volumen es PI*(radio al cuadrado)*altura. */
public class Main {

    static void areaVolumenCilindro(double radio, double altura, int opcion) {
        double volumen, area;

        switch (opcion) {
            case 1:
                volumen = Math.PI * Math.pow(radio, 2) * altura; //aplicamos la fórmula
                System.out.println("El volumen es de: " + volumen);
                break;
            case 2:
                area = 2 * Math.PI * radio * (altura + radio);
                System.out.println("El área es de: " + area);
                break;
            default:
                System.out.println("Indicador del cálculo erróneo");
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca radio: ");
        double radio = sc.nextDouble();
        System.out.print("Introduzca altura: ");
        double alt = sc.nextDouble();
        System.out.print("Qué desea calcular (1 (área)/ 2 (volumen): ");
        int tipoCalculo = sc.nextInt();

        System.out.println();
        areaVolumenCilindro(radio, alt, tipoCalculo);
    }
}
```

- 4.4. Diseñar una función que reciba como parámetros dos números enteros y que devuelva el máximo (el mayor de los dos).

```
import java.util.Scanner;
/*
 * En caso de que ambos números sean iguales, el algoritmo también es válido.
 */
public class Main {

    //compara los parámetros, a y b, y devuelve el mayor de ambos
    static int maximo(int a, int b) {
        int max;

        if (a > b) { // si a es mayor que b
            max = a;
        } else { // si son iguales o b mayor que a
```

```

        max = b;
    } //del if

    return (max); //devuelve el valor de la variable max
}

/* main para probar la función */
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Introduzca un número: ");
    int a = sc.nextInt();
    System.out.print("Introduzca otro número: ");
    int b = sc.nextInt();

    System.out.println("El número mayor es: " + maximo(a, b));
}
}

```

4.5. Repetir el ejercicio anterior con una versión que calcule el máximo de 3 números.

```

/*
 * Vamos a sobrecargar la función para que tenga tres parámetros: maximo(a,b,c). Para im-
 * plementar la función tenemos dos opciones: escribimos el algoritmo desde cero, o nos
 * basamos en la función que ya hemos creado (en el ejercicio anterior). En este caso,
 * vamos a reutilizar el código existente. */
public class Main {

    // función maximo para tres números
    static int maximo(int a, int b, int c) {
        int aux = maximo(a, b); //la variable auxiliar contiene el mayor entre a y b
        return (maximo(aux, c)); //devuelve el mayor entre aux y c
    }

    // función máximo para dos números
    static int maximo(int a, int b) {
        ... //código del ejercicio anterior
    }

    // main para probar la función
    public static void main(String[] args) {
        int max = maximo(2, 9, 7);
        System.out.println("El mayor es: " + max);
    }
}

```

4.6. Crear una función que, mediante un booleano, indique si el carácter que se pasa como parámetro de entrada corresponde con una vocal.

```

/*
 * La función tendrá en cuenta las vocales minúsculas y mayúsculas. No consideraremos las
 * vocales acentuadas (á, é, ...) o con diéresis.
 */
public class Main {

    //programa principal para probar la función
    public static void main(String[] args) {
        System.out.println("La i es una vocal " + esVocal('i'));
        System.out.println("La E es una vocal " + esVocal('E'));
        System.out.println("La f es una vocal " + esVocal('f'));
    }
}

```

```
//comparamos el parámetro de entrada c, con cada posible valor de una vocal
static boolean esVocal(char c) {
    boolean resultado; //true si es una c es una vocal y false en caso contrario

    if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ||
        c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') {
        resultado = true;
    } else {
        resultado = false;
    }
    return resultado;
}
}
```

4.7. Diseñar una función que nos diga si un número es primo.

```
/*
 * La función esPrimo() indica, mediante un booleano, si el número pasado como parámetro
 * es primo. Un número n es primo si no es divisible por ningún número entre 2 y n-1
 * Recordemos que un número primo es solo divisible por el mismo y por 1. */
public class Main {

    static boolean esPrimo(int num) {
        boolean primo = true; // suponemos que el número es primo
        int i = 2; //primer número por el que veremos si es divisible

        if (num < 2) { // el primer primo es 2
            primo = false;
        }

        while (i < num && primo == true) {
            if (num % i == 0) { // si num es divisible por i
                primo = false; // entonces no es un número primo
            }
            i++;
        }

        // este algoritmo puede mejorar sabiendo que si un número no es divisible por
        // ningún entero comprendido entre 2 y su raíz cuadrada, entonces ya no será
        //divisible por ningún otro número y será primo. Quedaría:

        // while (i <= (int) Math.sqrt(num) && primo == true) {
        //     ...
        // }
        //lo cuál ahorra muchas vueltas para números primos grandes
        return (primo);
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 15; i++) {
            if (esPrimo(i)) {
                System.out.println(i + " es primo");
            } else {
                System.out.println(i + " es compuesto");
            }
        }
    }
}
```

4.8. Escribir una función a la que se le pase un número entero y devuelva el número de divisores primos que tiene.

```

/*
 * Para calcular los divisores de un número, solo tendremos en cuenta los números primos
 * comprendidos entre 1 y el número que nos interese.
 * Un ejemplo: los divisores de 24 son: 1, 2 y 3.
 * Aunque 4 y 6 también dividen a 24, no se consideran al no ser primos.
 */

public class Main {

    // la función esPrimo() está implementada en el ejercicio anterior.
    static boolean esPrimo(int num) {
        ... //en el ejercicio anterior
    }

    static int numDivisoresPrimos(int num) {
        int cont; // contador de divisores
        cont = 1; // siempre habrá un divisor seguro, el 1.
        for (int i = 2; i <= num; i++) {
            if (esPrimo(i) && num % i == 0) { // si i es primo y divide a num
                cont++; // incrementamos el número de divisores
            }
        }
        return (cont);
    }

    //programa principal para probar la función
    public static void main(String[] args) {
        System.out.println("Divisores de 24: " + numDivisoresPrimos(24));
    }
}

```

4.9. Implementar la función `divisoresPrimos()` que muestra, por consola, todos los divisores primos del número que se le pasa como parámetro.

```

import java.util.Scanner;

/*
 * La función comprobará todos los números comprendidos entre 1 y el número que se pasa
 * como parámetro (num). Para cada uno de ellos, verificaremos si es primo y si divide
 * a num, en cuyo caso, significa que es un divisor primo de num.
 */

public class Main {

    //ejemplo de uso
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca número: ");
        int num = sc.nextInt();
        divisoresPrimos(num); //mostramos los divisores primos de num
    }

    static boolean esPrimo(int num) {
        ... //en el ejercicio 7
    }

    //Comprobaremos todos los números en el rango comprendido entre el 1 y num (el número
    //1 siempre será divisor primo de cualquier número, podríamos obviarlo).
    static void divisoresPrimos(int num) {
        System.out.println("Divisores primos de " + num); //mostramos una cabecera

        //recorremos los números del rango 1..num
        for (int i = 1; i <= num; i++) {

```

```
        if (esPrimo(i) && num % i == 0) { // si i es primo y divide a num
            // i es un divisor primo de num
            System.out.println("El número " + i); //mostramos
        }
    }
}
```

4.10. Escribir una función que decida si dos números enteros positivos son amigos. Dos números son amigos si la suma de sus divisores propios (distintos de ellos mismos) son iguales.

```
import java.util.Scanner;
/*
 * Antes de ver qué son los números amigos, una definición:
 * divisores propios de un número son todos los divisores, primos y no primos,
 * de dicho número excepto él mismo.
 * Dos números a y b son amigos cuando la suma de los divisores propios de a
 * es b y viceversa. Ejemplos conocidos de pares de números amigos son:
 * (220, 284), (1184, 1210), (6232, 6368). Un número amigo de sí mismo se dice
 * que es perfecto, como 6, 28, 496 y 8128. */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca a: ");
        int a = sc.nextInt();
        System.out.print("Introduzca b: ");
        int b = sc.nextInt();

        if (amigos(a, b)) {
            System.out.println(a + " y " + b + " son amigos.");
        } else {
            System.out.println("No son amigos. La próxima vez prueba con 220 y 284.");
        }
    }

    //calcula la suma de sus divisores y comprueba
    static boolean amigos(int a, int b) {
        boolean amigos;

        if (a == sumaDivisoresPropios(b) && b == sumaDivisoresPropios(a)) {
            amigos = true; //son amigos
        } else {
            amigos = false; //no son amigos
        }
        return (amigos);
    }

    // sumaDivisores devuelve la suma de los divisores propios, divisores en el
    // rango 1..num-1
    static int sumaDivisoresPropios(int num) {
        int suma = 0;

        for (int i = 1; i < num; i++) { // al ser i<num no usamos el propio num
            if (num % i == 0) { // si i divide a num
                suma += i; // acumulamos i
            }
        }
        return (suma);
    }
}
```

4.11. Diseñar una función que calcule a^n , donde a es real y n entero no negativo. Realizar una versión iterativa y otra recursiva.

Solución a)

```
import java.util.Locale;
import java.util.Scanner;
/*
 * Como 0 elevado a 0 no está definido y admitimos el exponente 0,
 * excluirémos dicho valor para la base a. */
public class Main {

    static double aElevadoN(double a, int n) {
        double res = 1; // el resultado se inicializa a 1, ya que multiplicamos

        if (n == 0) { // por definición cualquier número elevado a 0 es 1
            res = 1;
        } else {
            for (int i = 1; i <= n; i++) {
                res = res * a; //multiplicamos
            }
        }

        return (res);
    }

    //programa principal para probar la función
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US); //para permitir puntos (.) en los decimales

        System.out.print("Introduzca base (real): ");
        double base = sc.nextDouble();
        System.out.print("Introduzca exponente (entero): ");
        int exp = sc.nextInt();

        double res = aElevadoN(base, exp);
        System.out.println(base + " elevado a " + exp + " = " + res);
    }
}
```

Solución b)

```
import java.util.Locale;
import java.util.Scanner;
/*
 * La funciones recursivas tienen casi todas la misma estructura:
 * - caso base, que permite salir de la recursividad
 * - llamada recursiva
 * En nuestro caso: el caso base es aElevadoN(x, 0) = 1
 * y la llamada recursiva: aElevadoN(a, n) = aElevadoN(a, n-1) * a */
public class Main {
    //programa principal para probar la función aElevadoN(), de forma recursiva.
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US);

        System.out.print("Introduzca base (real): ");
        double base = sc.nextDouble();
        System.out.print("Introduzca el exponente: ");
        int exp = sc.nextInt();
        System.out.println("El resultado es: " + aElevadoN(base, exp));
    }
}
```

```
static double aElevadoN(double a, int n) {
    double res;

    if (n == 0) { // caso base
        res = 1; //a elevado a 0 es 1
    } else {
        res = a * aElevadoN(a, n - 1); //llamada recursiva
    }

    return (res);
}
}
```

4.12. Diseñar la función `calculadora()`, a la que se le pasan dos números enteros (operandos) y qué operación se desea realizar con ellos. Las operaciones disponibles son: sumar, restar, multiplicar o dividir. Estas se especifican mediante un número: 1 para la suma, 2 para la resta, 3 para la multiplicación y 4 para la división. La función devolverá el resultado de la operación mediante un número real.

```
public class Main {

    //programa para probar
    public static void main(String[] args) {
        for (int operacion = 1; operacion <= 4; operacion++) { //todas las operaciones
            double resultado = calculadora(3, 4, operacion); //operamos con 3 y 4
            System.out.println(resultado);
        }

        //Realiza la operación indicada:
        // suma(1), resta(2), multiplicación(3) o división(4)
        static double calculadora(int a, int b, int operacion) {
            double result; // resultado de la operación

            switch (operacion) {
                case 1: //suma
                    result = a + b;
                    break;
                case 2: //resta
                    result = a - b;
                    break;
                case 3: //multiplicación
                    result = a * b;
                    break;
                case 4: //división
                    result = (double)a / b; //el cast fuerza que la división sea real
                    //falta comprobar que no es una división por 0
                    break;
                default:
                    result = 0; //si la operación no tiene sentido devolveremos 0
                    System.out.println("Operación no válida");
                    break;
            }
            return (result);
        }
    }
}
```

4.13. Calcular el factorial de n recursivamente. Recordar que por definición el factorial de 0 ($0!$) es 1.


```
import java.util.Scanner;
/*
 * Para calcular n! (factorial de n), una forma es n!= n*(n-1)*(n-2)...1, y otra consiste
 * en n! = n*(n-1)!. Por ejemplo, 3! = 3*2*1, pero también 3! = 3*2!
 * El caso base es 0! = 1 */

public class Main {
    //programa principal que pide un número y calcula su factorial
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduzca un número: ");
        int num = sc.nextInt();
        System.out.println(num + "! es igual a " + factorial(num));
    }
    //función recursiva para calcular el factorial
    static int factorial(int num) {
        int res;

        if (num == 0) { // caso base: 0! (factorial de 0) es 1 (por definición)
            res = 1;
        } else {
            res = num * factorial(num - 1); //llamada recursiva
        }

        return (res);
    }
}
```

4.14. Diseñar una función que calcule el n -ésimo término de la serie de Fibonacci. En esta serie el n -ésimo valor se calcula sumando los dos valores anteriores. Es decir:

$$\begin{aligned} fibonacci(n) &= fibonacci(n-1) + fibonacci(n-2) \\ fibonacci(0) &= 1 \\ fibonacci(1) &= 1 \end{aligned}$$

```
import java.util.Scanner;

/*
 * En la serie de Fibonacci tendremos:
 * - caso general: fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
 * - existen dos casos base:
 *     fibonacci(0) = 1
 *     fibonacci(1) = 1
 */

public class Main {

    //programa principal para probar la función fibo()
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Vamos a calcular fibonacci(n)");
        System.out.print("Introduzca n (se recomienda n<40): ");
        int num = sc.nextInt();

        int resultado = fibo(num); // si n es muy grande esto puede tardar bastante
        System.out.println("\nfibonacci(" + num + ") = " + resultado);
    }

    //función recursiva
    static int fibo(int num) {
        int res;
    }
```

```
    if (num == 0) { // primer caso base
        res = 1;
    } else {
        if (num == 1) { // segundo caso base
            res = 1;
        } else {
            res = fibo(num - 1) + fibo(num - 2); // caso general recursivo
        }
    }

    return (res);
}
}
```

Ejercicios propuestos

- 4.1. Diseñar una función que calcule la superficie y el volumen de una esfera.

$$Superficie = 4\pi \times radio^2$$

$$Volumen = \frac{4}{3}\pi \times radio^3$$

- 4.2. Crear la función `muestraPares(int n)` que muestre por consola los primeros `n` números pares.

- 4.3. Implementar la función,

```
static double distancia(double x1, double y1, double x2, double y2)
```

que calcula y devuelve la distancia euclídea que separa los puntos $(x1, y1)$ y $(x2, y2)$. La fórmula para calcular esta distancia es,

$$distancia = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

- 4.4. Escribir una función a la que se pasen como parámetros de entrada una cantidad de días, horas y minutos. La función calculará y devolverá el número de segundos que existen en los datos de entrada.
- 4.5. Diseñar una función a la que se le pasan las horas y minutos de dos instantes de tiempo, con el prototipo

```
static int diferenciaMin(int hora1, int minuto1, int hora2, int minuto2)
```

La función devolverá la cantidad de minutos que existen de diferencia entre los dos instantes utilizados.

Capítulo 5

Tablas

Responda a una sencilla pregunta: ¿cuántos valores puede almacenar simultáneamente una variable? Según vimos en el primer capítulo, la respuesta es obvia: un solo valor en cada instante. Analicemos el siguiente código:

```
edad = 6;  
edad = 23;
```

La variable `edad` inicialmente almacena el valor 6 y a continuación 23. Cada nueva asignación modifica `edad`, pero, independientemente del número de asignaciones, la variable contiene únicamente un valor en cada momento y se conoce como una variable *escalar*.

¿Existe una forma de almacenar más de un valor simultáneamente en una variable? La respuesta es sí, mediante el uso de tablas¹.

5.1. Variables escalares *vs* tablas

Una tabla es una variable que permite guardar más de un valor simultáneamente. Podemos ver una tabla como una «supervariable» que engloba a otras variables, llamadas *elementos* o *componentes* referidas con un mismo nombre, con la condición de que todas sean del mismo tipo. En la Figura 5.1 se representa la tabla `edad` que guarda los valores —años— de los asistentes a una fiesta: 85, 3, 19, 23 y 7.

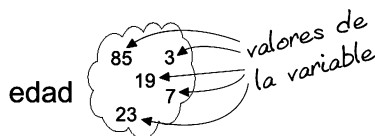


Figura 5.1. Representación de una tabla con varios valores

Siempre que necesitemos manejar varios datos del mismo tipo simultáneamente, en general, se recomienda utilizar una tabla en lugar de varias variables escalares. Es mucho

¹En la bibliografía es común encontrar autores que denominan a las tablas *arrays* o vectores.

más cómodo trabajar con una tabla que almacena, por ejemplo, 100 valores, que hacerlo con 100 variables escalares. Cuando desconocemos cuántos datos son necesarios gestionar, no existe otra alternativa que utilizar tablas, ya que a la hora de crearlas, su tamaño se puede elegir dinámicamente.

5.2. Índices

El problema es cómo distinguir entre cada uno de los elementos o componentes que constituyen una tabla. En nuestro caso, ¿cómo podemos utilizar el valor 85 o el valor 19 que se encuentran almacenados en la tabla `edad`? La solución consiste en asignar un número de orden, llamado *índice*, a cada elemento, para así poder diferenciarlos. Al primer elemento se le asigna el índice 0, al segundo el índice 1 y así sucesivamente. Al último elemento le corresponde como índice el número total de elementos menos uno (Figura 5.2)².

La forma de utilizar un elemento concreto de una tabla es a través del nombre de la variable junto al número —índice— que distingue ese elemento entre corchetes (`[ ]`). Por ejemplo, para utilizar el cuarto elemento de `edad` —elemento con índice 3—, escribiremos: `edad[3]`, que contiene un valor de 23.

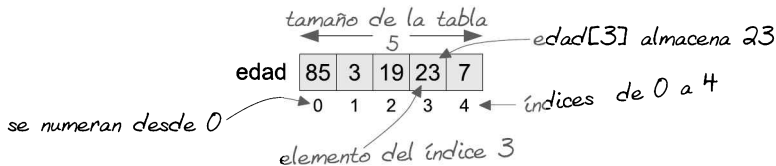


Figura 5.2. Una tabla de cinco elementos enteros

Veamos un ejemplo de cómo mostrar y asignar un elemento:

```
System.out.println(edad[0]); //muestra el primer elemento: 85
edad[3] = 8; //asigna un nuevo valor al cuarto elemento
```

5.2.1. Índices fuera de rango

La variable `edad` es una tabla de 5 enteros. Ni que decir tiene que podemos utilizar cada uno de los 5 elementos que la componen, de la forma: `edad[0]`, `edad[1]`..., `edad[4]`.

¿Qué ocurrirá si utilizamos un índice que se encuentra fuera del rango de 0 a 4? Es decir, qué efecto produce escribir,

```
edad[-2]
edad[7]
```

Obtendremos un error en tiempo de ejecución que provoca que el programa termine de forma inesperada, ya que se detecta que los elementos asociados a los índices utilizados

²Esto es consecuencia de comenzar a numerar en 0.

no existen. Cuando se comienza a programar, la mayoría de errores al trabajar con tablas provienen de utilizar índices fuera de rango. Se recomienda prestar especial atención a esto.

5.3. Construcción de tablas

En el momento de crear una tabla tendremos que decidir qué tipo de datos vamos a almacenar y cuántos elementos necesitamos, declarar variables para tablas y crear la propia tabla.

5.3.1. Tamaño y tipo

Una tabla se define mediante dos características fundamentales: su tamaño y su tipo. El *tamaño* o *longitud* es el número de elementos que componen una tabla. Y el *tipo* de una tabla es el de los datos que almacena en todos y cada uno de sus elementos. En la Figura 5.3 podemos ver dos tablas: la primera compuesta por tres elementos de tipo `double` y la segunda por seis elementos de tipo entero.

1.2	0.3	9.0	<i>tamaño: 3</i> <i>tipo: double</i>	
0	1	2		

5	-1	0	2	0	-7	<i>tamaño: 6</i> <i>tipo: int</i>	
0	1	2	3	4	5		

Figura 5.3. Tablas con distintos tamaños y tipos

5.3.2. Variables de tabla

El procedimiento para crear tablas se compone de dos etapas: primero declarar la variable, utilizando corchetes (`[]`), símbolo que diferencia entre una variable escalar y una que es tabla. Es posible utilizar dos sintaxis equivalentes:

```
tipo nombreVariable[];
tipo [] nombreVariable;
```

donde *tipo* representa cualquier tipo primitivo³. Veamos cómo declarar la variable `edad` como una tabla de tipo `int`:

```
int edad[];
```

o mediante la forma (ambas son equivalentes),

```
int[] edad;
```

En este punto la variable está declarada, pero no hemos construido ninguna tabla.

³En el Capítulo 7 veremos cómo utilizar cualquier clase como tipo de una tabla.

5.3.3. Operador `new`

Una vez que hemos declarado una variable es posible crear una tabla con el tamaño adecuado. Así dispondremos de los elementos para almacenar valores. La sintaxis es:

```
nombreVariable = new tipo [tamaño];
```

Veamos cómo crear la tabla `edad`, de tipo `int` y con un tamaño de 5 elementos:

```
edad = new int[5];
```

El operador `new` construye una tabla donde todos los elementos se inicializan a 0, para tipos numéricos, o `false` si la tabla es booleana⁴. Es posible declarar la variable y crear la tabla en una única sentencia,

```
int edad[] = new int[5];
```

5.3.4. Construcción mediante asignación

Existe una alternativa para crear una tabla sin necesidad de utilizar `new`. En la misma declaración se asignan valores a los elementos de la tabla, que se crea con el tamaño necesario para albergar todos los valores asignados. Veamos un ejemplo:

```
int datos[] = {2, -3, 0, 7}; //tabla de tamaño 4
```

Esta sentencia declara la variable `datos` y crea una tabla de cuatro enteros, donde cada elemento tiene asignado el valor correspondiente. Equivalente a:

```
int datos[]; //declaramos la variable  
datos = new int[4]; //creamos la tabla
```

```
datos[0] = 2; //asignamos valores  
datos[1] = -3;  
datos[2] = 0;  
datos[3] = 7;
```

La creación de una tabla mediante la asignación de valores solo puede realizarse en la misma instrucción donde se declara. No se puede escribir,

```
int datos[];  
datos = {2, -3, 0, 7} //no permitido, solo en la declaración
```

5.4. Referencias

La siguiente intrucción,

```
edad = new int[10];
```

construye una tabla de 10 elementos de tipo `int` y la asigna a la variable `edad`. Estudiemos cómo funciona el operador `new`:

⁴Los elementos de las tablas de otros tipos se inicializan a `null` (véase apartado 5.4.2).

- En primer lugar calcula el tamaño físico de la tabla, es decir, el número de bytes que ocupará la tabla en la memoria. Este cálculo es sencillo y se obtiene de multiplicar el tamaño de la tabla por el tamaño de su tipo⁵. Como ejemplo vamos a calcular el tamaño de una tabla de diez enteros,

tamaño físico = tam. tabla × tam. tipo = 10 × tam. entero = 10 × 4 bytes = 40 bytes

es decir, una tabla de 10 enteros ocupa 40 bytes en la memoria del ordenador.

- Conociendo el tamaño físico de la tabla, busca en la memoria un hueco libre —memoria no utilizada— con tamaño suficiente para albergar todos los elementos de la tabla consecutivamente (véase la Figura 5.4).

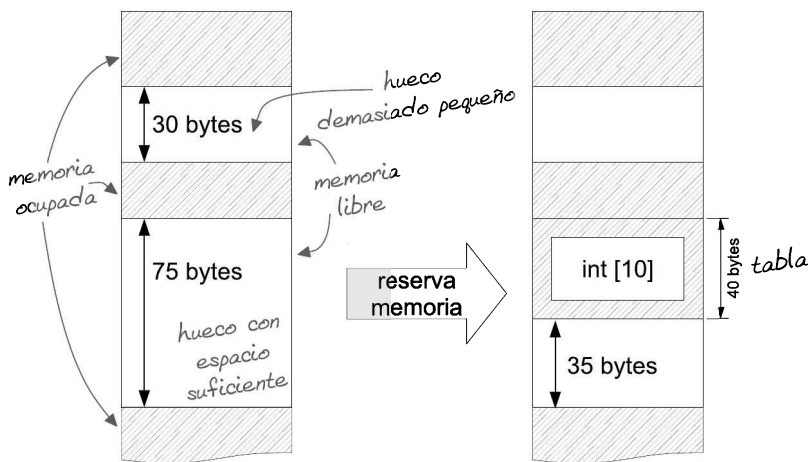


Figura 5.4. Reserva de memoria. Representación de la memoria antes y después de construir una tabla

- Reserva la memoria necesaria para almacenar la tabla y la marca como memoria ocupada, siendo este el sitio donde se almacenarán los elementos de la tabla.
- Por último, recorre todos los elementos de la tabla inicializándolos de la siguiente manera: 0 si es una tabla numérica o **false** si la tabla es booleana.

En este punto hemos creado la tabla. El siguiente paso es asignarla a la variable correspondiente; para ello Java dispone de un mecanismo para indicar dónde está la tabla en la memoria. Cada posición de la memoria de un ordenador tiene una dirección única que la identifica. Así la primera posición de memoria tiene la dirección 000, la siguiente la dirección 001 y así sucesivamente⁶. En Java a cada dirección de memoria se le denomina *referencia*.

⁵ Véase Apartado 1.8.1.

⁶ Realmente es algo más complicado, pero con esta simplificación es suficiente para entender el concepto.

La forma de que una variable sepa donde está la tabla en la memoria es asignándole la referencia de la primera posición que ocupa (una tabla puede ocupar varias posiciones consecutivas). Las variables de tablas, lo que almacenan realmente es una referencia; por ese motivo se las conoce también como *variables de referencia*. La Figura 5.5 muestra la zona de la memoria reservada para la nueva tabla, que suponemos se encuentra en la referencia 0723.

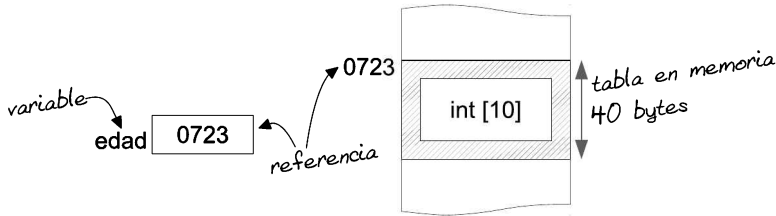


Figura 5.5. Asignación de una referencia a una variable

Si ejecutamos las siguientes líneas,

```
int t[] = new int[10];  
System.out.println(t);
```

no muestra el contenido de cada elemento de la tabla. Lo que se obtiene es la referencia que guarda la variable `t`, que suele tener una forma similar: `I0659e0bfd`

Las referencias se modifican en cada ejecución, dependiendo de la ocupación de la memoria. En las representaciones gráficas es mucho más intuitivo sustituir los valores de la referencia por una flecha que tiene el mismo significado: «la variable está referenciando esta tabla». Esta representación se muestra en la Figura 5.6, donde también hemos cambiado el trozo de memoria por casillas que representan los elementos de la tabla.



Figura 5.6. Representación de una tabla referenciada por una variable

Ahora que entendemos cómo funcionan las referencias podemos plantear tres posibles escenarios:

- Variable sin tabla. Podemos declarar una variable de tipo tabla, pero mientras no construyamos una tabla, la variable, por sí sola, no sirve de nada, ya que no disponemos de ningún elemento para almacenar datos.

```
int a[]; //variable de tabla
```

En definitiva: sin elementos, obtenidos con `new`, la variable `a` no es operativa.

- Tabla sin variable. Igual ocurre si construimos una tabla pero no disponemos de variable donde asignar su referencia,


```
new int[10];
```

La sentencia anterior construye una tabla con diez elementos enteros, pero al no guardar su referencia en ninguna variable, no tenemos forma de utilizarla.

- Lo útil es combinar la declaración de variables con la construcción de tablas,

```
int b[], c[]; //variables
```

```
b = new int[4]; //tabla de cuatro enteros accesible desde a
```

```
c = new int[5]; //tabla de cinco enteros accesible desde b
```

Cuando a una variable de tabla se le asigna una tabla, se dice que la variable *referencia* a la tabla. La Figura 5.7 muestra todos los casos posibles de referencias.

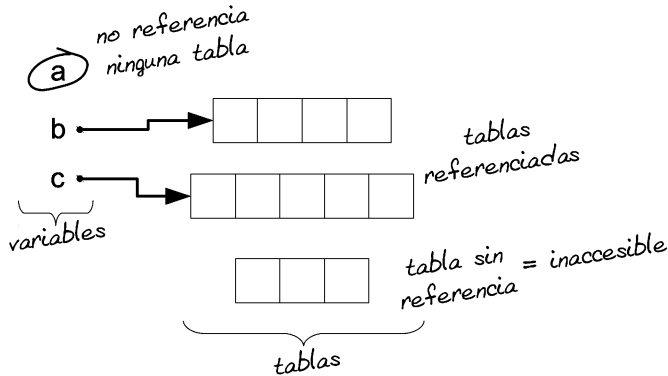


Figura 5.7. Variables y tablas

Las variables pueden verse como medios para acceder a los elementos de las tablas a las que referencian. Es posible acceder a una misma tabla mediante más de una variable, o dicho de otra forma, la misma tabla puede estar referenciada por más de una variable. La Figura 5.8 representa el resultado del siguiente código,

```
int d[], e[]; //variables
```

```
d = new int[6]; //construimos una tabla referenciada por d
```

```
e = d; //ahora la variable e referencia la misma tabla que d
```

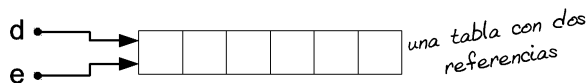


Figura 5.8. Tabla multireferenciada

Ahora podemos acceder a los mismos datos utilizando la variable `d` o la variable `e`: utilizar `d[2]` es equivalente a utilizar `e[2]`, ya que ambas variables referencian la misma tabla. La única condición para que una variable pueda referenciar a una tabla es que el tipo de la variable y el de la tabla coincidan. El siguiente código es erróneo debido a que los tipos no coinciden,

```
boolean t1[]; //variable para tablas booleanas
int t2[]; //variable para tablas enteras

t1 = new boolean[10]; //construye y asigna una tabla de 10 booleanos
t2 = t1; //ERROR: tipos incompatibles
       //t2 puede referenciar tablas enteras, pero no booleanas
```

5.4.1. Recolector de basura

Hay un detalle en el que es interesante profundizar: ¿qué ocurre cuando tenemos una tabla que no está referenciada por ninguna variable? Lo primero y más obvio es que dicha tabla es inútil; no hay forma de acceder a sus elementos. Pero existe un segundo problema: la tabla está ocupando espacio en la memoria. Quizás el tamaño de unas pocas tablas inútiles en memoria no sea significativo, pero una aplicación puede dejar, durante su ejecución, grandes cantidades de memoria ocupada inaccesible. Esto puede ocurrir por un mal diseño o de forma malintencionada. Java soluciona el problema mediante un mecanismo muy ingenioso: periódicamente se inicia un proceso llamado *recolector de basura*⁷ que comprueba todas las tablas construidas. Si encuentra alguna inaccesible —sin variables que la referencien— la destruye, dejando libre el espacio que estaba ocupando en memoria.

5.4.2. null

Hemos visto las formas de asignar a una variable la referencia de una tabla, directamente con `new` o a través de otra variable. Pero existe la forma de hacer justo lo contrario, es decir, hacer que una variable que referencia a una tabla no referencie nada. Para ello disponemos del literal `null` que significa «vacío».

Veamos un ejemplo de como dejar sin referencia a una tabla,

```
int t1[], t2[]; //variables de tipo tabla entera

t1 = new int[100]; //t1 referencia una tabla de 100 elementos
t2 = t1; //ahora t2 también referencia la misma tabla

t1 = null; //anulamos t1: no referencia a ninguna tabla
          //la tabla con 100 elementos sigue siendo accesible desde t2

t2 = null; //anulamos t2: tampoco hace referencia a nada
          //la tabla es inaccesible: el recolector de basura se encargará de ella
```

⁷En la bibliografía es habitual encontrarlo con su nombre original *Garbage Collection*.

5.5. Clasificación de tablas

Dependiendo de la forma de utilizar los datos en las tablas, podremos clasificarlas atendiendo a distintos criterios: si utiliza todos los elementos para albergar datos, si los datos están ordenados, etc.

5.5.1. Según el número de elementos con datos

Por ejemplo, en una tabla de 10 elementos podemos almacenar 10 datos o valores como máximo, pero también tenemos la posibilidad de almacenar un número menor de datos.

Tablas completas

Son aquellas donde se utilizan siempre todos los elementos para almacenar valores. Por ejemplo, la siguiente tabla tiene 5 elementos, donde se almacena 5 datos.

datos	6	1	2	6	4

Tablas incompletas o con huecos

El número de datos almacenados es menor que el tamaño de la tabla. Por ejemplo, la tabla **datos** tiene un tamaño de 5, pero solo almacena 3 datos.

datos		0		7	4

La dificultad está en cómo distinguir un elemento ocupado por un dato válido, de uno que no lo está. Cuando se crea una tabla, Java inicializa automáticamente todos sus elementos, así que la tabla anterior realmente contiene:

datos	0	0	0	7	4
	0	1	2	3	4

Tanto `datos[0]`, `datos[1]` como `datos[2]` almacenan un 0 pero, ¿cómo distinguir un dato válido de un valor basura⁸? Para ello utilizaremos una variable entera a modo de indicador. La idea es tener los datos agrupados al principio, o al final, en lugar de desperdigados por la tabla, con lo cual, las celdas vacías quedan al final, o al principio. En lugar de:

datos		0		7	4

tendríamos:

datos	0	7	4		
	0	1	2	3	4

⁸Cualquier valor que se encuentre almacenado en la tabla pero que no sea útil. Los elementos de una tabla siempre almacenan algún valor.

Aquí es sencillo distinguir con un simple número entero la parte de datos de la parte sin ellos. El indicador puede hacer referencia (véase Figura 5.9) al:

- Número de elementos que contienen datos válidos. En el ejemplo anterior: 3.
- Índice del último elemento que contiene un dato válido. En el ejemplo anterior: 2.
- Índice del primer hueco libre. En el ejemplo anterior: 3.

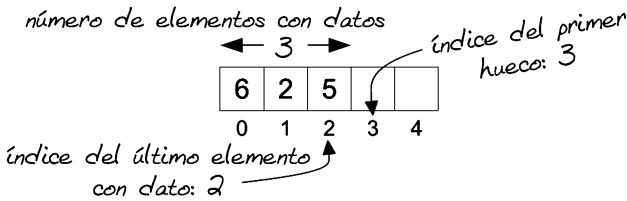


Figura 5.9. Representación del significado de distintos indicadores

Cada programador define el indicador según sus gustos. Para nuestros ejercicios utilizaremos habitualmente el número de datos válidos en la tabla.

El indicador se incrementa cada vez que se inserta un nuevo valor o se decrementa cuando eliminamos un elemento de la tabla. En este último caso, el elemento nunca se llega a eliminar; lo que realmente ocurre es que marcamos el valor como basura.

5.5.2. Según el orden

Los datos de una tabla, independientemente de si se utiliza completa o con huecos, se pueden almacenar ordenados o sin mantener orden alguno.

Tablas ordenadas

En este caso, tanto si la tabla está completa como si tiene huecos, los datos están ordenados siguiendo algún criterio. Un ejemplo de una tabla completa ordenada en sentido creciente es:

precios	12.3	18.0	34.65	80.19	102.0

En el momento de insertar o eliminar un elemento tendremos que realizar la operación teniendo cuidado de que la tabla continúe ordenada.

Tablas desordenadas

El orden no es un criterio para el almacenamiento de los datos. Las operaciones como insertar o eliminar un dato son más sencillas, ya que no tenemos que preocuparnos del orden de los elementos de la tabla después de realizar la operación.

5.6. Tablas como parámetros de funciones

Recordemos cuál es el mecanismo de paso de parámetros al invocar una función: el valor de la variable que se utiliza en la llamada se copia al parámetro de la función (que es una variable local en la función). Veamos un ejemplo,

```
func(a); //llamada a la función
...

void func(int b) { //declaración de la función
    ...
}
```

El valor de la variable **a** se copia en el parámetro **b**. En este caso, al ser variables escalares, si en el cuerpo de la función se modifica la variable **b**, se está modificando una copia, no la variable **a**.

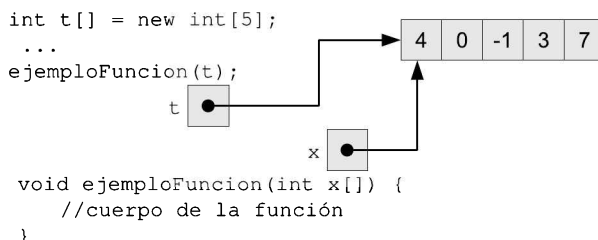


Figura 5.10. Paso de parámetro de una tabla

Cuando utilizamos tablas como parámetros el mecanismo es el mismo, es decir, el valor de la variable utilizada en la llamada se copia al parámetro de la función. Pero en este caso, lo que se copia es la referencia de una tabla, con lo que se consigue que la tabla esté referenciada tanto por la variable utilizada en la llamada como por el parámetro. La modificación de un componente de la tabla dentro de la función también es visible desde la variable externa a la función.

En el ejemplo de la Figura 5.10, si dentro del cuerpo de la función **ejemploFuncion()** ejecutamos,

```
x[2] = 10;
```

estamos cambiando también **t[2]**, ya que, tanto **x** como **t** referencian a la misma tabla.

5.7. Clase Arrays

La API de Java, como ya sabemos, proporciona herramientas que facilitan el trabajo de un programador. Hasta ahora, hemos utilizado las clases **Scanner** y **Math** entre otras. Con respecto a las tablas disponemos de la clase **Arrays**, que tiene una serie de métodos estáticos

para manipular tablas y realizar operaciones con ellas. Se ubica en el paquete `java.util`, y para poder utilizarla necesitamos importarla de la forma,

```
import java.util.Arrays;
```

No existe ningún motivo por el que no podamos implementar nuestras propias operaciones para tablas, pero **Arrays** nos proporciona la seguridad de un código probado, sin errores y la comodidad que supone ahorrarnos el tiempo de escribir nuestra implementación. Durante este capítulo solo implementaremos aquello que no se encuentre disponible en un método de **Arrays**.

5.8. Operaciones con tablas

Las posibilidades al trabajar con tablas son prácticamente infinitas, pero casi todo lo que necesitemos puede sintetizarse a partir de las operaciones que veremos a lo largo de este apartado.

5.8.1. Obtención del tamaño

Java proporciona un mecanismo para conocer el tamaño de una tabla,

```
nombreVariable.length
```

que se evalúa como un entero con el número de elementos reservados en memoria, utilizados o no, de la tabla. Por ejemplo, el código:

```
int notas[] = new int [10];  
System.out.println("Tamaño de la tabla notas: " + notas.length);
```

muestra por pantalla:

```
Tamaño de la tabla notas: 10
```

Averiguar el tamaño de una tabla puede ser necesario cuando manipulamos, en el cuerpo de una función, una tabla pasada como parámetro. En este contexto desconocemos con qué tamaño se creó.

5.8.2. Recorrido

Muchas operaciones con tablas implican realizar un recorrido, que consiste en visitar sus elementos para procesarlos. Por procesar se entiende cualquier operación que realicemos con un elemento, como por ejemplo, asignarle un valor, mostrarlo por consola o realizar algún tipo de cálculo con él. El recorrido puede ser total, cuando se recorren todos sus elementos, o parcial, cuando solo visitamos un subconjunto de los elementos de una tabla. El código tipo para recorrer una tabla es,

```
for (int i = desde; i <= hasta; i++) {  
    //procesado t[i]  
    ...  
}
```

siendo **desde** el índice del primer elemento a visitar y **hasta** el índice del último elemento visitado. Por ejemplo, si deseamos incrementar un 10% todos los elementos de la tabla `sueldos`,

```
for (int i = 0; i <= sueldos.length - 1; i++) { //recorremos toda la tabla
    sueldos[i] = sueldos[i] + 0.1 * sueldos[i]; //procesamos
}
```

otra forma alternativa, más usual, para el bucle `for` es,

```
for (i = 0; i < sueldos.length; i++) {
```

donde utilizamos `<` en lugar de `<=` y nos ahorramos restarle 1 al tamaño de la tabla.

La instrucción `for` tiene una sintaxis alternativa, conocida como *for-each*, que permite recorrer los elementos de una tabla,

```
for (declaración variable: tabla) {
    ...
}
```

donde se declara una variable, que tiene que ser del mismo tipo que la tabla, a la que se le asignará, en cada iteración, el valor de un elemento. El bucle se ejecutará tantas veces como elementos existan en `tabla`. Es importante tener en cuenta que la variable es una copia de cada elemento, y que en el caso de que se modifique, estamos modificando una copia no el elemento de la tabla. Veamos cómo sumar todos los elementos de la tabla `sueldos`,

```
double total = 0;
for (double s : sueldo) {
    total += s;
}
```

5.8.3. Mostrar una tabla

Como se vio anteriormente, cada variable de tipo tabla contiene una referencia, o `null`. Por tanto, si ejecutamos,

```
System.out.println(t); //muestra referencia
```

no muestra el contenido de la tabla `t`, sino la referencia almacenada en la variable. Si deseamos mostrar el contenido de una tabla, tendremos dos opciones: recorrer la tabla, mostrando cada uno de sus elementos o utilizar la función estática `toString()` de la clase `Arrays` en combinación con `System.out.println()`, de la forma,

```
System.out.println(Arrays.toString(t));
```

que muestra el contenido de la tabla `t`. Veamos un ejemplo,

```
int t[] = {8, 41, 37, 22, 19};
System.out.println(Arrays.toString(t));
```

que muestra los valores de la tabla entre corchetes,

```
[8, 41, 37, 22, 19]
```

Si preferimos escribir nuestra propia implementación tendremos que recorrer la tabla y mostrar sus elementos, de la forma,

```
for (i = 0; i < t.length; i++) { //recorremos toda la tabla
    System.out.println(t[i]); //mostramos cada elemento
}
```

o utilizando *for-each*,

```
for (int v: t) {
    System.out.println(v);
}
```

Evidentemente, es más cómodo utilizar `toString()`.

5.8.4. Inicialización

Por defecto una tabla se inicializa con valores específicos (0 para tipos numéricos y `false` para booleanos). Si deseamos inicializar con un valor distinto, tendremos que recorrer la tabla y asignar a cada elemento el valor en cuestión. Existe un método de la clase `Arrays` que hace exactamente esto,

`static void fill(tipo t[], tipo valor)`: que inicializa todos los elementos de la tabla `t` con `valor`. Esta función está sobrecargada, siendo posible utilizarla con cualquier tipo primitivo, representado por *tipo*, con la restricción de que el tipo de la tabla coincida con el tipo del valor pasado como parámetro.

Veamos cómo inicializar la tabla sueldos con un valor de 1234.56,

```
Arrays.fill(sueldo, 1234.56); //inicializa todos los elementos de la tabla
```

Puede ocurrir que solo interese inicializar algunos elementos de una tabla, para ello disponemos de,

`static void fill(tipo t[], int desde, int hasta, tipo valor)`: que asigna los elementos de la tabla `t`, comprendidos entre los índices `desde` y `hasta`, sin incluir este último, con `valor`. *tipo* representa cualquier tipo primitivo, con la restricción de que el tipo de la tabla coincida con el tipo del valor pasado como parámetro.

Como ejemplo, veamos cómo inicializar los elementos con índices del 3 al 6 (en la llamada utilizaremos 7) de la tabla `sueldo`,

```
Arrays.fill(sueldo, 3, 7, 1234.56); //inicializa solo el rango 3..6
```

El rango de índices es el comprendido entre el 3 y el anterior al 7, es decir, del 3 al 6.

5.8.5. Comparación de dos tablas

Dos tablas no se pueden comparar directamente mediante el operador `==`, ya que este operador no compara los elementos de las tablas, sino sus referencias. Por ejemplo,


```
int t1[] = {7, 9, 20};
int t2[] = {7, 9, 20}; //t1 y t1 son iguales
System.out.println(t1 == t2); //sin embargo muestra false
```

ya que las referencias de ambas tablas son distintas.

Dos tablas son iguales si contienen los mismos elementos en el mismo orden. Para comparar dos tablas disponemos del método,

static boolean equals(*tipo* a[], *tipo* b[]): que compara las tablas a y b, elemento a elemento. En el caso de que sean iguales devuelve **true**, y en caso contrario, **false**.

Veamos cómo comparar las dos tablas anteriores,

```
System.out.println(Arrays.equals(t1, t2)); //muestra true
```

5.8.6. Búsqueda

Consiste en averiguar si entre los elementos de una tabla se encuentra, y en qué posición, un valor determinado. El algoritmo de búsqueda depende de si la tabla está ordenada, ya que una búsqueda en una tabla ordenada siempre es más eficiente —requiere menos operaciones— que hacerlo en una tabla desordenada.

Búsqueda en una tabla no ordenada

Se denomina búsqueda secuencial y consiste en un recorrido de la tabla donde se comprueban los valores de los elementos. El proceso finalizará cuando encontremos el valor buscado o cuando no existan más elementos que revisar. Dicho de otro modo, mientras no encontremos el valor buscado o el final de la tabla, hemos de continuar con la búsqueda. El siguiente algoritmo busca el valor **aBuscar**, que se suele llamar *clave de búsqueda*, en una tabla **t** desordenada, con **numElem** elementos.

```
/* búsqueda secuencial */
i = 0; //índice de búsqueda

while (i < numElem && //no hemos llegados al final de la tabla
      t[i] != aBuscar) { //no hemos encontrado el elemento buscado
    i++; //incrementamos el índice de búsqueda
}

if (i < numElem) {
    // aBuscar se encuentra en la posición i
    . . .
} else { //el índice se ha salido de rango
    // no encontrado
    . . .
}
```

Cuando salimos del bucle **while** es por dos motivos: o bien hemos encontrado el elemento buscado, o por el contrario, no lo hemos encontrado y no hay más elementos donde buscar.

Búsqueda en una tabla ordenada

Aprovechamos que la posición de los valores nos proporciona información extra. Supongamos que en la tabla `datos`, ordenada de forma creciente, buscamos el valor 20, y sabemos que `datos[5] = 16`. Sin necesidad de conocer más valores de la tabla, sabemos que,

- Los valores de `datos[0]...`, `datos[4]` serán menores o iguales que 16, y aquí es imposible encontrar el 20.
- `datos[5]` vale 16.
- En caso de encontrarse, el valor 20 estará después de `datos[5]`.

Esta propiedad de las tablas ordenadas se aprovecha en el algoritmo de búsqueda dicotómica —también llamado búsqueda binaria—, que comprueba si el valor a buscar se encuentra en el elemento central de la tabla. Con esta información sabe si debe seguir buscando el valor en la primera o en la segunda mitad de la tabla. El proceso se repite con la mitad donde es posible encontrar el valor, que se subdivide de nuevo en dos partes. El algoritmo continúa hasta encontrar el valor o hasta que no existan más partes donde buscar.

Podemos escribir nuestro propio algoritmo de búsqueda dicotómica pero, al ser una operación tan utilizada, se encuentra implementada en la clase `Arrays`,

`static int binarySearch(tipo t[], tipo aBuscar)`: que busca de forma dicotómica en la tabla `t`, que supone ordenada, el elemento con valor `aBuscar`. Devuelve el índice donde se ha encontrado la primera ocurrencia del elemento buscado o un valor negativo en caso contrario.

Veamos un ejemplo: deseamos buscar, en la tabla ordenada `precios`, si existe y en qué posición está, algún producto de 19.95 euros,

```
int pos = Arrays.binarySearch(precios, 19.95);
if (pos >= 0) {
    System.out.println("Encontrado en el índice: " + pos);
} else {
    System.out.println("Lo sentimos, no se ha encontrado.");
}
```

Cuando el elemento a buscar no se encuentra, el valor negativo devuelto tiene un significado especial: informa de la posición donde tendría que colocarse el elemento buscado para que la tabla continúe ordenada. El índice de inserción se calcula,

```
indiceInsercion = -pos - 1;
```

siendo `pos` el valor negativo devuelto por `binarySearch()`. Veamos un ejemplo,

```
int a[] = {2, 4, 5, 6, 9};
int pos = Arrays.binarySearch(a, 3); //pos vale -2
int indiceInsercion = -pos - 1; //vale 1
```

Es decir, si insertáramos el valor buscado 3 en la tabla debemos hacerlo en el índice 1 para que la tabla continúe ordenada.

El método anterior realiza la búsqueda en toda la tabla, pero puede ocurrir que solo interese buscar en un subconjunto de elementos. Para ello disponemos de,

static int binarySearch(*tipo* t[], int desde, int hasta, *tipo* aBuscar): que solo busca en los elementos comprendidos entre los índices **desde** y **hasta**, sin incluir en la búsqueda este último.

En los métodos de distintas clases de la API, cuando se describe un rango mediante dos índices, **desde** y **hasta**, es habitual que se incluya en el rango el valor **desde** y se excluya **hasta**.

5.8.7. Inserción

La forma de añadir un nuevo valor a una tabla depende de si está o no ordenada. Si el orden no importa, basta con insertar el nuevo dato detrás del último elemento utilizado, es decir, en el primer hueco libre. Y cuando los datos están ordenados, hemos de insertar el nuevo valor de forma que todos los valores sigan ordenados.

Inserción no ordenada

Veamos el algoritmo para insertar el valor **nuevo**, al final de la tabla **t**, con **numElem** elementos utilizados. El valor de **numElem** coincide con el índice donde vamos a insertar el nuevo elemento. Por tanto, con la tabla vacía, **numElem** se inicializa a 0 y se incrementa después de cada inserción.

```
/* insercción al final: en el primer elemento disponible */
if (numElem == t.length) {
    // tabla llena, no insertamos
    ...
} else {
    t[numElem] = nuevo;
    numElem++; //ahora tenemos un elemento más
}
```

Inserción ordenada

Primero, hay que buscar el lugar que le corresponde al nuevo valor, **indiceInsercion**. Una vez localizado, a partir de dicho lugar, se han de desplazar los valores una posición hacia el final de la tabla, para hacer un hueco donde insertarlo. De esta forma, la tabla continúa ordenada tras la inserción. Una vez localizado el lugar de inserción (**indiceInsercion**) hacemos un hueco comenzando por el final, desde el último índice utilizado (**numElem - 1**) hasta **indiceInsercion**

```
for (int i = numElem - 1; i > indiceInsercion; i--) {
    t[i] = t[i - 1]; //desplazamos haciendo un hueco
}
t[indiceInsercion] = nuevo; //asignamos el nuevo elemento
```

Veamos un algoritmo que aprovecha la información extra que proporcionan los datos ordenados, y directamente va desplazando valores, desde el final, para hacer un hueco,

```
int i = numElem - 1; //índice del último elemento válido

//comenzamos por el final, para no machacar ningún elemento
while (i >= 0 && t[i] > nuevo) { //mientras nuevo sea menor que t[i]
    //i >= 0 evita salirse de la tabla
    t[i + 1] = t[i]; //desplazamos haciendo un hueco
    i--;
}
t[i + 1] = nuevo; //i+1 es la posición del nuevo elemento
```

5.8.8. Borrado

El paso previo a eliminar un elemento de una tabla, es buscarlo. Una vez localizado, la operación dependerá del tipo de tabla con la que estemos trabajando.

Tabla no ordenada

Para eliminar un elemento en una tabla, después de buscarlo, lo sustituimos por el último dato de la tabla, con lo que conseguimos que no queden huecos y que todos los datos continúen contiguos. Para finalizar, decrementamos el número de elementos útiles, ya que el que estaba último ha sido duplicado y queda fuera de la zona de datos válidos. La Figura 5.11 muestra una tabla, antes y después de eliminar un elemento.

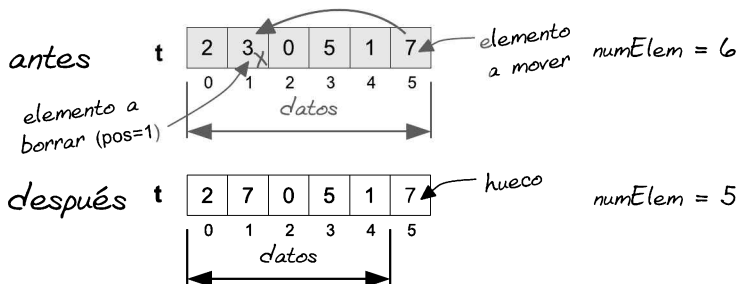


Figura 5.11. Borrado en una tabla no ordenada

```
//utilizamos un algoritmo de búsqueda secuencial
//para buscar la posición (pos) del elemento a borrar

if (pos == -1) { //no encontrado
    //elemento no encontrado, no hacemos nada
} else {
    t[pos] = t[numElem - 1];
    numElem--; //ahora tenemos un elemento menos
}
```

Tablas ordenadas

En tablas ordenadas, al eliminar el valor situado en la posición `pos`, tenemos que seguir manteniendo los demás valores contiguos y en el mismo orden. Para ello, tenemos que desplazar los valores que siguen a `pos` hacia la izquierda (véase Figura 5.12). Con esta técnica sobrescribimos el valor a borrar y obtenemos un hueco libre al final de la tabla.

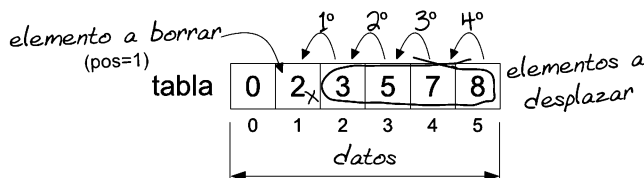


Figura 5.12. Desplazamiento de los elementos

El resultado final puede apreciarse en la Figura 5.13.

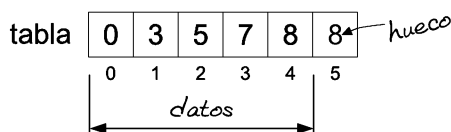


Figura 5.13. Valor eliminado y hueco

El código para desplazar los valores de una tabla comienza siempre copiando en cada elemento el valor que tiene a su derecha, desde la posición dada `pos` hacia el final de la tabla.

```
for (i = pos; i < numElem - 1; i++) {
    t[i] = t[i + 1];
}
numElem--; //tenemos un elemento menos
```

donde `numElem` es el número de elementos válidos o el tamaño de la tabla.

5.8.9. Ordenación

Ordenar una tabla consiste en cambiar de posición sus elementos para que, en conjunto, resulten ordenados. La clase `Arrays` permite ordenar tablas mediante el método,

`static void sort(tipo t[])`: que ordena los elementos de la tabla `t` de forma creciente. El método se encuentra sobrecargado para cualquier tipo primitivo; de ahí que *tipo* pueda ser `int`, `double`, etc.

Vamos a ver cómo ordenar una tabla,

```
int edad = {85, 19, 3, 23, 7}; //tabla desordenada
Arrays.sort(edad); //ordena la tabla. Ahora edad = [3, 7, 19, 23, 85]
```

Hay que señalar que el proceso de ordenación es muy costoso en tiempo. Por ejemplo, la ordenación de una tabla de 100 elementos puede requerir, en el peor de los casos, unas 600 operaciones. Dependiendo de la posición de los elementos es posible ordenar la tabla con un menor número de operaciones. Cuando decidimos ordenar una tabla tenemos que estar seguros de que la cantidad de tiempo empleado compensa otras operaciones. Por ejemplo, buscar un elemento en una tabla desordenada de 100 elementos requiere, en el peor de los casos, recorrer los 100 elementos; mientras que hacer una búsqueda en una tabla ordenada de 100 elementos requiere a lo sumo 7 operaciones. Con estas cifras, es totalmente ineficiente ordenar una tabla de 100 elementos (que requiere un máximo de 600 operaciones) para que la búsqueda ordenada sea más eficiente. Ordenar la tabla tendrá sentido cuando vayamos a realizar muchas búsquedas.

5.8.10. Copia

El procedimiento para realizar la copia exacta de una tabla es:

- Crear una nueva tabla, que llamaremos tabla *destino*, del mismo tipo y tamaño que la original.
- Recorrer la tabla original, copiando el valor de cada elemento en su lugar correspondiente en la tabla destino.

Sin embargo, `Arrays` proporciona:

static *tipo*[] copyOf(*tipo* origen[], int longitud): construye y devuelve una tabla de tamaño *longitud* y copia en ella el contenido de *origen*. Si el tamaño de la copia es menor que la original, se copian los elementos que caben. En el caso de que la nueva tabla sea mayor que la original se copia todo su contenido inicializando el resto de los elementos. Este método, como la mayoría de los métodos de `Arrays`, está sobrecargado para todos los tipos primitivos.

Vemos un ejemplo,

```
int t[] = {1, 2, 1, 6, 23};
int a[], b[];
a = Arrays.copyOf(t, 3); //a = [1, 2, 1]
b = Arrays.copyOf(t, 10); //b = [1, 2, 1, 6, 23, 0, 0, 0, 0, 0]
```

Existe otro método que también realiza una copia de una tabla, pero en este caso de un subconjunto de elementos,

static *tipo*[] copyOfRange(*tipo* origen[], int desde, int hasta): que construye y devuelve una tabla donde se han copiado los elementos de *origen* comprendidos entre los índices *desde* y *hasta*, sin incluir este último.

Un ejemplo,

```
int t[] = {7, 5, 3, 1, 0};
int a[] = Arrays.copyOfRange(t, 1, 4); //a = [5, 3, 1]
```

que realiza una copia desde los índices 1 al 3 (el anterior al 4).

5.9. Tablas *n*-dimensionales

Hasta el momento las tablas que hemos utilizado han sido unidimensionales: solo tienen longitud; dicho de otra forma, los elementos solo se extienden a lo largo del eje *X*, y basta con un índice para recorrerlas.

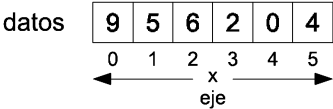


Figura 5.14. Tabla unidimensional

Pero puede ocurrir que nuestros datos se refieran a entidades que se caracterizan por más de una propiedad, de las cuales alguna o algunas sirven para identificarlo. En este caso, no nos basta con un solo índice para describirlas.

5.9.1. Matrices bidimensionales

Podemos ampliar el concepto de tabla haciendo que los elementos se extiendan en dos dimensiones, utilizando los ejes *X* e *Y*. Ahora la tabla posee longitud y anchura (véase la Figura 5.15). Para identificar cada elemento de una tabla unidimensional hemos utilizado un índice; para las tablas bidimensionales, compuestas por filas y columnas, se necesita un par de índices [*x*] [*y*]. Una tabla bidimensional recibe el nombre de *matriz*.

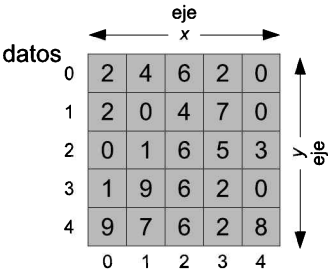


Figura 5.15. Tabla bidimensional de 5×5 elementos

La declaración de una tabla bidimensional se hace de la forma:

```
tipo nombreTabla[] [];
```

En nuestro ejemplo anterior:

```
int datos[] [];
```

A continuación creamos la tabla, indicando el tamaño de cada dimensión:

```
datos = new int[5][5];
```

y con ello, estamos reservando espacio en la memoria para (5 × 5) 25 elementos.

Los algoritmos que utilizan matrices requieren dos bucles anidados. Un bucle se encarga del índice para la dimensión X y el otro para el índice del eje Y . Veamos un ejemplo para introducir por teclado la matriz `datos`.

```
for (i = 0; i < 5; i++) { //eje X
    for (j = 0; j < 5; j++) { //eje Y
        datos[i][j] = sc.nextInt(); //leemos el elemento [i][j]
    }
}
```

5.9.2. Matrices tridimensionales

Añadiendo una nueva dimensión podemos crear matrices con anchura, altura y profundidad, o matrices tridimensionales (Figura 5.16). Estas utilizan tres índices ($[x][y][z]$) para especificar cada elemento que la compone.

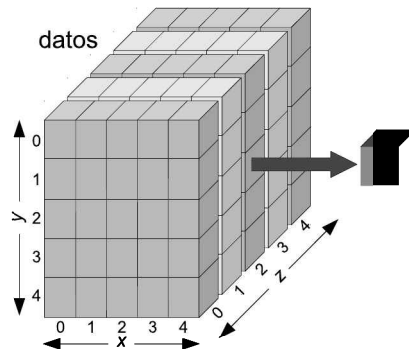


Figura 5.16. Tabla tridimensional de $5 \times 5 \times 5$

5.9.3. Matrices n -dimensionales

Dibujar una matriz n -dimensional es algo complicado. Nosotros vivimos en un mundo 4-dimensional, con tres dimensiones para localizar cada objeto en el espacio y una cuarta dimensión, el tiempo, para ver la evolución de un objeto en movimiento. Pero más allá de nuestro mundo, es complicado representar, o siquiera imaginar, una matriz multidimensional. Un truco sencillo consiste en descomponer la matriz en otras más simples. Por ejemplo, una matriz de 5 dimensiones puede verse como una matriz tridimensional, donde en cada elemento de la matriz se almacena una matriz bidimensional. De los cinco índices necesarios para identificar los elementos de una matriz 5-dimensional, podemos utilizar los tres primeros en la matriz tridimensional y utilizar los otros dos índices para situarnos en la segunda matriz bidimensional.

Pero el hecho de que no podamos dibujarla ni imaginarla no significa que no se puedan manipular sus elementos.

Las matrices n -dimensionales son útiles para manejar la información atendiendo a criterios de clasificación. No es necesario que la información tenga una representación gráfica. Veamos un ejemplo: supongamos una máquina que procesa naranjas y necesitamos, por motivos de calidad, clasificarlas y llevar la cuenta del número de frutas recogidas de cada tipo, atendiendo a cinco criterios:

- Diámetro.
- Color.
- Maduración.
- Forma.
- Peso.

Al utilizar cinco criterios, lo más apropiado para almacenar los datos es una matriz 5-dimensional, haciendo corresponder cada dimensión con un criterio de clasificación. Falta, para cada una de las dimensiones (criterios), formalizar una correspondencia entre los posibles valores reales de un criterio (color: naranja, amarillo o verde; nivel de maduración: madura o inmadura; etc.) con los tamaños de cada dimensión, que utilizaremos como índices. Una posible correspondencia puede ser:

- Diámetro.
 0. Pequeño, diámetro ≤ 4 cm.
 1. Mediano, $4 \text{ cm} < \text{diámetro} \leq 8$ cm.
 2. Grande, $8 \text{ cm} < \text{diámetro}$.
- Color.
 0. Naranja.
 1. Amarillo.
 2. Verde.
- Maduración.
 0. Pasada.
 1. Óptima.
 2. Ligeramente inmadura.
 3. Completamente inmadura.
- Forma.
 0. Redondeada.
 1. Otra forma.

- Peso.

0. Menos de 100 g.
1. Entre 100 y 200 g.
2. Entre 200 y 300 g.
3. Entre 300 y 400 g.
4. Entre 400 y 500 g.
5. Más de 500 g.

Crearemos la variable `naranjas`, que será una matriz con 5 dimensiones, donde cada una de ellas representa un criterio:

```
naranjas[diámetro][color][maduración][forma][peso]
```

La declaración y creación de la variable es:

```
int naranjas[][][][][];  
naranjas = new int[3][3][5][2][6];
```

Si la máquina contabiliza 25 naranjas con:

1. Diámetro de 11 cm: primer índice 2.
2. De un color naranja intenso: segundo índice 0.
3. En su punto óptimo de maduración: tercer índice 1.
4. La forma es redonda: cuarto índice 0.
5. Pesa 285 g: quinto índice 2.

Haremos la asignación:

```
naranjas[2][0][1][0][2] = 25;
```

Ejercicios de tablas

5.1. Diseñar un programa que solicite al usuario que introduzca por teclado 5 números decimales a continuación, debe mostrar los números en el mismo orden que se han introducido.

```
import java.util.Arrays;  
import java.util.Locale;  
import java.util.Scanner;  
  
/*  
 * Para guardar 5 números es posible utilizar cinco variables escalares, pero es mucho  
 * más cómodo una tabla con 5 elementos. Los números pueden tener decimales, por  
 * tanto declaramos la tabla de tipo double.  
 * Una vez creada la tabla tendremos que realizar dos recorridos, el primero, para  
 * insertar los valores y, el segundo, para mostrar los datos. */
```

```
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US); //para separar los decimales con punto (no con coma)
        double t[] = new double[5]; //declaración y creación de la tabla con tamaño 5

        for (int i = 0; i < 5; i++) { //recorremos para leer lo valores
            System.out.print("Introduzca un número: ");
            t[i] = sc.nextDouble();
        }

        System.out.println("Los números son:");
        for (int i = 0; i < 5; i++) { // recorremos para mostrar
            System.out.println(t[i]);
        }
        //podemos ahorrarnos el segundo recorrido, aprovechando Arrays.toString()
        System.out.println("Otra forma de mostrar la tabla: ");
        System.out.println(Arrays.toString(t)); //muestra t
    }
}
```

5.2. Escribir una aplicación que solicite al usuario cuántos números desea introducir. A continuación, se introducirá por teclado esa cantidad de números enteros, y por último, los mostrará en el orden inverso al introducido.

```
import java.util.Scanner;

/*
 * Primero leeremos la cantidad de números a introducir. Con esta información crearemos
 * una tabla del tamaño adecuado para albergar todos los datos que se introducirán por
 * teclado. Por último, recorreremos la tabla, pero comenzando en el último elemento y
 * finalizando en el primero, con lo que conseguimos mostrarlos en orden inverso.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Cuántos números desea introducir: ");
        int cuantosNumeros = sc.nextInt();

        int t[] = new int[cuantosNumeros]; //tabla con el tamaño adecuado

        for (int i = 0; i < t.length; i++) { //recorremos desde 0 hasta t.length-1
            System.out.print("Introduzca un número: ");
            t[i] = sc.nextInt();
        }

        System.out.println("Los números en orden inverso son: ");
        for (int i = t.length - 1; i >= 0; i--) { // recorremos en orden inverso
            System.out.println(t[i]);
        }
        //en este caso no podemos utilizar Arrays.toString() para mostrar la tabla
    }
}
```

5.3. Introducir por teclado un número n ; a continuación solicitar al usuario que teclee n números. Realizar la media de los números positivos, la media de los negativos y contar el número de ceros introducidos.

```
import java.util.Scanner;
/*
 * Una vez leído n, crearemos una tabla de n enteros. A continuación leeremos n números
 * que guardaremos en cada elemento de la tabla. Por último, recorreremos la tabla y
 * procesaremos, sumando los valores (positivo y negativos), para calcular las medias
 * y contar los ceros. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Escriba n: ");
        int n = sc.nextInt(); //leemos n
        int numeros[] = new int[n]; // declaramos y creamos una tabla de n elementos
        int sumaPositivos = 0, // acumulador de números positivos
            contPositivos = 0, // contador de números positivos
            sumaNegativos = 0, // acumulador de números negativos
            contNegativos = 0, // contador de números negativos
            contCeros = 0; // contador de ceros

        for (int i = 0; i < numeros.length; i++) { // recorremos para leer los números
            System.out.print("Introduzca un número: ");
            numeros[i] = sc.nextInt();
        }
        // Utilizamos un bucle para leer los datos y usaremos otro para procesar. Se
        // podría incluir todo en un solo bucle. Es importante comprender que para leer
        // los elementos de una tabla no es posible utilizar un bucle for-each

        // procesamos utilizando un for-each
        for (int x : numeros) { //x es una copia del valor de cada elemento
            if (x == 0) { //si es un cero
                contCeros++; //incrementamos el contador de ceros
            } else {
                if (x > 0) { //si es positivo
                    sumaPositivos += x; //acumulamos el número positivo para hacer la media
                    contPositivos++; //e incrementamos la cantidad de positivos
                } else {
                    sumaNegativos += x; //igual para los negativos
                    contNegativos++;
                }
            }
        }
        // al hacer las medias hay que tener cuidado de no realizar una división por cero
        if (contPositivos == 0) {
            System.out.println("Imposible realizar la media de los positivos");
        } else {
            System.out.println("Media de los positivos: "
                + (double) sumaPositivos / contPositivos); // el cast es para
            // evitar la división entera, que suprime la parte decimal
        }

        if (contNegativos == 0) {
            System.out.println("Imposible realizar la media de los negativos");
        } else {
            System.out.println("Media de los negativos: "
                + (double) sumaNegativos / contNegativos);
        }
        System.out.println("Cantidad de ceros: " + contCeros);
    }
}
```

- 5.4. Implementar un programa que inicialice una tabla con nuestros números favoritos. A continuación, pedir al usuario el índice de un elemento que será eliminado de la tabla. Continuaremos eliminando elementos hasta que el índice introducido sea negativo o

hasta que no existan más elementos que borrar. Es imprescindible controlar que el índice leído corresponde a un dato válido.

```
import java.util.Arrays;
import java.util.Scanner;

/*
 * En este ejercicio eliminaremos un elemento de la tabla, sin dejar hueco. La tabla no
 * no tiene por qué estar ordenada y, aunque al principio esté completamente llena,
 * tendremos que utilizar un indicador para llevar la cuenta de los elementos que
 * quedan tras los borrados. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t[] = {23, 8, 1, -3, 17, 5, 99}; //mis números favoritos

        int numElem = t.length; //el número de elementos válidos de t
        System.out.println(Arrays.toString(t)); //mostramos la tabla completa
        System.out.println("Índice del elemento a borrar: ");
        int indiceBorrar = sc.nextInt();
        while (indiceBorrar >= 0 && numElem != 0) { //mientras el índice sea positivo y
            if (indiceBorrar < numElem) { //existan elementos válidos en la tabla
                //algoritmo de eliminación en una tabla no ordenada
                t[indiceBorrar] = t[numElem - 1]; //sustituimos el valor del elemento por el
                //último elemento válido
                numElem--; //ahora tenemos un dato menos en la tabla
                mostrarTabla(t, numElem); //muestra solo los datos válidos de t
            } else {
                System.out.println("No existe elemento a borrar");
            }
            System.out.println("Índice del elemento a borrar: ");
            indiceBorrar = sc.nextInt(); //leemos de nuevo el índice a borrar
        }
    }

    /* Esta función muestra solo los primeros n elementos de la tabla t. Suponemos que
     * la tabla pasada como parámetro siempre tiene un tamaño mayor o igual que n. */
    static void mostrarTabla(int a[], int n) {
        System.out.print("[");
        for (int i = 0; i < n; i++) {
            System.out.print(a[i]+ " ");
        }
        System.out.println("]");
    }
}
```

5.5. Desarrollar el juego *la cámara secreta*, que consiste abrir una cámara mediante su combinación secreta, que está formado por una combinación de dígitos del uno al cinco. El jugador especificará cuál es la longitud de la combinación, a mayor longitud mayor será la dificultad del juego. La aplicación genera, de forma aleatoria, una combinación secreta que el usuario tendrá que acertar. En cada intento se muestra como pista, para cada dígito de la combinación introducido por el jugador, si es mayor, menor o igual que el correspondiente en la combinación secreta.

```
import java.util.Arrays;
import java.util.Scanner;

/* La aplicación genera de forma aleatoria la combinación secreta, que se solicita al
 * usuario. El juego termina cuando la combinación secreta coincide con la introducida */
```

```

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Longitud de la combinación secreta: ");
        int longitud = sc.nextInt();
        int combSecreta[] = new int[longitud]; //tabla con la combinación secreta
        int combJugador[] = new int [longitud]; //combinación introducida por el jugador

        generaCombinacion(combSecreta); //generamos aleatoriamente la combinación secreta
        System.out.println(Arrays.toString(combSecreta));
        System.out.println("Escriba una combinación");
        leeTabla(combJugador);

        while (!Arrays.equals(combSecreta, combJugador)) { //terminamos cuando sean iguales
            muestraPistas(combSecreta, combJugador); //mostramos las pistas del juego
            System.out.println("Escriba una combinación: ");
            leeTabla(combJugador); //volvemos a pedir otra combinación
        }

        System.out.println("¡La cámara está abierta!");
    }

    // Esta función inicializa los valores de la tabla pasada como parámetro con valores
    //aleatorio. La constante MAX determina el valor máximo que se asigna a un elemento,
    //estando comprendido en el rango 1..MAX
    static void generaCombinacion(int t[]) {
        final int MAX = 5; //cada dígito de la combinación estará en el rango 1..MAX
        for (int i = 0; i < t.length; i++) {
            t[i] = (int) (Math.random()*MAX+1); //número aleatorio de 1 a MAX
            //t referencia a la tabla combSecreta del programa principal. Por ese motivo
            //asignar un valor a t[i] es lo mismo que hacerlo a combSecreta[i]
        }
    }

    //Recorre la tabla pasada como parámetro y asigna a cada elemento el valor leído
    static void leeTabla(int t[]) {
        Scanner sc = new Scanner(System.in);
        for (int i = 0; i < t.length; i++) { //recorremos para leer
            t[i] = sc.nextInt();
        }
    }

    //Recorre las dos tablas, secret y jug, e indica para cada elemento de la combinación
    //introducida por el usuario si es mayor, menor o igual que el equivalente en la
    //combinación secreta
    static void muestraPistas(int secret[], int jug[]) {
        for (int i = 0; i < jug.length; i++) { //recorremos ambas tablas con el bucle
            System.out.print(jug[i]);
            if (secret[i] > jug[i]) { //comparamos el i-ésimo elemento de ambas tablas
                System.out.println(" mayor");
            } else if (secret[i] < jug[i]) {
                System.out.println(" menor");
            } else {
                System.out.println(" igual");
            }
        }
    }
}

```

5.6. Diseñar una aplicación para gestionar un campeonato de programación, donde se introducen la puntuación (enteros) obtenidos por 5 programadores, conforme van terminando su prueba. La aplicación debe mostrar las puntuaciones ordenadas de los 5 participantes. En ocasiones, cuando finalizan los 5 participantes anteriores, se suman

al campeonato un máximo de 3 programadores de exhibición, cuyos puntos se incluyen con el resto. La forma de especificar que no intervienen más programadores de exhibición es introducir como puntuación un -1. La aplicación debe mostrar, finalmente, los puntos ordenados de todos los participantes.

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * En este ejercicio leeremos una serie de datos y los ordenaremos. A continuación,
 * realizaremos una inserción ordenada (por cada programador de exhibición).
 * Una mala idea sería insertar al final la puntuación de los programadores de
 * exhibición y volver a ordenar, ya que esto es muy costoso en tiempo. Es mucho más
 * eficiente una inserción ordenada.
 * A lo sumo participan 8 (5+3) programadores: creamos una tabla de tamaño 8. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int puntos[] = new int[8]; // como máximo intervienen 8 programadores

        for (int i = 0; i < 5; i++) {
            System.out.print("Puntos programador (" + (i + 1) + "): ");
            puntos[i] = sc.nextInt(); // leemos los datos, que no están ordenados
        }

        Arrays.sort(puntos, 0, 5); // ordenamos los elementos con índices del 0 hasta el 5
        System.out.println(Arrays.toString(puntos)); // mostrará tres 0, al final

        int numElem = 5; // número de elementos útiles en la tabla puntos
        System.out.print("Puntos del programador de exhibición: ");
        int puntosProgExh = sc.nextInt(); // leemos la puntuación del prog. de exhibición
        while (puntosProgExh != -1 && numElem < puntos.length) {
            int pos = Arrays.binarySearch(puntos, 0, numElem, puntosProgExh); // buscamos
            int indiceInsercion; // lugar donde insertar para que la tabla siga ordenada
            if (pos < 0) {
                indiceInsercion = -pos - 1; // índice para que la tabla esté ordenada
            } else {
                indiceInsercion = pos; // puntuación repetida, ya está en la tabla
            }

            // desplazaremos los elementos desde la posición pos hasta el
            // final, así haremos un hueco para la puntuación del programador de exhibición
            for (int i = numElem - 1; i >= indiceInsercion; i--) {
                puntos[i + 1] = puntos[i];
            }

            // asignamos puntosProgExh en su posición para que la tabla siga ordenada
            puntos[indiceInsercion] = puntosProgExh;
            numElem++; // ahora tenemos un elemento válido más en la tabla
            if (numElem < puntos.length) { // la última lectura no tiene sentido
                System.out.print("Puntos del programador de exhibición: ");
                puntosProgExh = sc.nextInt(); // leemos la puntuación del prog. de exhibición
            }
        }

        System.out.println("Puntuación final: ");
        System.out.println(Arrays.toString(puntos));
    }
}
```

- 5.7. Leer una serie de 6 enteros que se almacenarán en una tabla que hay que ordenar y mostrar. Leer otra serie de 6 enteros, que también se guardarán en una tabla y se

mostrarán ordenados. A continuación, fusionar las dos tablas en una tercera, de forma que los 12 números sigan ordenados. Fusionar dos tablas ordenadas significa copiar en el orden correcto para que los datos resultantes continúen ordenados, sin necesidad de volver a realizar una ordenación.

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * En este ejercicio fusionaremos dos tablas ordenadas(a y b) en una tercera (c), de
 * forma que siga ordenada. Comparamos un elemento de cada tabla, y copiamos el menor
 * en c. El índice correspondiente se incrementará. Hay que tener cuidado cuando
 * se ha copiado todos los elementos de la tabla a o de la tabla b, en este caso,
 * tendremos que copiar el resto de los elementos de la otra tabla en c. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final int TAM = 6; //constante para el tamaño de las tablas
        int a[] = new int[TAM]; //tabla para la primera serie de números

        System.out.println("Escriba la primera serie de números: "); // leemos a
        for (int i = 0; i < a.length; i++) {
            System.out.print("Introduzca número: ");
            a[i] = sc.nextInt();
        }
        Arrays.sort(a); //ordenamos
        System.out.println(Arrays.toString(a)); // y mostramos

        //hacemos lo mismo para la tabla b
        int b[] = new int[TAM]; //tabla para la segunda serie de números
        System.out.println("Escriba la segunda serie de números: "); // leemos b
        for (int i = 0; i < b.length; i++) {
            System.out.print("Introduzca número: ");
            b[i] = sc.nextInt();
        }
        Arrays.sort(b); //ordenamos
        System.out.println(Arrays.toString(b)); //y mostramos

        // creamos c
        int c[] = new int[2 * TAM]; //para albergar los datos de a y b tiene un tamaño doble

        // comenzamos a fusionar a y b en c
        int indA = 0, // utilizamos como índice de a
            indB = 0, // " " " de b
            indC = 0; // " " " de c
        while (indA < TAM && indB < TAM) {
            if (a[indA] < b[indB]) { // si el elemento menor de las dos tablas es
                c[indC] = a[indA]; // de a, se copia en c
                indA++; // incrementamos indA para pasar al siguiente elemento de a
            } else { // si es de b
                c[indC] = b[indB]; //se copia en c
                indB++; // incrementamos indB para pasar al siguiente elemento de b
            }
            indC++; // como hemos copiado a c[indC], incrementamos indC para que, en
            // la siguiente vuelta, utilicemos el siguiente hueco de la tabla
        }
        // se sale del bucle porque en alguna de las tablas (a o b) se ha copiado el
        //último elemento en c (no hay nada más que copiar)

        if (indA == TAM) { // en este caso hemos copiado la tabla a en c
            while (indB < TAM) { // queda por copiar un resto de b
                c[indC] = b[indB];
                indB++;
                indC++;
            }
        }
    }
}
```



```

    } else { // hay que copiar el resto de a en c
        while (indA < TAM) {
            c[indC] = a[indA];
            indA++;
            indC++;
        }
    }

    System.out.println("Mostramos todos los datos: ");
    System.out.println(Arrays.toString(c));
}
}

```

5.8. Implementar la función sinRepetidos() con el prototipo,

```
int[] sinRepetidos(int t[])
```

que construye y devuelve una tabla del tamaño apropiado, con los elementos de *t*, donde se han eliminado los datos repetidos.

```

import java.util.Arrays;
/*
 * Programa principal para comprobar el funcionamiento. Para evitar leer los datos
 * construimos la tabla mediante asignación. */
public class Main {

    public static void main(String[] args) {
        int sin[], con[] = {1, 2, 3, 2, 1, 3, 4, 2, 3, 5};
        sin = sinRepetidos(con);
        System.out.println(Arrays.toString(sin));
    }

    /* Vamos a realizar un doble recorrido de la tabla t. En el recorrido principal, se
     * comprueba el elemento i (t[i]), para cada elemento, vamos a volver a recorrer la
     * tabla, en este caso, entre 0 y i-1, donde comprobamos si se encuentra t[i].
     * Si encontramos el elemento, es que es una repetición, lo que hacemos es eliminar
     * t[i] (borrado en tabla no ordenada). */
    static int[] sinRepetidos(int t[]) {
        int copia[] = Arrays.copyOf(t, t.length); //hacemos una copia exacta de t. No
        //podemos trabajar con t, porque vamos a modificar la tabla. Modificamos la copia.
        int numElem = copia.length; //número de elementos válidos en la tabla copia
        int i = 0; //índice del elemento a comprobar

        while (i < numElem) {
            //vamos a ver si entre los elementos de 0 a i-1, se encuentra copia[i]
            int aBuscar = copia[i];
            int j = 0; //índice para el segundo recorrido: para buscar copia[i] (o aBuscar)

            // en este segundo recorrido, el número de elementos es i
            while (j < i && copia[j] != aBuscar) {
                j++; //incrementamos el índice del recorrido secundario
            }

            if (j < i) {
                //aBuscar (copia[i]) se encuentra en la posición j. Tenemos que eliminarlo
                //algoritmo de borrado no ordenado
                copia[i] = copia[numElem - 1]; //en copia[i] se copia el último elemento
                numElem--; //tenemos un elemento menos en la tabla
                //no incrementamos i, para ver si el nuevo copia[i] está repetido
            } else { //el índice se ha salido de rango, no encontrado
                i++; //incrementamos para comprobar el siguiente elemento
            }
        }
    }
}

```

```
        return Arrays.copyOf(copia, numElem); //devolvemos una copia de la tabla "copia" de
        //longitud numElem elementos
    }
}
```

5.9. Queremos desarrollar una aplicación que nos ayude a gestionar las notas de un centro educativo. Cada grupo (o clase) está compuesto por 5 alumnos. Se pide leer las notas (números enteros) del primer, segundo y tercer trimestre de un grupo. Debemos mostrar al final la nota media del grupo en cada trimestre, y la media del alumno que se encuentra en la posición *pos* (que se lee por teclado).

Solución a)

```
import java.util.Scanner;
/*
 * Utilizaremos una tabla para guardar las notas de cada trimestre. El i-ésimo elemento
 * de cada tabla corresponde a la nota del mismo alumno, en los distintos trimestres.*/
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final int NUM_ALUM = 5; //número de alumnos por grupo
        double mediaAlumno;

        // declaramos y creamos las tablas necesarias
        int primer[] = new int[NUM_ALUM]; //notas de cada trimestre
        int segundo[] = new int[NUM_ALUM];
        int tercer[] = new int[NUM_ALUM];

        System.out.println("Notas del primer trimestre:");
        leerNotas(primer); //leemos las notas del primer trimestre
        System.out.println("Notas del segundo trimestre:");
        leerNotas(segundo); //leemos las notas del segundo trimestre
        System.out.println("Notas del tercer trimestre:");
        leerNotas(tercer); //leemos las notas del tercer trimestre

        // calculamos las medias
        int sumaPrimer = 0, // ponemos a 0 los acumuladores de cada trimestre
            sumaSegundo = 0,
            sumaTercer = 0;

        for (int i = 0; i < NUM_ALUM; i++) { //recorremos las tres tablas a la vez
            sumaPrimer += primer[i]; //acumulamos las notas para calcular las medias
            sumaSegundo += segundo[i];
            sumaTercer += tercer[i];
        }

        // mostramos datos
        System.out.println("Media primer trimestre: " + (double) sumaPrimer / NUM_ALUM);
        System.out.println("Media segundo trimestre: " + (double) sumaSegundo / NUM_ALUM);
        System.out.println("Media tercer trimestre: " + (double) sumaTercer / NUM_ALUM);
        System.out.println();

        // leemos la posición del alumno que nos interesa, suponemos que el usuario
        // introduce un índice que no se sale de la tabla
        System.out.print("Introduzca posición del alumno: ");
        int pos = sc.nextInt();

        // la media del alumno es la suma de sus notas entre 3
        mediaAlumno = (double) (primer[pos] + segundo[pos] + tercer[pos]) / 3;
        System.out.println("La media del alumno es: " + mediaAlumno);
    }
}
```

```

//esta función lee las notas de un grupo en la tabla t
static void leerNotas(int t[]) {
    Scanner sc = new Scanner(System.in);
    for (int i = 0; i < t.length; i++) {
        System.out.print("Alumno (" + i + "): ");
        t[i] = sc.nextInt();
    }
}
}
}

```

Solución b)

```

import java.util.Scanner;
/* En esta versión, en lugar de utilizar una tabla para cada trimestre, utilizaremos una
 * tabla bidimensional, con tamaño [3][5], donde el primer índice indica el trimestre
 * y el segundo determina el alumno. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final int NUM_ALUM = 5; //número de alumnos por grupo
        // declaramos y creamos la tabla bidimensional
        int notas[][] = new int[3][NUM_ALUM]; //notas de cada trimestre

        for (int trimestre = 0; trimestre < 3; trimestre++) {
            System.out.println("Notas para el trimestre " + (trimestre + 1));
            leerNotas(notas, trimestre); //leemos las notas del trimestre
        }

        // calculamos las medias, utilizamos una tabla de 3 elementos, donde acumulamos
        //la notas de cada trimestre
        int suma[] = new int[3]; // por defecto los elementos se inicializan a 0

        for (int alum = 0; alum < NUM_ALUM; alum++) { //recorremos los alumnos, y para
            for (int trim = 0; trim < 3; trim++) { //cada uno, recorremos los trimestres
                suma[trim] += notas[trim][alum]; //el primer índice siempre es el trimestre,
                // y el segundo índice siempre es el alumno
            }
        }

        // mostramos datos
        System.out.println("Media primer trimestre: " + (double) suma[0] / NUM_ALUM);
        System.out.println("Media segundo trimestre: " + (double) suma[1] / NUM_ALUM);
        System.out.println("Media tercer trimestre: " + (double) suma[2] / NUM_ALUM);
        System.out.println();

        // leemos la posición del alumno que nos interesa, suponemos que el usuario
        // introduce un índice que no se sale de la tabla
        System.out.print("Introduzca posición del alumno: ");
        int pos = sc.nextInt();

        // la media del alumno es la suma de sus notas entre 3
        double mediaAlumno = (double) (notas[0][pos] + notas[1][pos] + notas[2][pos]) / 3;
        System.out.println("La media del alumno es: " + mediaAlumno);
    }

    //esta función lee las notas de un grupo en la tabla t, para el trimestre indicado
    static void leerNotas(int t[][], int trimestre) {
        Scanner sc = new Scanner(System.in);
        for (int i = 0; i < t[trimestre].length; i++) { //longitud del segundo índice
            System.out.print("Alumno (" + i + "): ");
            t[trimestre][i] = sc.nextInt();
        }
    }
}
}

```

- 5.10. Leer y almacenar n números enteros en una tabla, a partir de la que se construirán otras dos tablas con los elementos con valores pares e impares de la primera, respectivamente.

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * Tras leer la tabla con los n números, tendremos que contar cuántos elementos hay con
 * valor par y cuántos impares. Con esta información se crean las tablas del tamaño
 * oportuno. Por último, volvemos a procesar la tabla inicial, para copiar clasificar y
 * copiar los valores en las tablas par o impar. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t[]; //tabla para los datos iniciales

        System.out.print("Escriba n: ");
        int n = sc.nextInt(); // n es el número de datos a leer
        t = new int[n]; //creamos la tabla de tamaño n

        // leemos del teclado los valores de la tabla
        for (int i = 0; i < t.length; i++) {
            System.out.print("Introduzca un número: ");
            t[i] = sc.nextInt();
        }

        // contamos la cantidad de elementos pares e impares. También se
        // podrían contar solo lo pares y calcular los impares= n - pares
        int contPar = 0, //contador de pares. También se usará como índice de la tabla par
            contImpar = 0; //igual para los impares

        for (int i = 0; i < t.length; i++) { // consideramos 0 como un número par
            if (t[i] % 2 == 0) { // si t[i] es par
                contPar++;
            } else {
                contImpar++;
            }
        }

        //contado el número de elementos pares e impares, declaramos y creamos las tablas
        int par[] = new int[contPar]; //la tabla par albergará contPar elementos
        int impar[] = new int[contImpar]; //la tabla impar almacenará contImpar elementos

        // volvemos a procesar para copiar cada elemento en la tabla adecuada
        contPar = 0; //ahora reutilizamos estas variables como índices de sus
        contImpar = 0; // respectivas tablas

        for (int i = 0; i < t.length; i++) {
            if (t[i] % 2 == 0) {
                par[contPar] = t[i]; // asignamos el elemento, que es par
                contPar++;
            } else {
                impar[contImpar] = t[i]; // asignamos el elemento, que es impar
                contImpar++;
            }
        }

        System.out.println("Tabla par: " + Arrays.toString(par));
        System.out.println("Tabla impar: " + Arrays.toString(impar));
    }
}
```

- 5.11. Escribir un programa que solicite los elementos de una matriz de tamaño 4×4 . La aplicación debe decidir si la matriz introducida corresponde a una matriz mágica, que

es aquella donde la suma de lo elementos de cualquier fila o de cualquier columna vale lo mismo.

```
import java.util.Scanner;
/*
 * La forma de comprobar que una matriz es mágica consiste en realizar la suma de todas
 * las filas y columnas, comprobando que todas son iguales. Utilizaremos como patrón la
 * suma de la primera fila. Se podría utilizar como patrón la suma de cualquier fila o
 * columna.
 * Este algoritmo es mejorable, ya que cuando tenemos la certeza que la matriz no es
 * mágica, no es necesario seguir realizando sumas de filas y columnas. Invitamos al
 * lector a mejorar el algoritmo. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int patron = 0; //suma de la primera fila, que utilizamos como patrón
        final int TAM = 4; // el tamaño se define como una constante. Si queremos utilizar
        //un tamaño distinto para la matriz, basta con redefinir la constante.
        int matriz[][] = new int[TAM][TAM];

        for (int i = 0; i < TAM; i++) { // leemos
            for (int j = 0; j < TAM; j++) {
                System.out.print("Elemento [" + i + "][" + j + "]: ");
                matriz[i][j] = sc.nextInt();
            }
        }

        boolean esMagica = true; //suponemos la matriz como mágica, cuando encontremos
        //un caso que contradiga esto, modificaremos la variable a false

        // obtenemos la suma (patrón) para comparar con el resto.
        for (int c = 0; c < TAM; c++) {
            patron += matriz[0][c]; //calculamos la suma de la primera fila
        }

        //sumamos las columnas
        for (int c = 0; c < TAM; c++) { // utilizamos c para las columnas
            int sumaColumna = 0;
            for (int f = 0; f < TAM; f++) { // utilizamos f para las filas
                sumaColumna += matriz[f][c];
            }
            if (sumaColumna != patron) {
                esMagica = false;
            }
        }

        // sumamos todos los elementos que forman parte de una fila
        for (int f = 0; f < TAM; f++) { // f para las filas
            int sumaFila = 0;
            for (int c = 0; c < TAM; c++) { // c para las columnas
                sumaFila += matriz[f][c];
            }
            if (sumaFila != patron) {
                esMagica = false;
            }
        }

        if (esMagica) { //si la variable esMagia es true
            System.out.println("La matriz es mágica");
        } else {
            System.out.println("La matriz no es mágica");
        }
    }
}
```

- 5.12. Crear una tabla bidimensional de tamaño 5×5 y rellenarla de la siguiente forma: la posición $[n, m]$ debe contener $n + m$. Después se debe mostrar su contenido.

```
/*
 * Para cargar los datos en la tabla bidimensional, utilizaremos los propios
 * índices para asignar los valores de sus elementos, de la forma t[i][j] = i + j */
public class Main {

    public static void main(String[] args) {
        int t[][]; // declaramos t como una tabla bidimensional
        t = new int[5][5]; // creamos la tabla de 5x5

        for (int i = 0; i < 5; i++) { // utilizamos i para la primera dimensión
            for (int j = 0; j < 5; j++) { // y j para la segunda dimensión
                t[i][j] = i + j;
            }
        }

        System.out.println("Tabla: ");
        for (int i = 4; i >= 0; i--) { //mostramos las filas al revés, como las matrices
            System.out.println();
            for (int j = 0; j < 5; j++) {
                System.out.print(t[i][j] + " ");
            }
        }
    }
}
```

- 5.13. Sobrecargar la función `maximo()` (ejercicios resueltos 4.4 y 4.5) utilizando una versión que calcule el máximo de una tabla de enteros.

```
/*
 * Vamos a sobrecargar de nuevo la función maximo() (ejercicios resueltos 4.4 y 4.5), pero
 * en esta ocasión, pasamos una tabla como parámetro de entrada. */
public class Main {

    static int maximo(int t[]) {
        int max = t[0]; // el primer elemento será en principio el máximo. Suponemos que
        // la tabla siempre tendrá al menos un elemento

        for (int e : t) { //recorremos la tabla para buscar un elemento mayor que max
            if (e > max) { // si e (t[i]) es mayor que max, es el nuevo máximo
                max = e;
            }
        }
        return (max);
    }

    /* un ejemplo para probar la función */
    public static void main(String[] args) {
        int numeros[] = {4, -6, 8, 9, 0, 3};
        System.out.println("\nEl número mayor es: " + maximo(numeros));
    }
}
```

- 5.14. Definir una función que tome como parámetros dos tablas, la primera con los 6 números de una apuesta de la primitiva, y la segunda con los 6 números de la combinación ganadora. La función devolverá el número de aciertos.

```
import java.util.Arrays;
/*
 * A la hora de realizar este ejercicio suponemos que:
 * - no hay números repetidos ni en la apuesta ni en la combinación ganadora
 * - los números empleados están en el rango 1..49
 * - el tamaño de las tablas será de 6 */
public class Main {

    //programa principal para comprobar la función
    public static void main(String[] args) {
        int combinacionGanadora[] = {3, 13, 25, 33, 41, 48}; //valores de prueba
        int apuesta[] = {3, 25, 41, 42, 45, 49};
        System.out.println("Aciertos: " + primitiva(combinacionGanadora, apuesta));
    }

    //devuelve el número de coincidencias entre los elementos de las tablas pasadas
    static int primitiva(int apuesta[], int premiado[]) {
        int aciertos = 0; //contador de aciertos

        for (int a : apuesta) { // recorremos la tabla de apuesta
            //aprovechamos que la tabla con la combinación está ordenada
            if (Arrays.binarySearch(premiado, a) >= 0) { //si a está en la tabla
                aciertos++; //hemos acertado un número más
            }
        }
        return (aciertos);
    }
}
```

5.15. Escribir la función `rellenaPares()` a la que se le pasa como parámetro una tabla que debe rellenar de la siguiente forma: se leerá por teclado una serie de números, guardando en la tabla los pares hasta que esté llena, e ignorando los impares. La función tiene que devolver la cantidad de impares desechados.

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * La tabla t almacenará solo números pares, mientras se ignoran los impares, que
 * tenemos que contabilizar. */
public class Main {

    public static void main(String[] args) {
        int t[] = {2, 5, 5, -3, 0}; //los valores que tuviera t se pierden
        int ignorados = rellenaPares(t); //rellenamos todos los elementos de números pares

        System.out.println("El número de impares ignorados es de: " + ignorados);
        System.out.println(Arrays.toString(t));
    }

    //lee números por teclado. Los pares los guarda en la tabla e ignora los impares
    static int rellenaPares(int pares[]) {
        Scanner sc = new Scanner(System.in);

        int i = 0; //indica con que elemento de la tabla estamos trabajando
        int imparesIgnorados = 0;

        // terminaremos de rellenar la tabla cuando el
        // número de pares sea igual que el tamaño de la tabla
        while (i < pares.length) {
            System.out.print("Introduzca número: ");
            int num = sc.nextInt(); //leemos un número
        }
    }
}
```

```
        if (num % 2 == 0) { // si es par
            pares[i] = num;    // lo guardamos. El elemento i de la tabla se pierde
            i++; //incrementamos el indicador
        } else {
            imparesIgnorados++; //si el número es impar, incrementamos el contador
        }
    }
    return (imparesIgnorados);
}
}
```

Ejercicios propuestos

- 5.1. El ayuntamiento de nuestra localidad nos ha encargado una aplicación que ayude a realizar encuestas estadísticas para conocer el nivel adquisitivo de los habitantes del municipio. Para ello, tendremos que preguntar el sueldo a cada persona encuestada. A priori no conocemos el número de encuestados. Para finalizar la entrada de datos, introduciremos un sueldo con valor -1.

Una vez terminada la entrada de datos, hemos de mostrar la siguiente información:

- Todos los sueldos introducidos ordenados de forma decreciente.
- El sueldo máximo y mínimo.
- La media de los sueldos.

Nota. Como a priori se desconoce el número de sueldos, la tabla donde se almacenan los datos tendrá que incrementar su tamaño conforme necesitemos más espacio.

- 5.2. Los diseñadores de una aplicación necesitan obtener ordenados los datos de una tabla, pero por restricciones de la aplicación, la tabla debe permanecer inmutable. Una posible solución es hacer una copia de la tabla y ordenarla, manteniendo intacta la tabla original, pero esta alternativa se ha desechado. En su lugar, se ha pensado en crear una segunda tabla donde se almacenan ordenados los índices de la tabla original. Se pide diseñar un algoritmo en el que, dada una tabla, cree otra donde se ordenen mediante los índices la tabla original.

Veamos un ejemplo:

```
tablaOriginal: [3, 5, 1, 4]
tablaConIndices: [2, 0, 3, 1]
```

Donde `tablaConIndices` especifica el lugar que ocupan de forma ordenada los datos de `tablaOriginal`. Por ejemplo, el primer elemento de `tablaOriginal`, que vale 3, en caso de ordenar los datos, ocupará la posición 2 (que le corresponde en `tablaConIndice`). En este caso, el i -ésimo elemento de `tablaOriginal` ocupará la posición que contiene el i -ésimo elemento de `tablaConIndices`.

- 5.3. Una tabla bidimensional `t` puede representar un mapa con distintos lugares (numerados de 0 a n) e indicar si existe paso del lugar i al lugar j , mediante el elemento `t[i][j]`

Veamos un ejemplo: lugar0 $\rightarrow$ lugar2 $\rightarrow$ lugar1 $\rightarrow$ lugar3

la matriz mapa será:

$$\begin{pmatrix} 0 & false & false & true & false \\ 1 & true & false & false & true \\ 2 & false & true & false & false \\ 3 & false & false & false & false \\ & 0 & 1 & 2 & 3 \end{pmatrix}$$

5.4. Implementar el juego de *wari* o *macala* para dos jugadores. El jugador que tenga el turno solo debe indicar cuál de sus hoyos utilizará para jugar. La aplicación debe hacer automáticamente el resto de las operaciones, indicando el número de piedras capturadas. Aunque el juego es encillo, sus reglas son demasiado extensas para describirlas aquí. Recomendamos al lector buscar información sobre el juego.

Capítulo 6

Cadenas

En los programas escritos hasta ahora hemos utilizado los tipos primitivos: `char`, `byte`, `int`, `float`, `double` y `boolean`. Estos bastan para implementar muchas aplicaciones, sobre todo las que trabajan con datos numéricos, pero proporcionan pocas herramientas para trabajar con texto. El tipo `char`, que almacena un solo carácter, es insuficiente para manejar textos complejos.

Por texto entendemos una palabra, una frase, e incluso uno o más párrafos de cualquier longitud. En definitiva, un texto, como por ejemplo este mismo párrafo, es una secuencia de caracteres, de ahí que también se le denomine *cadena de caracteres* o, por economía del lenguaje, simplemente *cadena*.

Para manipular textos disponemos en la API de las clases `Character` y `String` —ambas ubicadas en el paquete `java.lang`—, que proporcionan multitud de funcionalidades para trabajar con un solo carácter la primera, y con textos de cualquier longitud la segunda.

6.1. Tipo primitivo `char`

De forma general un carácter se define como una letra —de cualquier alfabeto—, un número, un ideograma o cualquier símbolo. En Java un carácter o literal carácter se escribe entre comillas simples ('). Ejemplos de ellos son: 'p', 'ç', 'Ψ', '7', '♡' o '#'.

6.1.1. *Unicode*

Mediante un teclado podemos escribir algunos caracteres, como 'a', pero otros muchos, como '♡', no se pueden escribir mediante una combinación de teclas. Para solventar este problema, un conglomerado de empresas fundó el *Unicode Consortium*, un organismo que mediante un comité técnico, diseñó y mantiene un estándar de codificación de caracteres, denominado *Unicode*. Esta codificación consiste en identificar cada carácter mediante un número¹ único, llamado *code point*, que puede representarse en decimal, con números del 0 al 65 535, o con números de 4 cifras hexadecimales (XXXX). Para evitar una posible confusión entre una u otra representación, para los números hexadecimales se utiliza la

¹Codificado en 2 bytes, lo que permite disponer de 65 536 números distintos

secuencia `\uXXXX`². A la hora de seleccionar un carácter es posible utilizar su codificación Unicode o el propio carácter, si es posible escribirlo mediante un teclado. Por ejemplo, el carácter 'a' puede ser escrito pulsando la tecla adecuada de un teclado o mediante su *code point*, en decimal (97) o en hexadecimal (`\u0061`). Veamos tres formas de asignar este valor a una variable de tipo `char`,

```
char c;  
c = 'a';  
c = 97;  
c = '\u0061';
```

Mientras que la única forma de designar el carácter '♡' es mediante su *code point*. Veamos cómo utilizar este carácter,

```
char c = '\u2661'; //o bien, c = 9825;  
System.out.println(c); //muestra un ♡
```

Para conocer la codificación de cualquier carácter necesitamos recurrir a las tablas diseñadas por el Unicode Consortium o bien a las bases de datos de caracteres —véase a modo de ejemplo la Tabla 6.1—. Estas tablas agrupan los caracteres según el alfabeto (latino, braille, etc.) o por el conjunto de símbolos al que pertenecen (matemáticos, jeroglíficos egipcios, etc.) Por ejemplo, si deseamos trabajar en Java con el ideograma ♡, lo primero que tenemos que hacer es buscar su *code point* en las tablas de caracteres, donde encontramos que se codifica mediante 9825 o `\u2661`.

Tabla 6.1. Ejemplos de codificación Unicode

Carácter	Code point	
	decimal	hexadec.
'A'	65	\u0041
'a'	97	\u0061
'Ψ'	936	\u03A8
'♡'	9825	\u2661

6.1.2. Secuencias de escape

Un carácter precedido de una barra invertida (`\`) se conoce como *secuencia de escape*. Al igual que los caracteres escritos mediante la codificación Unicode, representan un único carácter, pero en este caso poseen un significado especial. En la Tabla 6.2 se muestran las secuencias de escape de Java.

Veamos algunos ejemplos,

```
char c = '\\';  
System.out.println(c); //muestra una comilla simple: '  
c = '\"';  
System.out.println(c); //muestra una comilla doble: "  
c = '\\t'; //tabulador  
System.out.println("1" + c + "2"); //muestra "1    2";
```

²Es habitual llamar *code point* tanto al número que identifica al carácter como a la secuencia `\uXXXX`.

Tabla 6.2. Caracteres especiales

Carácter	Nombre
\b	Borrado a la izquierda
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador
\f	Nueva página
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida

6.1.3. Conversión `char`↔`int`

Cada *code point* no es más que un número codificado en 2 bytes; no existe ningún inconveniente en representarlo en decimal o en hexadecimal. La conversión de decimal a hexadecimal, o viceversa, crea una estrecha relación entre el tipo `char` y el tipo `int`. Es posible asignar un valor entero a una variable de tipo `char` (siempre y cuando el valor del entero esté comprendido entre 0 y 65 535) y asignar un carácter a una variable de tipo `int`, ya que Java se encarga de realizar las conversiones oportunas (el tipo `int` representa los números en decimal por defecto). Veamos un ejemplo: el carácter 'a' tiene asociado el *code point* \u0061. Si convertimos el número hexadecimal 0061 a decimal obtenemos,

$$0061_{16} = 97_{10}$$

Aprovechando este mecanismo de conversión podemos realizar asignaciones del tipo,

```
int e = 'a'; //asigna un carácter a una variable int
System.out.println(e); //muestra 97
e = '\u0061'; //asigna un carácter a una variable int
System.out.println(e); //muestra 97
char c = 97; //asigna un entero a una variable char
System.out.println(c); //muestra 'a'
```

También es posible forzar una conversión por medio de un `cast`,

```
char c = 'a';
System.out.println((int)c); //muestra 97
int e = 97;
System.out.println((char) e); //muestra 'a'
```

Y por supuesto, es posible realizar asignaciones entre variables de tipo `char` e `int`,

```
int e = 97;
char c = e; //c vale 'a'
```

y viceversa

```
char c = 'a';
int e = c; //e vale 97
```

6.1.4. Aritmética de caracteres

La relación existente entre un carácter y su representación numérica en Unicode, ya sea en hexadecimal o en decimal, permite realizar operaciones aritméticas con ellos. En realidad, no es posible operar con un carácter; lo que ocurre es que la operación se realiza con su representación numérica. Veamos como ejemplo la suma,

```
System.out.println('a' + 1);
```

Para poder realizar la suma, el primer paso es transformar el carácter en su representación numérica; sea en hexadecimal o en decimal, ambas representan el mismo valor. Como hemos visto: $'a' \Leftrightarrow 61_{16} = 97_{10}$. La operación quedaría,

$$'a' + 1 = 62_{16} + 1 = 97_{10} + 1 = 98_{10}$$

Es importante entender que el número obtenido (98) puede ser interpretado, a su vez, como un carácter,

```
char c = 98;
```

Si hacemos la conversión a hexadecimal,

$$98_{10} = 62_{16}$$

Si buscamos qué carácter corresponde al *code point* `'\u0062'`, resulta que es el carácter `'b'`. Resumiendo,

$$'a' + 1 = 'b'$$

mediante las conversiones oportunas a `char`.

Otra forma de entender la aritmética de caracteres consiste en que al realizar una suma o una resta, por ejemplo $'x' \pm n$, el resultado es el carácter situado n posiciones delante o detrás, en la codificación Unicode, del carácter con el que estamos operando. Veamos un ejemplo en Java,

```
char c = 'e' - 2; //vale 'c'
c = 'e' + 2; //vale 'g'
```

Es decir, la letra *e* tiene en el código Unicode, dos puestos por delante la letra *g*, y dos puestos por detrás la letra *c*.

Este comportamiento permite transformar un carácter de minúscula a mayúscula y viceversa,

```
char c = 'h' + 'A' - 'a';
System.out.println(c) //muestra 'H'
```

$'a' - 'A'$ representa la distancia que existe en el código Unicode entre las letras minúsculas y las mayúsculas.

6.2. Clase Character

El tipo `char` es a todas luces insuficiente, por sí solo, para realizar operaciones con caracteres. La clase `Character` amplía su funcionalidad para trabajar con caracteres simplificando mucho el trabajo. Pensemos en lo engorroso que puede ser, por ejemplo, decidir si un carácter dado es una letra minúscula: para ello tendríamos que realizar una serie de comprobaciones. Por el contrario, esta funcionalidad se encuentra en `Character`, que dispone de una batería de métodos de uso estático, útiles para clasificar y convertir valores de tipo `char`.

6.2.1. Clasificación de caracteres

Un carácter puede clasificarse dentro de algunos de los grupos siguientes,

- **Dígitos:** este grupo está formado por los caracteres `'0'`, `'1'`..., `'9'`.
- **Letras:** formado por todos los elementos del alfabeto³, tanto en minúscula (`'a'`, `'b'`...) como en mayúscula (`'A'`, `'B'`...).
- **Caracteres blancos:** como el espacio o el tabulador, entre otros. Estos caracteres no tienen una representación al mostrarlos en pantalla o al imprimirlos.
- **Otros caracteres:** signos de puntuación, matemáticos, etc.

Los métodos de `Character` para consultar si un carácter pertenece a alguno de estos grupos devuelven un booleano, `true` en caso de que pertenezca o `false` en caso contrario. Estos métodos son,

`boolean isDigit(char c):` indica, devolviendo `true`, si el carácter `c` es un dígito o `false` en caso contrario.

```
char c1 = '8', c2 = 'p';
boolean b;
b = Character.isDigit(c1); //b vale true, ya que '8' es un dígito
b = Character.isDigit(c2); //b es false, 'p' no es un dígito
```

`boolean isLetter(char c):` determina si el carácter pasado como parámetro es una letra, minúscula o mayúscula.

```
Character.isLetter('8'); //false: el carácter '8' no es una letra
Character.isLetter('e'); //true: el carácter 'e' sí es una letra
```

`boolean isLetterOrDigit(char c):` indica si el carácter es una letra o un dígito. Estos se conocen como un caracteres alfanuméricos.

³Utilizaremos el alfabeto latino, pero en realidad incluye las letras de cualquier alfabeto.

```
boolean b;  
b = Character.isLetterOrDigit('%'); //false: '%' no es alfanumérico  
b = Character.isLetterOrDigit('p'); //true: 'p' es una letra  
b = Character.isLetterOrDigit('2'); //true: '2' es un dígito
```

boolean isLowerCase(char c): especifica si c es una letra y además está en minúscula.

```
char c1 = 'i', c2 = 'I';  
Character.isLowerCase('*'); //false: ni siquiera es una letra  
Character.isLowerCase(c1); //true: es una letra en minúscula  
Character.isLowerCase(c2); //false: es una letra, pero no minúscula
```

boolean isUpperCase(char c): funciona igual que el método anterior, pero indicando si el carácter es una letra mayúscula.

```
boolean b;  
b = Character.isUpperCase('t'); //false  
b = Character.isUpperCase('T'); //true
```

boolean isSpaceChar(char c): devolverá true si el carácter utilizado como parámetro de entrada es un espacio (' '), que se consigue pulsando en la barra espaciadora. Para evitar problemas con el espacio en blanco, que no se ve, lo representaremos de la forma: '␣'. En la bibliografía es habitual encontrar otras representaciones, como por ejemplo una letra b tachada ('ḅ') para simbolizar un espacio en blanco.

```
char c = '␣';  
Character.isSpaceChar(c); //devuelve true  
Character.isSpaceChar('a'); //false: es obvio que no es un espacio
```

boolean isWhiteSpace(char c): amplía el método anterior y determina si el carácter pasado es cualquier tipo de blanco. Los caracteres que hacen que el método devuelva true son,

Espacio en blanco ('␣'): que se escribe mediante la barra espaciadora.

Retorno de carro ('\r'): dependiendo del sistema operativo, este carácter tendrá distinto comportamiento al imprimirse.

Nueva línea ('\n'): es el carácter que se consigue al pulsar la tecla *Intro*.

Tabulador ('\t'): que equivale a varios espacios en blanco. Se genera con la tecla *Tab*.

Otros: existen otros caracteres considerados blancos como el tabulador vertical, el separador de ficheros, etc.

Veamos un ejemplo,

```
char c = '\t';  
boolean b;  
b = Character.isWhiteSpace(c); //true: tabulador  
b = Character.isWhiteSpace('\n'); //true: carácter nueva línea  
Character.isWhiteSpace('a'); //false: evidentemente no es un blanco
```


6.2.2. Conversión

Los métodos que realizan conversiones son aquellos que devuelven transformado el valor que se le pasa como parámetro, normalmente un carácter, en otro carácter o en un valor de un tipo distinto. También existen los que realizan la operación inversa, es decir, convierten un valor de otro tipo en un carácter.

Conversión entre caracteres

Son los métodos que transforman un carácter en otro. Cuando la conversión no es posible, por ejemplo, si no se puede transformar un número a mayúscula, se devuelve el mismo carácter pasado como parámetro. Disponemos de los métodos,

char toLowerCase(char c): si el carácter pasado es una letra lo devuelve convertido a minúscula. En otro caso, devuelve el mismo.

```
char c1 = 'A', c2;
c2 = Character.toLowerCase(c1); //la variable c2 toma el valor 'a'
c2 = Character.toLowerCase('3'); //al no ser una letra, devuelve el
                                //mismo valor pasado: '3'
```

char toUpperCase(char c): muy similar al anterior método, pero convierte el carácter, si es una letra, a mayúscula. En otro caso devuelve el mismo carácter.

```
char c1 = 'g', c2;
c2 = Character.toUpperCase(c1); //a c2 se le asigna el valor 'G'
```

Conversión de carácter a otro tipo

Aquí encontramos los métodos que transforman un carácter(tipo `char`) en otro tipo distinto: `String`, `int`, etc.

int digit(char c, int base): devuelve el valor numérico de un carácter `c` en la base indicada. Si no es posible realizar la conversión devuelve `-1`.

```
int num = Character.digit('3', 10); //num vale 3
```

convierte el carácter '3' en el valor entero 3 (en base 10).

```
int num = Character.digit('c', 16); //num vale 12
```

convierte el carácter 'c' en su valor en hexadecimal: 12.

String toString(char c): devuelve una cadena de longitud uno que contiene a `c`. Con la cadena que obtenemos se puede trabajar con todas las funciones de la clase `String`, que se verán en la segunda parte de este capítulo.

```
String cad;
cad = Character.toString('b'); //cad toma el valor de la cadena "b"
```

Hay que observar que mientras los literales carácter se escriben entrecomillados con comillas simples ('), los literales cadenas utilizan comillas dobles (").

Conversión de otro tipo a carácter

`char forDigit(int digito, int base):` devuelve el carácter que representa a `digito` en la base indicada. Si no es posible realizar la conversión devuelve el carácter nulo (`'\u0000'`).

```
Character.forDigit(3, 10) //devuelve el carácter '3'  
Character.forDigit(12, 16) //devuelve el carácter 'c', que representa  
                        //el 12 en hexadecimal
```

6.3. Clase String

Las cadenas, un conjunto secuencial de caracteres, se manipulan mediante la clase **String**, que funciona de forma dual. Por un lado, de manera general, tiene un funcionamiento no estático; pero a su vez, dispone de algunos métodos que si lo son. Es posible definir variables de tipo **String** de la forma habitual,

```
String cad; //cad es una variable de tipo cadena
```

Una variable de tipo **String** almacenará una cadena de caracteres, que provendrá de la manipulación de otra cadena o de un literal. Un literal cadena consiste en un texto entrecomillado utilizando comillas dobles (`"`). Es posible utilizar cualquier carácter, incluidos los codificados mediante Unicode y las secuencias de escape. Algunos ejemplos de literal cadena son,

```
"Hola\n"  
"En un lugar de la Mancha"  
"Un corazón: \u2661"
```

Los literales carácter y cadena se diferencian en el tipo de comillas utilizado, mientras `'a'` es un carácter, `"a"` es una cadena que está compuesta por un único carácter.

Con una cadena de caracteres se pueden realizar multitud de operaciones. Algunas trabajan con la cadena como un todo y otras trabajan carácter a carácter.

6.3.1. Inicialización de cadenas

Literal cadena

De forma análoga a como lo hacemos con la clase **Scanner**, podemos utilizar **new** para crear y asignar un valor a una variable de tipo **String**,

```
cad = new String("literal cadena");
```

Lo que ocurre, es que el uso de la clase **String** es tan habitual que Java permite una forma abreviada, funcionalmente idéntica a la anterior,

```
cad = "literal cadena";
```

Ambas formas son equivalentes. Por economía en la escritura del código utilizaremos, habitualmente, la primera forma de asignación.

Una cosa a tener en cuenta es el uso de comillas (") dentro de un literal cadena, ya que es el carácter que inicia y finaliza un texto. Para ello, disponemos de la secuencia de escape \". Un ejemplo:

```
String cad = "Mi perro \"Perico\" es de color blanco";
System.out.print(cad);
```

que muestra en pantalla:

```
Mi perro "Perico" es de color blanco
```

Valores de otros tipos

A menudo necesitaremos representar un valor de un tipo primitivo en forma de cadena. Veamos un ejemplo: sea el valor entero *1234* (mil doscientos treinta y cuatro), que podemos representar como la cadena "1234", es decir, la cadena formada por el carácter '1', seguido del carácter '2', el '3' y finalizada con el carácter '4'. De igual manera podemos representar el valor de cualquier tipo primitivo. El método estático que construye una cadena para representar un valor es,

static String.valueOf (*tipo* valor): que construye y devuelve una cadena con la representación del valor pasado como parámetro. Aquí *tipo* hace referencia a cualquier tipo primitivo. En realidad, el método `valueOf()` es un método sobrecargado para cada tipo de datos primitivo. Veamos varios ejemplo,

```
String cad;
cad = String.valueOf(1234); //cad = "1234"
cad = String.valueOf(-12.34); //cad = "-12.34"
cad = String.valueOf('C'); //cad = "C"
cad = String.valueOf(false); //cad = "false"
```

6.3.2. Comparación

Los operadores de comparación disponibles para números: igual (==), menor que (<) y mayor que (>) no se encuentran disponibles directamente para comparar cadenas de caracteres, pero en su lugar disponemos de métodos de la clase **String** que realizan las comparaciones oportunas de forma similar a como se hacen con números.

Igualdad

Un error común es comparar dos variables de tipo cadena utilizando el operador habitual de comparación (==). Este operador no se puede utilizar con **String** debido a que es una clase y no un tipo primitivo. Por ello, para comparar cadenas utilizaremos,

boolean equals(String otraCadena): compara el contenido de la cadena que invoca el método y de *otraCadena*. El resultado de la comparación se indica devolviendo **true** o **false**, según sean iguales o distintas. Para que las cadenas se consideren iguales deben estar formadas por la misma secuencia de caracteres. Veamos un ejemplo,

```
String cad1 = "Hola mundo";
String cad2 = "Hola mundo";
String cad3 = "Hola, buenos días"
boolean iguales;
iguales = cad1.equals(cad2); //iguales vale true
iguales = cad1.equals(cad3); //iguales vale false
```

Puede ocurrir que nos interese realizar una comparación, pero sin tener en cuenta las letras mayúsculas y minúsculas, que `equals()` considera distintas. Es decir, poder comparar la cadena «Hola» con «hOLA» y que resulten iguales. Para ello, existe el método,

`boolean equalsIgnoreCase(String otraCadena)`: funciona igual que `equals()` pero sin distinguir mayúscula de minúsculas al realizar la comparación. Veamos un ejemplo,

```
String cad1 = "Hola mundo";
String cad2 = "HOLA Mundo";
boolean iguales;
iguales = cad1.equals(cad2); //false, no son iguales
iguales = cad1.equalsIgnoreCase(cad2); //true, sin atender a
//mayúsculas/minúsculas son iguales
```

Los métodos vistos comparan la totalidad de dos cadenas, pero es posible comparar solo una región, o fragmento, de cada cadena. Para ello disponemos de,

`boolean regionMatches(int inicio, String otraCad, int inicioOtra, int num)`: compara dos fragmentos de cadenas: el primero corresponde a la cadena invocante y comienza en el carácter con índice `inicio`; y el segundo corresponde a la cadena `otraCad` y comienza en el carácter con índice `inicioOtra`. Ambos fragmentos tienen una longitud de `num` caracteres. El método devuelve `true` o `false` para indicar si las regiones coinciden. Por ejemplo,

```
boolean b;
String cad = "Mi_perro_ladra_mucho";
String otra = "Un_bonito_perro_blanco";
b = cad.regionMatches(3, otra, 10, 5) //cierto
```

compara las regiones **"perro"** (comienza en el índice 3) de `cad` y **"perro"** (comienza en el índice 10) de `otra`. Ambas regiones tienen un tamaño de 5 caracteres.

`boolean regionMatches(boolean ignora, int ini, String otraCad, int iniOtra, int num)`: hace lo mismo que el método anterior con la diferencia de que, si el valor del parámetro `ignora` es `true`, la comprobación se realiza considerando iguales las mayúsculas y minúsculas.

Comparación alfabética

Otra forma de comparar dos cadenas es hacerlo de forma alfabética, es decir, saber qué cadena iría antes en un diccionario. Una cadena se considera alfabéticamente menor

que otra si en un diccionario va antes. El orden lo marca la posición de las letras en el alfabeto⁴ (las letras mayúsculas se consideran anteriores a las minúsculas). La comparación se lleva a cabo mirando el primer carácter distinto de cada cadena; si por ejemplo comparo "monitor" y "monzón", la comparación se realiza mirando el cuarto carácter, con índice 3, de cada cadena, que es el primer carácter distinto.

Los métodos disponibles para comparar alfabéticamente cadenas son,

int compareTo(String cadena): que compara alfabéticamente la cadena invocante y la que se pasa como parámetro, devolviendo un entero cuyo valor determina el orden de las cadenas de la forma,

0: si las cadenas comparadas son exactamente iguales.

negativo: si la cadena invocante es menor alfabéticamente que la cadena pasada como parámetro, es decir, va antes por orden alfabético.

positivo: si la cadena invocante es mayor alfabeticamente que la cadena pasada, es decir, va después.

Hay que recordar que las letras mayúsculas preceden, se consideran menores, que la minúsculas.

```
String cad1 = "Alondra";
String cad2 = "Nutria";
String cad3 = "Zorro";
System.out.println(cad2.compareTo(cad1)) //valor mayor que 0
//"Nutria" está después que "Alondra" alfabéticamente
System.out.println(cad2.compareTo(cad3)) //valor menor que 0
//"Nutria" está antes que "Zorro" alfabéticamente
```

int compareToIgnoreCase(String cadena): realiza una comparación alfabética sin distinguir entre letras mayúsculas ni minúsculas.

6.3.3. Concatenación

El operador + sirve para unir, o concatenar, dos cadenas. Veamos su funcionamiento con un ejemplo:

```
String nombre = "Miguel";
String apellidos = "de_Cervantes_Saavedra";
String nombreCompleto = nombre + apellidos;
System.out.println(nombreCompleto); //"Miguelde Cervantes Saavedra"
```

La concatenación une dos cadenas, pero no inserta nada entre ellas. En nuestro ejemplo, el nombre está completamente pegado a los apellidos. Esto puede evitarse haciendo,

```
String nombreCompleto = nombre + " " + apellidos;
System.out.println(nombreCompleto); //"Miguel de Cervantes Saavedra"
```

⁴Esta es una visión más que suficiente por ahora, pero realmente el orden lo marca la posición de los caracteres en la codificación Unicode.

La conversión de datos de tipo primitivo a tipo `String` permite concatenarlos a una cadena. Esta conversión la realiza Java automáticamente, y de forma transparente al programador, mediante el uso del método `valueOf()` de `String`. Veamos algunos ejemplos,

```
String a, b, c;
a = "Resultado: " + 3; //equivale a "Resultado: " + String.valueOf(3)
b = "Resultado: " + true; //equivale a "Resultado: " + String.valueOf(true)
c = "Resultado: " + 'a'; //equivale a "Resultado: " + String.valueOf('a')
```

Otra forma de concatenar cadenas, idéntico al operador `+`, es mediante el método,

`String concat(String cad)`: crea y devuelve una nueva cadena con la concatenación de ambas, es decir, la cadena invocante y `cad`. Este método funciona sin modificar ni la cadena invocante ni la que se le pasa como parámetro. Veamos un ejemplo,

```
nombre = "Miguel";
apellidos = "deCervantesSaavedra";
nombreCompleto = nombre.concat(apellidos);
//es equivalente a: nombreCompleto = nombre + apellidos;
```

`concat()` devuelve la concatenación, que se asigna a `nombreCompleto`, pero no modifica ni la variable `nombre` ni `apellidos`.

6.3.4. Obtención de caracteres

Todos los caracteres que forman una cadena pueden ser identificados mediante la posición que ocupan, al igual que los elementos de una tabla. Cada carácter se numera con un índice único que comienza en 0. Veamos un ejemplo donde se aprecia una cadena junto a los índices que identifican a cada carácter,

"	L	l	a	m	a	d	m	e		I	s	m	a	e	l	"
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		

En la cadena anterior, en la posición 2 encontramos el carácter 'a', en la posición 9 el carácter 'I' y en la posición 12, de nuevo, otro carácter 'a'.

Cuando hablamos de extraer uno o varios caracteres de una cadena nos referimos a obtener una copia del carácter o caracteres en cuestión, pero la cadena de la que extremos se mantiene intacta.

Obtención de un carácter

Para conocer qué carácter se encuentra en una posición dada de una cadena disponemos de:

`char charAt(int posicion)`: que devuelve el carácter que ocupa el índice `posicion` en la cadena que invoca el método. Hay que tener mucha precaución con no utilizar una posición que se encuentre fuera de rango, ya que esto provocará un error y la terminación abrupta del programa. Veamos un ejemplo:

```
String frase = "Nació con el don de la risa";
System.out.println(frase.charAt(4)); //muestra el carácter 'ó'
char c = frase.charAt(30); //¡error! No existe la posición 30
```

Obtención de una subcadena

Una subcadena es un fragmento de una cadena, es decir, un subconjunto consecutivo de caracteres que forman parte de una cadena. En ocasiones puede ser interesante extraer de una cadena un fragmento. Por ejemplo, si tenemos el nombre y los apellidos de alguien, puede ser interesante extraer solo los apellidos. Los métodos que llevan a cabo esto son,

String substring(int inicio): devuelve la subcadena formada desde la posición *inicio* hasta el final de la cadena. Lo que se devuelve es una copia y la cadena invocante no se modifica.

```
String cad1 = "Una mañana, al despertar de un sueño intranquilo";
String cad2 = cad1.substring(28); //cad2 vale "un sueño intranquilo"
```

String substring(int inicio, int fin): hace lo mismo que la anterior, devolviendo la subcadena comprendida entre los índices *inicio* y el anterior a *fin*⁵.

```
String cad1 = "Una mañana, al despertar de un sueño intranquilo";
String cad2 = cad1.substring(15, 36); //cad2 = "despertar de un sueño"
```

Hay que notar que en *cad1* el carácter que ocupa el índice 36 es el espacio en blanco que va justo antes de *intranquilo* y que este carácter no forma parte de la subcadena devuelta. Esta se forma con los caracteres que se encuentran desde el índice 18 hasta el carácter anterior al 36, es decir, el 35.

En ambos métodos ocurre que si utilizamos un índice que se encuentra fuera de rango, es decir, no corresponde a ningún carácter, se produce un error que termina la ejecución del programa.

Es habitual que una cadena leída del teclado o de algún fichero, venga acompañada de una serie de espacios en blanco (' ') y tabuladores ('\t' que representaremos '____') al comienzo y/o al final de la cadena. Antes de trabajar con la cadena tenemos que procesarla y eliminar estos caracteres *blancos*. Por suerte, esta funcionalidad está implementada en,

String trim(): que devuelve una copia de la cadena eliminando, al principio y al final, los caracteres *blancos*. La cadena invocante no se modifica.

```
String cad1 = "____Mi_perro_se_llama_Perico____", cad2;
cad2 = cad1.trim(); //cad2 vale "Mi perro se llama Perico"
```

6.3.5. Longitud de una cadena

Como hemos visto, en ciertos métodos es necesario utilizar algunos índices para localizar los caracteres que forman una cadena. Nos encontramos con el problema de que si los índices utilizados no corresponden a ningún carácter obtenemos un error. Cualquier índice negativo o cualquier valor mayor que el índice del último carácter generan este error. Para evitar una posición que se encuentre fuera de rango, existe,

⁵Es una norma general en Java que, cuando se especifica un rango de índices, mediante *inicio* y *fin*, el índice *inicio* esté incluido en el rango y el de *fin*, excluido. La razón es que así, la longitud del rango puede calcularse restando, *fin-inicio*.

int length(): que devuelve el número de caracteres, o longitud, de una cadena. Una vez que conocemos la longitud, podemos utilizar, sin miedo a generar un error, cualquier índice comprendido entre 0 y el índice del último carácter, que es la longitud de la cadena menos 1.

```
int longitud;
String cad1 = "Hola" , cad2 = "";
longitud = cad1.length(); //devuelve 4
longitud = cad1.length(); //devuelve 0
```

6.3.6. Búsqueda

Dentro de una cadena, entre los caracteres que la forman, es posible buscar un carácter o una subcadena. Disponemos de métodos que realizan la búsqueda de izquierda a derecha, o en sentido contrario, o buscan a partir de una posición dada. En cualquier caso, los métodos de búsqueda devuelven el índice donde se ha encontrado lo que se buscaba, o un `-1` en caso contrario.

int indexOf(int c): busca la primera ocurrencia del carácter `c` en la cadena invocante. Si lo encuentra, devuelve su índice, o un `-1` en caso contrario. Obsérvese que el carácter se pasa como un entero, esto no supone ningún problema, gracias a la conversión automática entre el tipo `char` e `int`.

int indexOf(String cadena): es un método sobrecargado que sirve para buscar la primera ocurrencia de una cadena. Veamos un ejemplo,

```
int pos;
String cad = "Mi_perro_se_llama_Perico";
pos = cad.indexOf('j'); //pos vale -1, no encontramos 'j' en cad
pos = cad.indexOf('e'); //pos vale 4, el índice de la primera 'e'

pos = cad.indexOf("hola"); //pos vale -1, no se encuentra "hola"
pos = cad.indexOf("perro"); //pos vale 3
```

Los métodos anteriores buscan desde el comienzo de la cadena, comenzando en la posición 0 y avanzando, pero es posible comenzar la búsqueda en otra posición que no sea la inicial. Para esto disponemos de,

int indexOf(int c, int inicio): busca la primera ocurrencia del carácter `c`, pero en lugar de comenzar a buscar en la posición 0, lo hacen desde la posición `inicio` en adelante. Devuelve el índice del elemento buscado si lo encuentra o `-1` en caso contrario.

int indexOf(String cadena, int inicio): buscan la primera ocurrencia a partir de la posición `inicio` de `cadena`. Un ejemplo,

```
int pos;
String cad = "Mi_perro_pequines_se_llama_perico";
pos = cad.indexOf("pe"); //devuelve 3
```



```
pos = cad.indexOf("pe", 4); //devuelve 9, la posición del primer
                        //"pe" a partir del índice 4

pos = cad.indexOf('s'); //devuelve 16
pos = cad.indexOf('s', 25); //devuelve -1, a partir de la
                        //posición 25 no se encuentra 's'
```

Los métodos anteriores realizan la búsqueda de izquierda a derecha, en el sentido de la escritura, pero es posible realizar la búsqueda en sentido contrario, es decir, empezando por el final. Los métodos que hacen esto son,

int lastIndexOf(int c): devuelve el índice de la última ocurrencia de **c** o **-1**, en el caso de que no se encuentre.

int lastIndexOf(String cadena): funciona igual que el anterior, pero buscando la última ocurrencia de **cadena**. Un ejemplo,

```
int pos;
String cad = "Mi_perro_pequines_se_llama_perico";
pos = cad.lastIndexOf('s'); //devuelve 18. Busca 's' desde el final
pos = cad.lastIndexOf("pe"); //devuelve 27
```

El método **lastIndexOf()** está sobrecargado para buscar a partir de una posición. En este caso, la búsqueda no comienza por el final de la cadena; lo hace desde la posición indicada, buscando de derecha a izquierda.

int lastIndexOf(int c, int inicio): devuelve la última ocurrencia de **c**. La búsqueda se realiza desde el final al inicio de la cadena, comenzando en la posición **inicio**,

int lastIndexOf(String cadena, int inicio): devuelve la posición de **cadena** en la cadena invocante, comenzando en el índice **inicio** y buscando desde el final hacia el principio de la cadena. En caso de no encontrar nada, devuelve **-1**.

6.3.7. Comprobaciones

Con una cadena de caracteres es posible realizar ciertas comprobaciones, como por ejemplo si está vacía, si contiene cierta subcadena, si comienza con un determinado prefijo o si termina con un sufijo dado, entre otras. Por regla general, los métodos que realizan estas comprobaciones devuelven un booleano que indica el éxito o el fracaso de la consulta.

Cadena vacía

Una cadena vacía es aquella que no está formada por ningún carácter, y se representa mediante **""** (comillas dobles seguido de otras comillas dobles). Otra forma de definir una cadena vacía es aquella que está formada por 0 caracteres, es decir, su longitud es 0. Veamos cómo asignar la cadena vacía a una variable,

```
String cad = "";
```

Si mostramos una cadena vacía no aparecerá nada en pantalla. Una operación frecuente es inicializar una variable con la cadena vacía para ir concatenándole otras cadenas. El método para comprobar si una variable contiene la cadena vacía es,

boolean isEmpty(): indica mediante un booleano, **true**, si la cadena está vacía, o **false** en caso contrario. Un ejemplo,

```
boolean b;  
String cad1 = "", cad2 = "Hola...";  
b = cad1.isEmpty(); //true  
b = cad2.isEmpty(); //false
```

Contiene

Si necesitamos comprobar si una cadena contiene, en cualquier posición, otra subcadena, disponemos del método,

boolean contains(CharSequence subcadena): que devuelve **true** si en la cadena invocante se encuentra, literalmente, **subcadena** en cualquier posición. Hay que notar que el parámetro de entrada que se le pasa al método es un objeto de la clase **CharSequence**, pero no tenemos que preocuparnos, ya que Java realiza una conversión automática desde la clase **String**, pudiendo utilizar la función directamente con cadenas. Veamos un ejemplo,

```
String frase = "En un lugar de la Mancha";  
String palabra = "lugar";  
System.out.println(frase.contains(palabra)); //muestra true  
System.out.println(frase.contains("silla")); //muestra false
```

Prefijos y sufijos

Un prefijo o un sufijo no es más que una subcadena que va al principio o al final de una cadena. Un ejemplo de prefijo en la palabra *descongelar* es *des*. En las cadenas de caracteres podemos comprobar si comienzan o terminan con un prefijo o sufijo dado. Para ello disponemos de los métodos,

boolean startsWith(String prefijo): comprueba si la cadena que invoca el método comienza con la cadena **prefijo** que se pasa como parámetro.

```
String cad = "Hola mundo...";  
boolean b = cad.startsWith("Hol"); //true, cad comienza por "Hol"  
b = cad.startsWith("mun"); //false, cad no comienza por "mun"
```

boolean startsWith(String prefijo, int inicio): hace lo mismo que el método anterior, comenzando la comprobación en la posición **inicio**. Dicho de otra forma, para realizar la comprobación ignora los caracteres desde el principio de la cadena hasta una posición anterior a **inicio**.

```
String cad = "Hola mundo...", p = "Hol", s = "mun";
b = cad.startsWith(p, 5); //false, "Hola mundo..." no empieza por "Hol"
b = cad.startsWith(s, 5); //true, "Hola mundo..." comienza por "mun"
```

boolean endsWith(String sufijo): indica si la cadena termina con el sufijo que le pasamos como parámetro.

```
String cad = "Hola mundo";
b = cad.endsWith("De"); //falso, evidentemente cad no termina en "De"
b = cad.endsWith("undo"); //cierto, cad finaliza con "undo"
```

6.3.8. Conversión

Una cadena puede transformarse sustituyendo todas las letras que la componen a minúscula o a mayúscula, lo que resulta útil a la hora de procesar, por ejemplo, valores que provienen de un formulario y que cada usuario puede escribir de una forma u otra. Por homogeneidad se suele trabajar con todos los valores convertidos a un solo tipo de letra. Para realizar esta operación disponemos de,

String toLowerCase(): que devuelve una copia de la cadena donde se han sustituido todas las letras por minúsculas.

String toUpperCase(): similar al método `toLowerCase()`, convirtiendo todas las letras a mayúsculas. Veamos un ejemplo de los dos métodos,

```
String cad1 = "Mi PeRrO: sE lLaMa PeRiCo23.";
String cad2;
cad2 = cad.toLowerCase(); // "mi perro: se llama perico23."
cad2 = cad.toUpperCase(); // "MI PERRO: SE LLAMA PERICO23."
```

Hay que destacar que solo se convierten las letras; el resto de caracteres se mantiene igual.

El método `replace()` permite convertir todas las ocurrencias de un carácter de una cadena en otro carácter distinto. Veamos la sintaxis,

String replace(char car, char otro): devuelve una copia de la cadena invocante donde se han sustituido todas las ocurrencia del carácter `car` por `otro`. Un ejemplo,

```
String cad = "Hola mundo";
cad = cad.replace('o', '\u2661') // "H\u2661la mund\u2661"
//recordamos que el code point 2661 identifica el carácter \u2661
```

6.4. Cadenas y tablas de caracteres

Existe una innegable relación entre las cadenas, clase `String`, y las tablas de caracteres, `char[]`, hasta el punto de que, en algunos lenguajes de programación no existe el tipo

cadena, sino tablas de caracteres. En Java, ambas, cadenas y tablas de caracteres, pueden convertirse sin problema unas en otras. En aquellas ocasiones en que interese manipular o cambiar de lugar los caracteres dentro de una cadena, resulta más cómodo trabajar con una tabla, cuyo acceso a los elementos es directo. Además, las cadenas en Java no se pueden modificar una vez creadas. Cuando modificamos una cadena lo que ocurre es que se crea una cadena distinta donde se incluyen las modificaciones. Afortunadamente, este proceso es transparente al programador.

Tabla 6.3. Ejemplo relación de String-char[]

Tipo	Descripción	Ejemplo										
String	Cadena de caracteres	"Hola mundo"										
char []	Tabla de caracteres	<table><tr><td>'H'</td><td>'o'</td><td>'l'</td><td>'a'</td><td>' '</td><td>'m'</td><td>'u'</td><td>'n'</td><td>'d'</td><td>'o'</td></tr></table>	'H'	'o'	'l'	'a'	' '	'm'	'u'	'n'	'd'	'o'
'H'	'o'	'l'	'a'	' '	'm'	'u'	'n'	'd'	'o'			

El método que crea una tabla de caracteres tomando como base una cadena es,

char[] toCharArray(): crea y devuelve una tabla de caracteres con el contenido de la cadena desde la que se invoca, a razón de un carácter en cada elemento.

```
String cad = "Hola_mundo";
char c[];
c = cad.toCharArray();
//la tabla c vale ['H','o','l','a',' ','m','u','n','d','o']
```

El método que realiza el proceso inverso, crear una cadena tomando como base una tabla de caracteres, es:

static String valueOf(char[] tabla): que devuelve un String con el contenido de la tabla de caracteres. Un ejemplo:

```
String cad;
char c[] = {'H', 'o', 'l', 'a'};
cad = String.valueOf(c); //cad vale "Hola"
```

En ocasiones, puede ser interesante obtener una cadena, pero no de una tabla de caracteres al completo, sino de un subconjunto de elementos de la tabla. Al siguiente método se le pasa la posición del primer índice que nos interesa y el número de caracteres que queremos utilizar,

static String valueOf(char[] t, int inicio, int numero): funciona de forma similar al método anterior, con la diferencia de que devuelve la cadena formada por un subconjunto de los caracteres de la tabla **t**. El parámetro **inicio** es el índice del primer elemento de la tabla que nos interesa y **numero** determina el número de caracteres que compondrán la cadena. Veamos cómo funciona:

```
String cad;
char c[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
cad = String.valueOf(c, 2, 4); //cad vale "cdef"
```

Para finalizar con los métodos que trabajan con cadenas y tablas de caracteres, veremos `getChars()` que copia una subcadena en algunos elementos consecutivos de una tabla de caracteres.

void getChars(int iniCad, int finCad, char t[], int iniT): que copia la subcadena comprendida entre el carácter con índice `iniCad` y el carácter anterior al índice `finCad` de la cadena invocante, en la posición `iniT` de la tabla `t`, a razón de un carácter en cada elemento de la tabla. Los elementos de la tabla donde se copia la subcadena son machacados por los nuevos caracteres. La tabla debe estar creada previamente y disponer del tamaño suficiente para albergar los caracteres a copiar. En caso contrario se producirá un error.

```
char t[] = new char[10];
for (int i = 0; i < t.length; i++) { //c = ['0', '1', '2'..., '9']
    t[i] = Character.forDigit(i, 10);
}
String cad = "En un lugar de la Mancha";
cad.getChars(3, 5, t, 4); //copia "un" a partir del índice 4 de t
//c = ['0', '1', '2', '3', 'u', 'n', '6', '7', '8', '9']
```

Ejercicios de cadenas

6.1. Introducir por teclado dos palabras e indicar cuál de ellas es la más corta, es decir, la que contiene menos caracteres.

```
import java.util.Scanner;

/*
 * Leeremos dos cadenas (String), y compararemos sus longitudes.
 * Para obtener el tamaño utilizamos length().
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String palabra1, palabra2;
        int longPal1, longPal2;

        // leemos las dos palabras
        System.out.println("Primera palabra:");
        palabra1 = sc.nextLine();
        System.out.println("Segunda palabra:");
        palabra2 = sc.nextLine();

        // calculamos la longitud de cada palabra
        longPal1 = palabra1.length();
        longPal2 = palabra2.length();
        // comparamos los tamaños
        if (longPal1 == longPal2) {
            System.out.println("Son de idéntica longitud");
        }
    }
}
```

```
    } else if (longPal1 < longPal2) {
        System.out.println(palabra1 + " es más corta que " + palabra2);
    } else {
        System.out.println(palabra2 + " es más corta que " + palabra1);
    }
}
}
```

- 6.2. Diseñar el juego *acierta la contraseña*. La mecánica del juego es la siguiente: el primer jugador introduce la contraseña; a continuación, el segundo jugador debe teclear palabras hasta que la acierte. Realizar dos versiones; en la primera las únicas pistas que se proporcionan son el número de caracteres y cuáles son el primer y el último carácter de la contraseña. En la segunda versión se facilita el juego indicando si la palabra introducida es mayor o menor, alfabéticamente, que la contraseña.

Solución a)

```
import java.util.Scanner;
/*
 * Las cadenas no se pueden comparar utilizando el operador ==
 * Para realizar comparaciones de cadenas disponemos de equals() y otros métodos
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String passwd, palabra;

        System.out.print("Jugador 1. Introduzca la contraseña: ");
        passwd = sc.nextLine(); //leemos la contraseña
        int l = passwd.length(); //calculamos la longitud

        /*suponemos que passwd no está vacía. Si lo estuviese se producirá un error,
        ya que no existen los caracteres con índice 0 ni l-1, ni ningún otro:*/
        char primer = passwd.charAt(0);
        char ultimo = passwd.charAt(l - 1);

        System.out.println("Pistas\nLongitud: " + l); //mostramos las pistas
        System.out.println("Primer carácter: " + primer);
        System.out.println("Último carácter: " + ultimo);

        do {
            System.out.print("Jugador 2. Palabra: ");
            palabra = sc.nextLine(); //leemos una palabra
        } while (!passwd.equals(palabra)); //mientras no sean iguales seguimos jugando

        System.out.println("¡Acertaste!"); //salir de while significa acertar
    }
}
```

Solución b)

```
import java.util.Scanner;
/*
 * El método equals() nos dice si dos cadenas son iguales y el método compareTo()
 * especifica qué cadena es mayor o menor, alfabéticamente, que la otra.
 */
public class Main {
```

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String passwd, palabra;

    System.out.print("Jugador 1. Introduzca la contraseña: ");
    passwd = sc.nextLine(); //leemos la contraseña

    do {
        System.out.print("Jugador 2. Palabra: ");
        palabra = sc.nextLine();

        int comparacion = passwd.compareTo(palabra); //comparamos alfabéticamente
        if (comparacion == 0) {
            System.out.println("¡Acertaste!"); //son iguales
        } else if (comparacion < 0) {
            System.out.println("La contraseña es menor que: " + palabra);
        } else {
            System.out.println("La contraseña es mayor que: " + palabra);
        }
    } while (!passwd.equals(palabra));
}
}

```

6.3. Introducir por teclado una frase palabra a palabra, y mostrar la frase completa separando las palabras introducidas con espacios en blanco. Terminar de leer la frase cuando alguna de las palabras introducidas sea la cadena «fin» escrita con cualquier combinación de mayúsculas/minúsculas. La cadena «fin» no aparecerá en la frase final.

```

import java.util.Scanner;
/*
 * Vamos a leer una serie de palabras que iremos concatenando. Hay que comprobar cada
 * palabra leída por si coincide con alguna combinación de mayúsculas/minúsculas de
 * la cadena "fin"
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase = "", palabra; //frase debe inicializarse con la cadena vacía
        //ya que vamos a concatenarle otra cadena.
        //leemos la primera palabra fuera del bucle por si es "fin"
        System.out.print("Escriba una palabra: ");
        palabra = sc.nextLine();
        while (!palabra.toLowerCase().equals("fin")) {

            frase = frase + " " + palabra; //concatenamos la palabra al final de la
            //frase, con un espacio en blanco. La primera vez, frase está
            //inicializada con la cadena vacía. Si no, produciría un error.
            System.out.print("Escriba una palabra: ");
            palabra = sc.nextLine();
        }
        //Sea cual sea la combinación de mayúsculas/minúsculas de palabra, la
        //convertimos a minúscula y comparamos con "fin". Se podría convertir a
        //mayúsculas y comparar con "FIN"
        System.out.println(frase); //mostramos el resultado
    }
}

```

6.4. Diseñar una aplicación que pida al usuario que introduzca una frase por teclado e indique cuántos espacios en blanco tiene.

Solución a)

```
import java.util.Scanner;
/*
 * Vamos a recorrer la cadena introducida por el usuario, comprobando carácter a
 * carácter si coincide con un espacio en blanco. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase;
        int numEspaciosBlanco = 0; //contador del número de espacios en blanco
        char c;
        System.out.print("Escriba una frase: ");
        frase = sc.nextLine();

        for (int i = 0; i < frase.length(); i++) { //recorremos del índice 0 a longitud-1
            c = frase.charAt(i); //vemos cual es el i-ésimo carácter
            if (Character.isSpaceChar(c)) { //es equivalente a: c == ' '
                numEspaciosBlanco++; //incrementamos
            }
        }
        System.out.println("Tiene: " + numEspaciosBlanco + " espacios en blanco");
    }
}
```

Solución b)

```
import java.util.Scanner;
/*
 * Vamos a convertir la cadena en una tabla de caracteres, que recorreremos para
 * comprobar si cada elemento (carácter) de la tabla es un espacio en blanco. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase;
        char t[]; //tabla en la que copiaremos los caracteres de frase
        int numEspaciosBlanco = 0; //contador del número de espacios en blanco

        System.out.print("Escriba una palabra: ");
        frase = sc.nextLine();
        t = frase.toCharArray(); //convertimos de cadena a tabla de caracteres
        for (int i = 0; i < t.length; i++) { //recorremos la tabla
            if (Character.isSpaceChar(t[i])) { //comprobamos el i-ésimo carácter de t
                numEspaciosBlanco++; //incrementamos
            }
        }
        System.out.println("Tiene: " + numEspaciosBlanco + " espacios en blanco");
    }
}
```

- 6.5.** Pedir el nombre completo (nombre y apellidos) al usuario. El programa debe eliminar cualquier vocal del nombre. Por ejemplo, “Álvaro Pérez” se mostrará “lvr Prz”. Solo se eliminan las vocales (mayúsculas, minúsculas y acentuadas). El resto de caracteres no se modifican.

```
import java.util.Scanner;
/*
 * La idea es recorrer el nombre, carácter a carácter, comprobando si es una vocal. En
```



```

* El caso de que no sea una vocal concatenaremos el carácter al final de una segunda
* cadena, que llamaremos sinVocales. Para comprobar si un carácter es una vocal
* crearemos la función: esVocal() que nos indicará si un carácter es una vocal. */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String nombre, sinVocales = "";
        char c;
        System.out.print("Escriba su nombre: ");
        nombre = sc.nextLine();

        for (int i = 0; i < nombre.length(); i++) {//recorremos la cadena
            c = nombre.charAt(i);

            if (!esVocal(c)) {
                sinVocales = sinVocales + c;
            }
        }
        System.out.println(sinVocales);
    }

    static boolean esVocal(char c) {
        boolean result; //resultado de la comprobación
        String vocales = "aeiouáéíóúü"; //cadena con todas las vocales posibles
        //en minúsculas
        c = Character.toLowerCase(c); //convertimos c en minúsculas
        if (vocales.indexOf(c) == -1) { //buscamos c en la cadena vocales
            result = false; //si no se encuentra es que no es una vocal
        } else {
            result = true; //en caso contrario: es una vocal
        }
        return result;
    }
}

```

6.6. Diseñar una función a la que se le pase una cadena de caracteres y la devuelva invertida. Un ejemplo, la cadena "Hola mundo" quedaría "odnum aloH"

```

import java.util.Scanner;
/*
* Vamos a crear una función a la que se le pasa una cadena y la devuelve invertida.
*/

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String antes, despues;

        System.out.print("Escriba una cadena: ");
        antes = sc.nextLine();

        despues = alReves(antes); //utilizamos la función
        System.out.println(despues); //mostramos
    }

    // Vamos a recorrer la cadena original en el sentido de la escritura, es decir, de
    //izquierda a derecha. Cada carácter se concatenará en la cadena nueva, con la pecu-
    //liaridad de que se concatena primero el carácter. Con lo que conseguimos invertir.
    static String alReves(String original) {
        String nueva = "";
        char t[] = original.toCharArray();
    }
}

```

```
        for (int i = 0; i < t.length; i++) {
            nueva = t[i] + nueva; //concatenamos el carácter antes que nueva
        }
        return nueva;
    }
}
```

- 6.7. Diseñar un programa que solicite al usuario una frase y una palabra. A continuación buscará cuántas veces aparece la palabra en la frase.

```
import java.util.Scanner;
/*
 * Buscamos la palabra en la frase usando la función indexOf(), una vez que encontremos
 * la primera ocurrencia (en el índice pos) seguiremos buscando, a partir de pos, por si
 * existe otra ocurrencia. Y así sucesivamente hasta que no encontremos más (cuando
 * pos sea -1).
 */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase, palabra;
        int veces = 0, pos; //variables contador y posición

        System.out.print("Introduzca una frase: ");
        frase = sc.nextLine();
        System.out.print("Introduzca una palabra: ");
        palabra = sc.nextLine();

        pos = frase.indexOf(palabra); //buscamos la primera ocurrencia
        while (pos != -1) { //mientras pos no sea -1: hemos encontrado la palabra
            veces++; //si hemos encontrado una ocurrencia, incrementamos veces
            pos = frase.indexOf(palabra, pos + 1); //volvemos a buscar a partir de la
            //posición siguiente a pos, por si encontramos otra ocurrencia de palabra
        }
        //cuando salimos del bucle es que ya no existen más ocurrencias

        if (veces == 0) { //no hemos encontrado la palabra en la frase
            System.out.println("\n" + palabra + "\" no se encuentra en la frase");
        } else {
            System.out.println("\n" + palabra + "\" está " + veces + " veces");
        }
    }
}
```

- 6.8. Realizar un programa que lea una frase del teclado y nos indique si es palíndroma, es decir, que la frase sea igual leyendo de izquierda a derecha, que de derecha a izquierda, sin tener en cuenta los espacios. Un ejemplo de frase palíndroma es: *Dábale arroz a la zorra el abad*.

Las vocales con tilde hacen que un algoritmo tome una frase palíndroma como si no lo fuese. Por esto, supondremos que el usuario introduce la frase sin tildes.

```
import java.util.Scanner;
/*
 * La frase "Dabale arroz a la zorra el abad" es palíndroma si no tenemos encuentra los
 * espacios en blanco. Por lo tanto, lo primero que tenemos que hacer, es eliminarlos.
 * A continuación, vamos a construir la frase invertida. Si ambas, original e invertida,
```

```

* coinciden es porque la frase original es palíndroma.
* Nota: escribiremos las frases sin vocales acentuadas. */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase, sinEspacios, invertida;

        System.out.print("Introduzca una frase: ");
        frase = sc.nextLine();
        sinEspacios = eliminaEspacios(frase); //la función devuelve una cadena sin espacios
        invertida = alReves(sinEspacios); //reutilizamos la función escrita en el Ej 6.

        if (sinEspacios.equals(invertida)) {
            System.out.println("La frase es palíndroma");
        } else {
            System.out.println("La frase no es palíndroma");
        }
    }

    //La función construye y devuelve una cadena idéntica a la pasada, con la diferencia
    //que se han eliminado todos los espacios en blanco
    static String eliminaEspacios(String cadena) {
        String sin = "";

        for (int i = 0; i < cadena.length(); i++) { //recorremos la cadena
            char c = cadena.charAt(i); //miramos el carácter en la i-ésima posición

            if (!Character.isWhitespace(c)) { //si no es un carácter blanco
                sin = sin + c; //construimos la cadena sin con c (que no es un blanco)
            }

        }
        return sin;
    }

    static String alReves(String original) {
        String nueva = "";
        char t[] = original.toCharArray();

        for (int i = 0; i < t.length; i++) {
            nueva = t[i] + nueva; //concatenamos el carácter antes que nueva
        }
        return nueva;
    }
}

```

6.9. Los habitantes de Javalandia tienen un idioma algo extraño; cuando hablan siempre comienzan sus frases con “Javalín, javalón”, para después hacer una pausa más o menos larga (la pausa se representa mediante espacios en blanco o tabuladores) y a continuación expresan el mensaje. Existe un dialecto que no comienza sus frases con la muletilla anterior, pero siempre las terminan con un silencio, más o menos prolongado y la coletilla “javalén, len, len”. Se pide diseñar un traductor que, en primer lugar, nos diga si la frase introducida está escrita en el idioma de Javalandia (en cualquiera de sus dialectos), y en caso afirmativo, nos muestre solo el mensaje sin muletillas.

```

import java.util.Scanner;
/*
* Para ver si la frase está escrita en javalandés, miramos si empieza o termina
* por el prefijo o el sufijo de sus dialectos. Para ello, usamos los métodos

```

```

* startsWith() y endsWith() de la clase String. Para extraer el mensaje,
* utilizamos dos versiones sobrecargadas de substring() */

public class Main {

    public static void main(String[] args) {
        final String prefijo = "Javalín, javalón"; //constantes con el comienzo y la
        final String sufijo = "javalén, len, len"; //terminación de las frases
        Scanner sc = new Scanner(System.in);

        System.out.print("Escriba una frase: ");
        String entrada = sc.nextLine();//texto de entrada al traductor
        boolean idiomaJavalandia = false; //suponemos que entrada no está javalándés

        //Vamos a comprobar si el texto de entrada empieza o termina con alguna muletilla
        if (entrada.startsWith(prefijo)) { //si la frase comienza con prefijo
            idiomaJavalandia = true; //el idioma es javalándés
            entrada = entrada.substring(prefijo.length()); //quitamos el prefijo
            //nos quedamos con los caracteres de entrada a partir del siguiente al prefijo

        } else if (entrada.endsWith(sufijo)) { //si la entrada termina con sufijo
            idiomaJavalandia = true; //es javalándés
            entrada = entrada.substring(0, entrada.length() - sufijo.length()); //quitamos
            //el sufijo. Nos interesa desde el primer carácter de la entrada (0) hasta el
            //carácter antes del sufijo
        }

        if (idiomaJavalandia) {
            entrada = entrada.trim(); // quitamos los espacios antes y después
            System.out.println(entrada); //mostramos
        } else {
            System.out.println("No está escrito en el idioma de Javalandia");
        }
    }
}

```

6.10. Disponemos de la siguiente relación de letras:

conjunto 1:	e	i	k	m	p	q	r	s	t	u	v
conjunto 2:	p	v	i	u	m	t	e	r	k	q	s

mediante la cual es posible codificar un texto, convirtiendo cada letra del conjunto 1, en su correspondiente del conjunto 2. El resto de las letras no se modifican. Los conjuntos se utilizan tanto para codificar mayúsculas como minúsculas, mostrando siempre la codificación en minúsculas.

Un ejemplo: la palabra «**PaquiTo**» se codifica como «**matqvko**».

Se pide realizar un programa codificador, en el que se implemente la función,

```
char codifica(char conjunto1[],char conjunto2[], char c)
```

que devuelve el carácter **c** codificado según los conjuntos 1 y 2 que se le pasan.

```

import java.util.Scanner;
/*
 * En primer lugar vamos a convertir el texto introducido a minúsculas, para que los
 * alfabetos conjunto 1 y 2 nos sirvan para todas las letras (mayúsculas o minúsculas).
 * El procedimiento a seguir será recorrer y codificar el texto introducido, carácter a
 * carácter. La codificación se almacenará en una tabla, que nos permite asignar valores
 * (caracteres) a cada uno de sus elementos.
 */

```

```
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final char conjunto1[] = {'e', 'i', 'k', 'm', 'p', 'q', 'r', 's', 't', 'u', 'v'};
        final char conjunto2[] = {'p', 'v', 'i', 'u', 'm', 't', 'e', 'r', 'k', 'q', 's'};
        char codificado[]; //tabla que contendrá la codificación del texto introducido
        String texto;

        System.out.print("Introduzca un texto a codificar: ");
        texto = sc.nextLine();
        texto = texto.toLowerCase(); //convertimos el texto a minúscula, para poder codifi-
        //car las mayúsculas y las minúsculas con el mismo conjunto.
        codificado = new char[texto.length()]; //creamos una tabla de igual tamaño que texto

        for (int i = 0; i < texto.length(); i++) { // recorremos el texto a codificar
            //codificamos el i-ésimo carácter del texto
            codificado[i] = codifica(conjunto1, conjunto2, texto.charAt(i));
        }
        texto = String.valueOf(codificado); //convertimos la tabla con la codificación
        //en una cadena
        System.out.println(texto);
    }

    //Esta función codifica el carácter c según los alfabetos conjunto 1 y 2.
    //Buscamos el carácter c en conjunto1. Si se encuentra en la posición pos,
    //se devuelve el carácter equivalente en el segundo conjunto: conjunto2[pos].
    //En caso de no encontrar c en conjunto 1 se devuelve c sin codificar.

    static char codifica(char conjunto1[], char conjunto2[], char c) {
        final String conj1 = String.valueOf(conjunto1); //conj1 es un String con los
        //elementos de la tabla conjunto1. Facilita la búsqueda.
        char codificado; //carácter codificado correspondiente a c

        int pos = conj1.indexOf(c); //buscamos c en el conjunto 1. Al ser conj1 una cadena,
        //indexOf() busca por nosotros. En otro caso, tendríamos que buscar mediante un
        //bucle un elemento en una tabla
        if (pos == -1) { //si no hemos encontrado c en conj1
            codificado = c; //no podemos codificar, devolveremos c
        } else {
            codificado = conjunto2[pos]; //pos marca la posición de c en conjunto 1
            //entonces elegimos el correspondiente en conjunto2
        }

        return codificado;
    }
}
```

6.11. Realizar un programa descodificador.

La solución es tan sencilla como utilizar la función diseñada en el ejercicio anterior intercambiando los conjuntos entre sí.

6.12. Un anagrama es una palabra, o frase, que resulta de la transposición de otra palabra o frase. Ejemplos de anagramas para la palabra *roma* son: *amor*, *ramo* o *mora*. Construir un programa que solicite al usuario dos palabras e indique si son anagramas una de otra.

```
import java.util.Arrays;
import java.util.Scanner;
/*
 * El algoritmo que comprueba si cada letra de la palabra 1 se encuentra en la palabra 2,
```

```

* y lo que es más importante, comprobar que cada letra, tanto de la palabra 1 como de la
* 2 solo se utilizan una vez. Este algoritmo puede ser algo más laborioso de escribir.
* Vamos a buscar una propiedad de los anagramas que nos facilite el trabajo. Para que
* dos palabras sean anagramas tienen que tener la misma longitud y, tienen que tener las
* mismas letras el mismo número de veces. Lo que haremos es ordenar las letras de cada
* palabra y comprobar si son iguales. Un ejemplo: (sin vocales acentuadas)
* "esponja" -> ordenamos las letras: "aejnops" -> son iguales
* "japones" -> ordenamos las letras: "aejnops" -> son iguales
*/

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String palabra1, palabra2;

        System.out.println("Escriba una palabra: ");
        palabra1 = sc.nextLine();
        System.out.println("Escriba otra: ");
        palabra2 = sc.nextLine();

        if (palabra1.length() != palabra2.length()) { //si son de distintos tamaño
            System.out.println("No son anagramas"); //no pueden ser anagramas
        } else {
            char p1[] = palabra1.toCharArray(); //es más fácil ordenar una tabla
            char p2[] = palabra2.toCharArray(); //convertimos las palabras a tablas

            Arrays.sort(p1); //ordenamos ambas tablas
            Arrays.sort(p2);

            palabra1 = String.valueOf(p1); //volvemos a convertir en cadenas las tablas
            palabra2 = String.valueOf(p2); //es más fácil comparar cadenas

            if (palabra1.equalsIgnoreCase(palabra2)) { //si son iguales
                System.out.println("Son anagramas"); //son anagramas
            } else {
                System.out.println("No son anagramas");
            }
        }
    }
}

```

6.13. Diseñar un algoritmo que lea del teclado una frase e indique, para la letras que aparecen en la frase, cuántas veces se repite cada una. Se consideran iguales las letras mayúsculas y las minúsculas para realizar la cuenta. Un ejemplo sería:

```

Frase: En un lugar de la Mancha.
Resultado:
a: 4 veces
d: 1 vez
e: 2 veces
. . .

```

```

import java.util.Scanner;
/*
* Vamos a utilizar una tabla de contadores (numVeces) donde cada elemento de la tabla
* corresponde a una letra y donde se almacena el número de veces que aparece en la
* frase dicha letra. numVeces tendrá tantos elementos como letras tiene el alfabeto,
* es decir, 'z'-'a'+1 elementos, ya que las letras del abecedario tienen valores
* Unicode correlativos.
* A un carácter cualquiera c, le corresponde el elemento de la tabla con posición c-'a':
* numVeces[c-'a'], que se incrementa cada vez que haya una ocurrencia de c en la frase.

```

```

* Si numVeces[c-'a'] es 10 significa que el carácter c aparece 10 veces, pero si es
* 0 significa que c no aparece en la frase introducida. */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase;
        int[] numVeces; //contador de las ocurrencias de cada letra

        System.out.print("Introduzca una frase: ");
        frase = sc.nextLine();
        //para contabilizar las mayúsculas pasamos todo a minúsculas
        frase = frase.toLowerCase();

        // Cada posición de numVeces,guardará el número de ocurrencias de una letra.
        // numVeces[0] para la 'a', numVeces[1] para la 'b', numVeces[2] para la 'c'...
        numVeces = new int['z' - 'a' + 1]; //tantas componentes como letras.
        //La tabla se crea con todos los elementos inicializados a 0

        for (int i = 0; i < frase.length(); i++) { //recorre la frase carácter a carácter
            if (Character.isLetter(frase.charAt(i))) { //si el i-ésimo carácter es una letra
                numVeces[frase.charAt(i) - 'a']++; //incrementamos el contador de esa letra
            }
        }

        for (int i = 0; i < 'z' - 'a'; i++) { //mostramos numVeces
            if (numVeces[i] != 0) { //solo las letras que aparecen en frase
                System.out.println("La letra " + (char) (i + 'a')
                    + " se repite " + numVeces[i] + " veces");
            }
        }
    }
}

```

- 6.14.** Implementar el *juego del anagrama*, que consiste en que un jugador escribe una palabra o frase, y la aplicación muestra un anagrama (transposición de los caracteres) del texto introducido generado al azar. A continuación otro jugador tiene que acertar cuál es el texto original. La aplicación no debe permitir que el texto introducido por el jugador 1 sea la cadena vacía. Por ejemplo, si el jugador 1 escribe “teclado”, la aplicación muestra como pista un anagrama al azar: “etcloda”.

```

import java.util.Scanner;
/*
* Para generar un anagrama del texto original, vamos a seguir el siguiente algoritmo:
* - Primero convertir la cadena original en una tabla, donde es más cómodo intercambiar
*   caracteres.
* - A continuación, elegiremos dos índices al azar de la tabla, y se intercambian
* - El punto anterior se puede repetir varias veces, nosotros los haremos tantas veces
*   como caracteres tenga el texto original. */

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String original, intento; //texto original (jugador1) e intentos del jugador 2
        char anagrama[]; //tabla de caracteres con un anagrama, al azar, de original

        do {
            System.out.print("Jugador 1. Introduzca una frase: ");
            original = sc.nextLine();
        } while (original.isEmpty());
    }
}

```

```
anagrama = original.toCharArray(); //convierte en tabla, más cómoda para modificar
int tam = original.length(); //longitud del texto introducido

//realizamos un intercambio al azar por cada carácter que forma el texto
for (int numCambios = 0; numCambios < tam; numCambios++) {
    int i = (int) (Math.random() * tam); //número al azar entre 0 y tam-1
    int j = (int) (Math.random() * tam);

    char aux = anagrama[i]; //intercambiamos los elementos anagrama[i] y anagrama[j]
    anagrama[i] = anagrama[j];
    anagrama[j] = aux;
}

String anagramaFinal = String.copyValueOf(anagrama); //para comparar
System.out.println("Teclee un anagrama de: " + anagramaFinal);

do {
    System.out.println("Jugador 2. ¿Cuál es el original? ");
    intento = sc.nextLine();
} while (!original.equals(intento)); //mientras no acierte el texto original
System.out.println("Muy bien..."); //si salimos del bucle es que ha acertado
}
}
```

6.15. Modificar el ejercicio anterior para que el programa indique al jugador 2, en cada intento, cuántas letras coinciden con el texto original.

```
import java.util.Scanner;
/*
 * El programa es exactamente igual, con la diferencia que utilizamos una
 * función para contar los aciertos.
 */

public class Main {

    public static void main(String[] args) {
        ... //código idéntico al ejercicio anterior

        //último bucle do-while
        do {
            System.out.println("Jugador 2. ¿Cuál es el original? ");
            intento = sc.nextLine();
            System.out.println("Letra que coinciden:" + letrasCoinciden(original, intento));
        } while (!original.equals(intento)); //mientras no acierte el texto original
        System.out.println("Muy bien..."); //si salimos del bucle es que ha acertado
    }

    //Cuenta y devuelve el número de letras que coinciden en dos cadenas
    static int letrasCoinciden(String a, String b) {
        int l = Math.min(a.length(), b.length());
        int cont = 0;

        for (int i = 0; i < l; i++) {
            if (a.regionMatches(i, b, i, 1)) { //compara letra a letra en a y b
                cont++;
            }
        }
        return cont;
    }
}
```


Ejercicios propuestos

6.1. Realizar el juego del ahorcado. Brevemente las reglas del juego son:

- a) El jugador A teclea una palabra, sin que el jugador B la vea.
- b) Ahora se le muestra tantos guiones como letras tenga la palabra secreta. Por ejemplo, para “hola” será “_ _ _ _”.
- c) El jugador B intentará acertar, letra a letra, la palabra secreta.
- d) Cada acierto muestra la letra en su lugar y las letras no acertadas seguirán ocultas como guiones. Siguiendo con el ejemplo anterior y suponiendo que se ha introducido: la ‘o’, la ‘j’ y la ‘a’, se mostrará: “_ o _ _ a”.
- e) Cada letra no acertada restará un intento. Supondremos que el jugador B solo tiene 7 intentos.
- f) La partida terminará cuando se acierten todas las letras que forman parte de la palabra secreta (ganado el jugador B) o cuando se agoten todos los intentos (ganando el jugador A).

6.2. El preprocesador del lenguaje C, elimina los comentarios (`/* ... */`) del código fuente antes de compilar. Diseñar un programa que lea por teclado una sentencia en C, y elimine los comentarios.

```
Sentencia: if (a==3) /* igual a tres */ a++; /* incrementamos a */
Salida : if (a==3) a++;
```

6.3. Diseñar una aplicación que se comporte como una pequeña agenda. Mediante un menú el usuario podrá elegir entre:

- 1) Añadir un nuevo contacto (nombre y teléfono), siempre y cuando la agenda no esté llena.
- 2) Buscar el teléfono de un contacto a partir de su nombre.
- 3) Mostrar la información de todos los contactos ordenados alfabéticamente mediante el nombre.

En la agenda guardaremos el nombre y el teléfono de un máximo de 20 contactos.

Capítulo 7

Clases

Hasta este capítulo, hemos utilizado un paradigma de programación llamado *Programación estructurada*, que emplea las estructuras de control —condicionales y bucles—, junto a datos y funciones. Una de sus principales características es que no existe un vínculo fuerte entre funciones y datos.

Pensamos que, desde el punto de vista didáctico, es mejor, para dar los primeros pasos, comenzar con la *Programación estructurada*. Pero llegado a un punto, es conveniente saltar a un nuevo paradigma, la *Programación orientada a objetos* (en adelante, POO), que amplía los horizontes de un programador, dotándolo de nuevas herramientas para afrontar problemas más complejos.

La POO se inspira en una abstracción del mundo real, en la que los objetos se clasifican en grupos. Por ejemplo, todos los mamíferos de cuatro patas que dicen *jguau!* los englobamos dentro del grupo de los perros. Si observamos a Pepa, Paco y Miguel, vemos que los tres pertenecen al mismo grupo: los tres son personas. Pepa es una persona, Paco es una persona y Miguel también es una persona. Todos¹ pertenecemos al grupo de personas. En el argot de la POO, a cada uno de estos grupos se le denomina *clase*.

Podemos definir cada grupo o clase mediante las propiedades y comportamientos que presentan todos sus miembros. Una propiedad es un dato que conocemos de cada miembro del grupo, mientras que un comportamiento es algo que puede hacer.

Vamos a definir la clase persona mediante,

- **Propiedades:** un nombre, una edad, una estatura. Tanto Pepa, como Paco, como Miguel tienen un nombre, una edad y una estatura; son datos que en cada uno tendrá un valor distinto.
- **Comportamientos:** Pepa, Paco y Miguel —en realidad todas las personas— pueden saludar, crecer o cumplir años, por ejemplo.

Como vemos, es posible definir una clase mediante un conjunto de propiedades y comportamientos. La sintaxis para definir una clase utiliza la palabra reservada **class**,

¹Siempre y cuando el lector sea un ser humano. Y para evitar problemas consideraremos a los autores como personas.

```
class NombreClase2 {  
    //definición de la clase  
}
```

Por ejemplo la clase `Persona`, se define,

```
class Persona {  
    //definición de Persona  
}
```

7.1. Crear una clase desde NetBeans

La definición de una clase tiene que ser escrita en un fichero independiente, que debe nombrarse igual que la clase y tendrá extensión `.java`. Gracias a NetBeans no tendremos que estar pendientes de estos detalles. La forma de crear una clase desde NetBeans es la siguiente:

1. Pulsar en la opción del menú *File/New File...*, que accede a la ventana que se muestra en la Figura 7.1

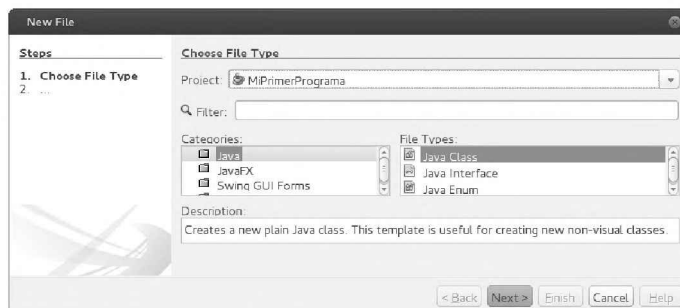


Figura 7.1. Crear un elemento: elegir el tipo

Tendremos que seleccionar el proyecto en el que queremos crear la clase, ya que es posible trabajar con más de un proyecto simultáneamente. A continuación elegimos la categoría **Java**, y dentro de esta, qué tipo de fichero deseamos crear. En nuestro caso una clase: *Java Class*.

2. Mediante el botón **Next** **>**, accedemos a una ventana (Figura 7.2) que nos permite asignar el nombre de la clase, así como en qué paquete deseamos que se ubique.
3. Finalmente, mediante el botón **Finish**, terminamos el proceso.

A partir de ahora disponemos de la nueva clase, en la que falta definir sus atributos y métodos.

²Para distinguir las clases de las variables se utiliza la notación *Camel* con la primera letra en mayúscula.

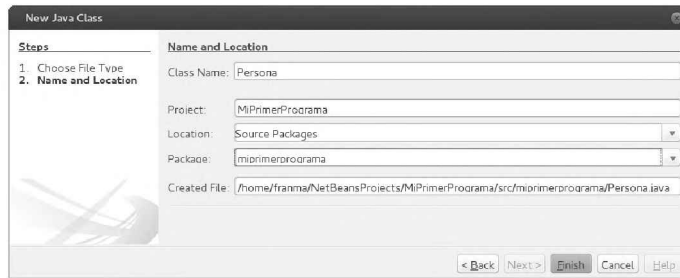


Figura 7.2. Propiedades de la clase

7.2. Atributos

Los datos que definen una clase se denominan atributos. Por ejemplo, la clase vehículo puede definirse mediante los atributos: matrícula, color, marca y modelo. Como se ha visto, la clase **Persona** dispone de los atributos: nombre, edad y estatura.

La forma de declarar los atributos en una clase es,

```
class NombreClase {
    tipo atributo1;
    tipo atributo2;
    . . .
}
```

El tipo especificado en *tipo* puede ser cualquier tipo primitivo —o una clase como veremos a lo largo de este capítulo—. El código para definir nuestra clase **Persona** será:

```
class Persona {
    String nombre;
    byte edad;
    double estatura;
}
```

7.2.1. Inicialización

Es posible asignar un valor por defecto a los atributos de una clase, esto se realiza en la propia declaración de la forma,

```
class NombreClase {
    tipo atributo1 = valor;
    . . .
}
```

7.3. Objetos

Los elementos que pertenecen a una clase se denominan *instancias* u *objetos*. Cada uno tiene sus propios valores de los atributos definidos en la clase. Explicaremos este concepto con un símil; supongamos que una clase son los planos para construir una casa. Estos planos se pueden utilizar en repetidas ocasiones, siendo cada una de las casas construidas, casas reales, un objeto de dicha clase. Todas las casas construidas —los objetos— tendrán la misma distribución. Esto es lógico, ya que utilizamos el mismo plano —la clase—. Sin embargo, cada una de las casas —objetos— tendrá distintas particularidades: distinto color de fachada, distinto modelo de puertas, etc.

Si nos fijamos de nuevo en Pepa, Paco y Miguel, nos damos cuenta de que cada uno de ellos es un objeto de la clase **Persona**. La Figura 7.3 muestra la clase³ junto a tres objetos con distintos valores para sus atributos.

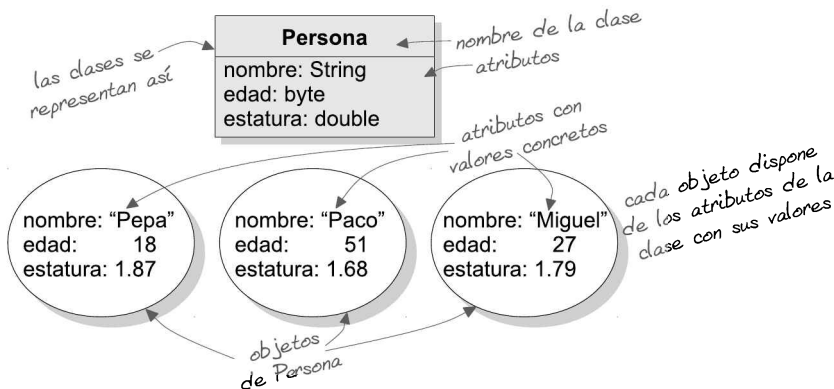


Figura 7.3. Ejemplo de valores de atributos para personas

7.3.1. Referencia

Antes de ver con más detalle la forma de crear objetos, necesitamos entender el concepto de dirección de memoria. La memoria de un ordenador está formada por pequeños bloques consecutivos de un tamaño predeterminado, que dependerá del tipo de hardware utilizado. Cada uno de estos bloques se identifica mediante un número único, que se denomina dirección de memoria. Veamos un ejemplo: si dispongo de una memoria de 1 Kbytes —1024 bytes— y la arquitectura de mi ordenador utiliza bloques de 64 bits —8 bytes—, tendré mi memoria formada por un total de 128 bloques⁴. Cada uno de estos bloques se identifica mediante su dirección de memoria, que serán: 1, 2, 3, ... 128.

A la hora de numerar los bloques de memoria se utilizan números hexadecimales⁵. Por este motivo las direcciones de memoria suelen tener un aspecto como `2a139f55`.

³Para representar las clases y sus relaciones entre ellas se utilizan los diagramas de clases.

⁴Dividimos 1024 entre 8.

⁵Son números que no solo utilizan los dígitos del 0 al 9, sino que también utilizan los dígitos A, B, C, D, E y F.

Cualquier dato almacenado en la memoria ocupará, dependiendo de su tamaño, una serie de bloques consecutivos y puede ser identificado mediante la dirección de memoria del primer bloque que ocupa. A esta primera dirección de memoria que identifica a un dato se le denomina en Java *referencia*.

Cualquier dato en memoria, incluidos los objetos, se identifica mediante su referencia, es decir, mediante la dirección del primer bloque que ocupa en la memoria.

7.3.2. Variables referencia

Antes de construir objetos necesitamos declarar variables cuyo tipo sea una clase. La declaración sigue las mismas reglas que las variables de tipo primitivo,

```
Clase nombreVariable;
```

donde *Clase* será el nombre de cualquier clase disponible.

Veremos cómo declarar la variable *p* de tipo *Persona*,

```
Persona p; //p es una variable de tipo Persona
```

La diferencia entre una variable de tipo primitivo y una variable de tipo referencia es que, mientras una variable de tipo primitivo almacena directamente un valor, una variable del tipo clase almacena la referencia de un objeto.

7.3.3. Operador new

La forma de crear objetos es mediante el operador **new**, que se utiliza,

```
p = new Persona();
```

En este caso, crea un objeto de tipo *Persona* y asigna su referencia a la variable *p*.

El operador **new**, primero, busca en memoria un hueco disponible donde construir el objeto. Este, dependiendo de su tamaño⁶ ocupará cierto número consecutivo de bloques de memoria. Por último, devuelve la referencia del objeto recién creado, que se asigna a la variable *p*.

Podemos comprobar qué aspecto tiene una referencia ejecutando,

```
p = new Persona();  
System.out.println(p); //muestra en consola la referencia del objeto
```

A la hora de trabajar con referencias, es bastante más sencillo pensar en ellas como flechas que se dirigen desde la variable hacia el objeto (véase Figura 7.4).

En el momento en que disponemos de un objeto podemos acceder a sus atributos mediante un punto (.) Por ejemplo, para asignar valores a los atributos del objeto referenciado por *p* escribimos,

```
p = new Persona();  
p.nombre = "Pepa";  
p.edad = 18;  
p.estatura = 1.87;
```

⁶El tamaño depende del tipo y la cantidad de atributos que tenga la clase.

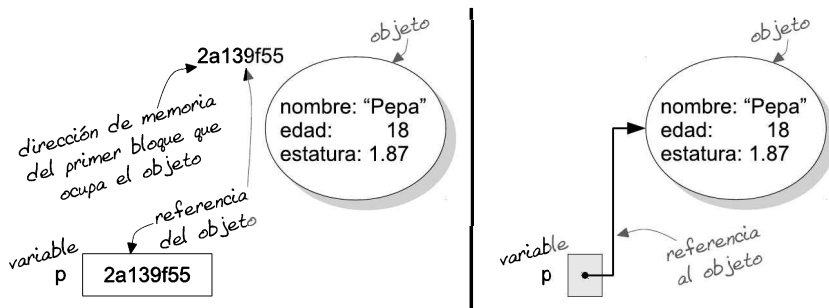


Figura 7.4. Dos formas de representar una misma referencia

Es importante comprender que podemos acceder al mismo objeto mediante distintas variables que almacenan la misma referencia. La Figura 7.5 representa el siguiente código, donde un objeto está referenciado por dos variables.

```
Persona p1, p2;
p1 = new Persona(); //p1 referencia al objeto creado
p2 = p1; //p2 referencia el mismo objeto que p1
p2.nombre = "Pepa" //es equivalente a utilizar p1.nombre
```

Ahora podemos acceder al objeto de dos maneras: mediante `p1` o mediante `p2`. En ambos casos estamos utilizando el mismo objeto.

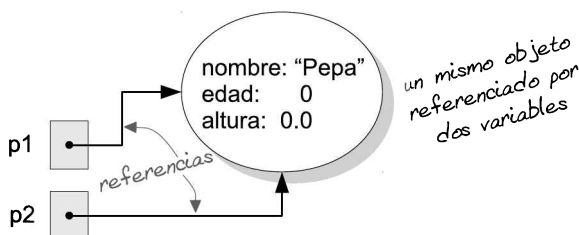


Figura 7.5. Dos formas de ver una misma referencia

En realidad, ya hemos estado usando clases y objetos. Las tablas son objetos especiales, con una sintaxis peculiar, pero nos sirven para ilustrar el uso de referencias. Por ejemplo, declaramos dos variables del tipo tabla de enteros,

```
int t1[], t2[];
```

Creamos una tabla de 4 enteros —un objeto— cuya referencia asignamos a la primera variable,

```
t1 = new int[4];
```

Dicha referencia también se puede asignar a `t2`,

```
t2 = t1;
```


Ahora `t1` y `t2` referencian al mismo objeto. Prueba de ello es que, si asignamos un valor a un elemento de `t1`, tendremos el mismo valor en el elemento correspondiente de `t2`,

```
t1[0] = 1000;
System.out.println(t2[0]); //aparece 1000 en la pantalla
```

7.3.4. Referencia null

El valor literal `null` es una referencia nula. Dicho de otra forma, una referencia a ningún bloque de memoria. Cuando declaramos una variable referencia se inicializa por defecto a `null`.

Hay que tener mucho cuidado con intentar acceder a los miembros de una referencia nula, ya que produce un error, terminando de forma inesperada la ejecución del programa.

```
Persona p; //se inicializa por defecto a null
p.nombre //¡error!
```

La última instrucción genera un error del tipo: *Null pointer exception*, que significa que estamos intentando acceder a los atributos de un objeto nulo.

El literal `null` se puede asignar a cualquier variable referencia,

```
Persona p = new Persona(); //p referencia un objeto
...
p = null; //p no referencia nada
```

7.3.5. Recolector de basura

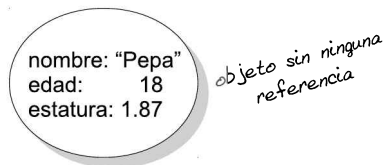


Figura 7.6. Objeto sin referencia

Existen dos formas de conseguir que un objeto no esté referenciado:

- Es posible, aunque no tenga mucho sentido, crear un objeto y no asignarlo a ninguna variable,

```
new Persona();
```

- Otra posibilidad es asignar `null` a todas las variables que contenían una referencia al objeto.

En ambos casos, el objeto se queda *perdido* en memoria, es decir, no existe forma de acceder a él (véase Figura 7.6). Y además, está ocupando memoria. Si este comportamiento se repite demasiado, fortuita o malintencionadamente, es posible que se agote toda la memoria libre disponible, lo que impediría el normal funcionamiento del ordenador.

Para evitar este problema, Java dispone de un mecanismo llamado recolector de basura —*garbage collection*—, que se ejecuta periódicamente de forma transparente al usuario, y que se encarga de comprobar, uno a uno, todos los objetos de la memoria. Si alguno de ellos no estuviera referenciado por nadie, lo que implica la imposibilidad de utilizarlo, se destruye, liberando la memoria que ocupa.

7.4. Métodos

Hemos declarado clases con atributos pero, como vimos, las clases también disponen de comportamientos. En el argot de la POO, a los comportamientos u operaciones que pueden realizar los objetos de una clase se les denomina *métodos*. Por ejemplo, las personas son capaces de realizar operaciones como saludar, cumplir años, crecer, etc.

Los métodos no son más que funciones que se implementan dentro de una clase, con la sintaxis:

```
public class NombreClase {  
    ... //declaración de atributos  
  
    tipo nombreMétodo (parámetros) {  
        cuerpo del método  
    }  
}
```

La definición de un método es la de una función, siendo *cuerpo del método* un bloque de instrucciones. A partir de ahora, dentro de una clase prescindiremos del modificador **static** delante de los métodos; más adelante explicaremos cuándo es necesario utilizarlo. Ampliemos la clase **Persona** con algunos métodos:

```
public class Persona {  
    String nombre;  
    byte edad;  
    double estatura;  
  
    void saludar() {  
        System.out.println("Hola. Mi nombre es " + nombre);  
        System.out.println("Encantando de conocerte");  
    }  
    void cumplirAños() {  
        edad++; //incrementamos la edad en 1  
    }  
    void crecer(double incremento) {  
        estatura += incremento; //la estatura aumenta cierto incremento  
    }  
} //de la clase
```

La Figura representa la clase **Persona** con sus atributos y métodos.

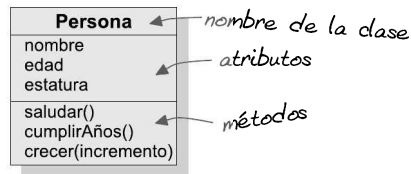


Figura 7.7. Clase Persona

A partir de ahora, los objetos de tipo **Persona** pueden invocar sus métodos utilizando un punto (.), al igual que se hace con los atributos. Veamos un ejemplo,

```
Persona p;
p = new Persona();

p.edad = 18;
p.cumplirAños(); // ¡felicidades! La edad de p se incrementa
```

Tanto a los atributos como a los métodos de una clase se les llama de forma genérica miembros. De esta forma, al hablar de miembros de una clase, hacemos referencia a los elementos declarados en su definición, ya sean atributos o métodos.

Los métodos de una clase tienen acceso a las siguientes variables: variables locales declaradas dentro del método, parámetros de entrada y atributos de la clase.

7.4.1. Ámbito de las variables y atributos

El ámbito de una variable define en qué lugar puede utilizarse. Este coincide con el bloque en el que se declara la variable que, como se vió en el Apartado 4.2, puede ser,

- Bloque de una estructura de control: **if**, **if-else**, **switch**, **while**, **do-while** o **for**. Las variables declaradas en este ámbito se denominan *variables de bloque*.
- Una función o método. A las variables declaradas aquí se conocen como *variables locales*.

Con la POO, aparece un nuevo ámbito,

- La clase. Cualquier miembro, atributo o método definido en una clase podrá ser utilizado en cualquier lugar de la misma. A las variables declaradas en su definición se les llama *atributos*.

Un ámbito puede contener otros, formando una estructura jerárquica. Por ejemplo, una clase puede contener dos métodos, y estos, distintos bloques de, por ejemplo, una estructura **while** o **if**.

Una variable puede utilizarse en el ámbito o bloque en el que se declara, que incluye sus bloques internos. Sin embargo, no ocurre lo contrario: una variable no podrá utilizarse en el ámbito padre del bloque en la que se declara. Es decir, un atributo puede emplearse

dentro de un método, y una variable local dentro del bloque de una estructura de control. Pero no podremos usar una variable local fuera de su método, ni una variable de bloque fuera de él. El código de la Figura 7.8 muestra el ámbito de tres variables, donde **atributo** puede

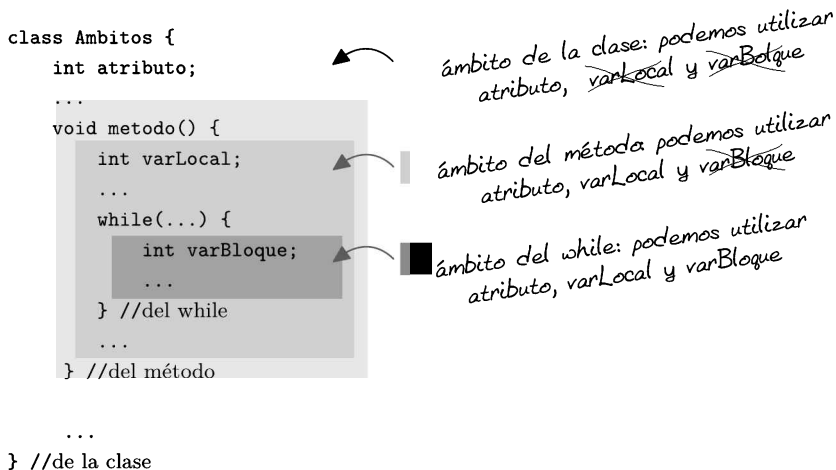


Figura 7.8. Ámbitos de las variables

utilizarse en cualquier lugar de la clase, **varLocal** en cualquier lugar dentro del método en el que se declara; por último, **varBloque** puede usarse solo dentro del bloque de instrucciones de **while**.

7.4.2. Ocultación de atributos

Dos variables declaradas en ámbitos anidados no pueden tener el mismo identificador, ya que esto genera un error. Existe una excepción, cuando una variable local en un método tiene el mismo identificador que un atributo de la clase. En este caso, la variable local tiene prioridad sobre el atributo, lo que provoca que al utilizar el identificador se acceda a la variable local y no al atributo. En la jerga de la POO se dice que la variable local *oculta* al atributo.

Veamos un ejemplo,

```

public class Ambito {
    int edad; //atributo entero

    void metodo() {
        double edad; //variable local. Oculta al atributo edad (entero)
        edad = 8.2; //variable local, no el atributo de la clase
        ...
    }
}

```

7.4.3. Objeto `this`

La palabra reservada `this` permite utilizar un atributo incluso cuando ha sido oculto por una variable local. De igual manera que nos referimos a nosotros mismos como *yo*, aunque tengamos un nombre que los demás utilizan para identificarnos, las clases se refieren a si mismas como `this`, que es una referencia al objeto actual y funciona como una especie de *yo* para clases. Al escribir `this` en el ámbito de una clase se interpreta como *la propia clase*, y permite acceder a los atributos aunque se encuentren ocultos.

Estudemos el siguiente fragmento de código donde, en el ámbito de un método, una variable local oculta un atributo de la clase,

```
public class Ambito {
    int edad; //atributo entero

    void metodo() {
        double edad; //oculta el atributo edad (entero)

        edad = 20.0; //variable local, no el atributo
        this.edad = 30; //atributo de la clase
    }
}
```

7.5. Atributos y métodos estáticos

Un atributo *estático* o *de clase* es aquel del que no existe una copia en cada objeto. Todos los objetos de una misma clase comparten su valor.

Supongamos que necesitamos añadir a la clase **Persona** un atributo que nos indique qué día de la semana es hoy. Evidentemente si hoy es lunes, será lunes para todo el mundo; e igualmente si es martes, será martes para todos. Por tanto, el valor del atributo **hoy** será compartido por todos los objetos. No tiene sentido que para una persona sea lunes y para otra sea martes. La Figura 7.9 representa este concepto.

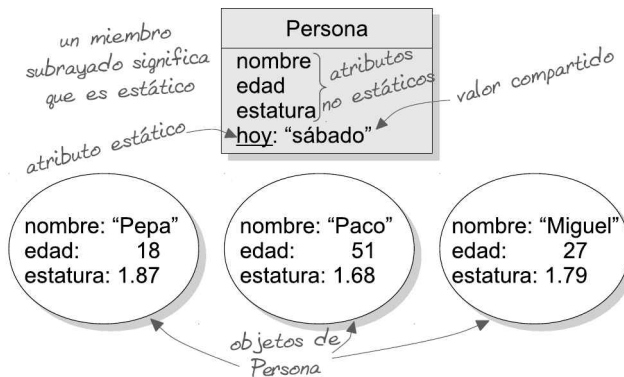


Figura 7.9. Atributo estático: hoy

Un atributo estático se declara mediante la palabra reservada **static**,

```
class Persona {  
    ...  
    static String hoy;  
}
```

Para acceder a un atributo estático es posible utilizar el nombre de la clase o cualquier objeto, de la siguiente forma:

```
Persona p = new Persona();  
  
Persona.hoy = "domingo";  
System.out.println(p.hoy); //mostrará "domingo"
```

Un atributo estático se inicializa una sola vez y se hace antes de crear ningún objeto. Si deseamos asignar un valor inicial al atributo **hoy** escribiremos,

```
class Persona {  
    ...  
    static String hoy = "lunes"; //valor inicial  
}
```

Con la misma filosofía, podemos declarar métodos estáticos. Son aquellos que no requieren de ningún objeto para ejecutarse y, por tanto, no pueden utilizar ningún atributo que no sea estático. En el caso de que un método estático intente utilizar un atributo no estático se producirá un error.

A modo de ejemplo, vamos a diseñar un método que actualice el atributo **hoy** a partir de un entero que se le pasa como parámetro. Este estará comprendido entre 1 y 7, que representan los días de la semana, de lunes a domingo,

```
static void hoyEs(int dia) {  
    switch (dia) {  
        case 1: hoy = "lunes";  
            break;  
        case 2: hoy = "martes";  
            break;  
        . . .  
  
        case 7: hoy = "domingo";  
            break;  
    }  
}
```

La forma de invocar un método estático es, al igual que con los atributos estáticos, mediante el nombre de la clase o cualquier objeto. Vamos a actualizar el día de hoy a martes,

```
Persona.hoyEs(2); //martes
```

Si tenemos un objeto **Persona** referenciado por la variable **p**, la línea anterior es equivalente a,

```
p.hoyEs(2); //martes
```

7.6. Constructores

¿Qué valor toman los atributos de un objeto recién creado? Los atributos a los que no se les asigna un valor en su declaración se inicializan por defecto, dependiendo de su tipo, de la siguiente manera: cero para valores numéricos, `null` para referencias y `false` para booleanos.

Antes de utilizar un objeto tendremos que asignar valores a cada uno de sus atributos. Por ejemplo, si deseamos crear un objeto de tipo `Persona` con nombre «Claudia», con una edad de 8 años y una estatura de 1.20 metros, es necesario escribir,

```
Persona p = new Persona(); //creamos el objeto
p.nombre = "Claudia"; //asignamos valores
p.edad = 8;
p.estatura = 1.20;
```

Este proceso —asignar valores— es necesario cada vez que creamos un objeto, si no queremos trabajar con los valores por defecto. El operador `new` facilita esta tarea mediante los *constructores*. Un *constructor* es un método especial que debe tener el mismo nombre que la clase, se define sin tipo devuelto —ni siquiera `void`—, y se ejecuta inmediatamente después de crear el objeto. El principal cometido de un constructor es asignar valores a los atributos, aunque también se puede utilizar para ejecutar cualquier código necesario: crear tablas, mostrar cualquier tipo de información, crear otros objetos que necesitemos, etc.

Al constructor, como cualquier otro método, se le pueden pasar parámetros y se puede sobrecargar. Vamos a implementar un constructor para `Persona`, que asigne los valores iniciales de sus atributos: `nombre`, `edad` y `estatura`. Escribiremos, dentro de la definición de la clase,

```
class Persona {
    ...
    Persona (String nombre, int edad, double estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
    }
    ...
}
```

Los valores de los parámetros de entrada se especifican al crear el objeto con `new` en la llamada al constructor. Si deseamos crear un objeto `Persona` con los datos anteriores sería,

```
Persona p = new Persona("Claudia", 8, 1.20); //creamos el objeto
//y lo inicializamos mediante el constructor
```

A la hora de sobrecargar un método tenemos que asegurarnos de que se puedan distinguir entre ellos mediante el número y/o el tipo de parámetros de entrada. La sobrecarga de constructores es útil cuando necesitamos inicializar objetos de varias formas. Hemos visto un constructor de `Persona` que permite asignar valores a todos los atributos. Podría darse el caso de que solo conociéramos el nombre de la persona, fijando valores arbitrarios para el resto de los atributos o dejando que se inicialicen con los valores por defecto.

```
class Persona {
    ...
    //constructor que asigna valores a todos los atributos
    Persona (String nombre, int edad, double estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
    }

    //constructor sobrecargado que solo asigna el nombre
    Persona (String nombre) {
        this.nombre = nombre;
        estatura = 1.0; //valor arbitrario para la estatura
        //al no asignar la edad se inicializa por defecto: a 0
    }
}
```

Ahora disponemos de dos constructores, que se utilizan de la forma:

```
Persona a = new Persona("Pepe", 20, 1.90);
Persona b = new Persona("Dolores");
```

7.6.1. this()

Cuando una clase dispone de un conjunto de constructores sobrecargados, es posible que un constructor invoque a otro y así reutilice su funcionalidad. El problema es que los constructores no se pueden invocar por su nombre directamente, sino mediante el constructor genérico `this()`, que es como se denominan a los constructores desde dentro de su clase. La forma de distinguir los distintos constructores, al igual que en cualquier método sobrecargado, es mediante el número y el tipo de los parámetros de entrada.

Vamos a reescribir los constructores de `Persona` utilizando `this()`,

```
class Persona {
    ...
    //constructor que asigna valores a todos los atributos
    Persona (String nombre, int edad, double estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
    }

    //constructor sobrecargado que solo asigna el nombre
    Persona (String nombre) {
        this(nombre, 0, 1.0); //invoca al primer constructor
        //la edad se pone a 0 y la estatura a 1.0
    }
}
```

Tenemos que tener presente que en el caso de utilizar `this()`, tiene que ser la primera instrucción de un constructor; en otro caso producirá un error.

7.7. Paquetes

En Java es importante controlar la accesibilidad de unas clases desde otras por razones de seguridad y eficiencia. Esto se consigue por medio de los *paquetes*, que son contenedores que nos permiten guardar las clases en compartimentos separados, de modo que podamos decidir, a través de la importación, qué clases son visibles desde una clase que estemos implementando.

Los paquetes se organizan de forma jerárquica, del mismo modo que las carpetas, y puede haber paquetes que contienen paquetes. Un *archivo fuente* de Java es un archivo de texto con extensión *.java*, que se guarda en un paquete y que contiene los siguientes elementos:

- Una sentencia donde se especifica el paquete al que pertenece, que empieza con la palabra clave **package** seguida del nombre del paquete.
- Una serie opcional de sentencias de importación, que empiezan con la palabra **import**, que nos permiten importar las clases definidas en otros paquetes.
- La definición de una o más clases, de las cuales solo una puede ser declarada pública (por medio del modificador de acceso **public**). De todas formas, es recomendable que, en cada archivo fuente se defina una sola clase, que debe tener el mismo nombre que el archivo.

No debemos olvidar que se importan las clases, no los paquetes. Si queremos importar todas las clases públicas de un paquete en una sola sentencia de importación, se usa el asterisco.

7.7.1. Crear un paquete desde NetBeans

Cada paquete se convierte físicamente en un directorio en nuestro ordenador que contiene las clases y a otros paquetes. Gracias a NetBeans no tendremos que estar pendientes ni del nombre, ni de la ubicación, ni de la estructura de estos directorios. La forma de crear un paquete desde NetBeans es la siguiente:

1. Pulsar en la opción del menú *File/New File...*, que accede a la ventana que se muestra en la Figura 7.10

Tendremos que seleccionar el proyecto en el que queremos crear el paquete, ya que es posible trabajar con más de un proyecto simultáneamente. A continuación elegimos la categoría **Java**, y dentro de esta, qué tipo de fichero deseamos crear. En nuestro caso un paquete: *Java Package*.

2. Mediante el botón **Next >**, accedemos a una ventana (Figura 7.11) que permite escribir el nombre del nuevo paquete y, mediante su nombre cualificado, en qué paquete se ubica, así como la localización, que será *Source Packages* por defecto.
3. Por último, mediante el botón **Finish**, terminamos el proceso.

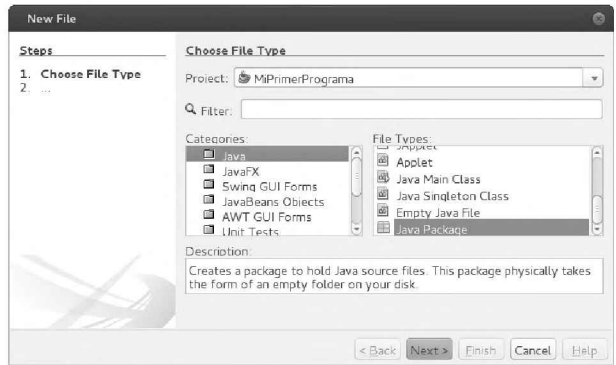


Figura 7.10. Crear un elemento: elegir el tipo

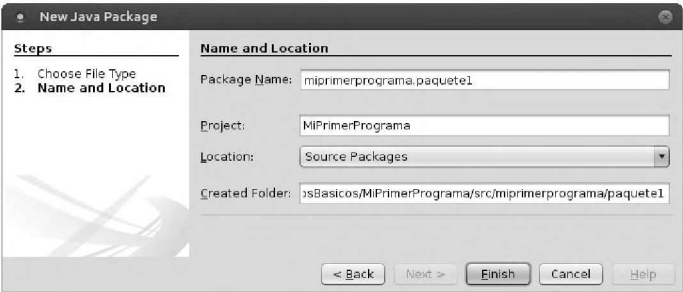


Figura 7.11. Propiedades del paquete

A partir de ahora disponemos del nuevo paquete, en el que falta definir las clases que contendrá. La estructura de paquetes de NetBeans queda se como muestra en la Figura 7.12

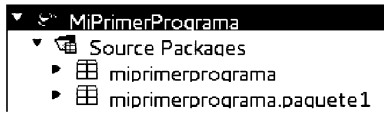


Figura 7.12. Estructura de paquetes resultante

7.8. Modificadores de acceso

Imaginemos que nos encontramos de excursión por la montaña en un precioso día soleado. Dependiendo del lugar donde nos situemos, disfrutaremos de una vista más o menos amplia. Si subimos al pico más alto, seremos capaces de ver toda la región colindante. Si

por el contrario, descendemos hasta el valle, solo podremos ver los lugares adyacentes. En Java, con las clases, ocurre algo similar.

Una clase será visible por otra dependiendo de si se ubican en el mismo paquete y de los modificadores de acceso que utilice. Estos alteran su visibilidad, permitiendo que se muestre u oculte.

De igual manera que podemos modificar la visibilidad entre clases, es posible modificar la visibilidad a nivel de miembros, es decir, qué atributos y métodos son visibles para otras clases.

7.8.1. Modificadores de acceso para clases

Debido a la estructura de clases, organizadas en paquetes, que utiliza Java, dos clases cualesquiera pueden definirse como (véase Figura 7.13):

- **Clases vecinas:** cuando ambas están definidas dentro del mismo paquete. Todas las clases que pertenecen a un paquete son vecinas entre sí.
- **Clases externas:** en el caso de que se encuentren definidas en paquetes distintos. Podemos ampliar este concepto entre una clase y un paquete, ya que una clase definida fuera de un paquete dado, no solo es externa a todas sus clases, sino al propio paquete.

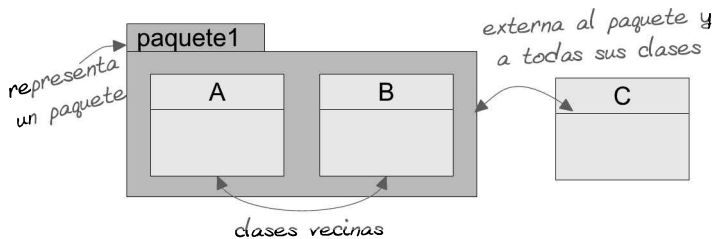


Figura 7.13. Clases vecinas y externas

Las clases siguen el lema *si lo ves, puedes utilizarlo*. Utilizar una clase significa crear objetos de ella o acceder a sus miembros estáticos.

Visibilidad por defecto

Cuando definimos una clase sin utilizar ningún modificador de acceso,

```
package paquete1;

class A { //sin modificador de acceso
    ...
}
```

se dice que utiliza visibilidad por defecto, que hace que solo sea visible por sus clases vecinas. En nuestro caso, A es visible por B, pero no será visible por C. La Figura 7.14 representa las visibilidad de A.

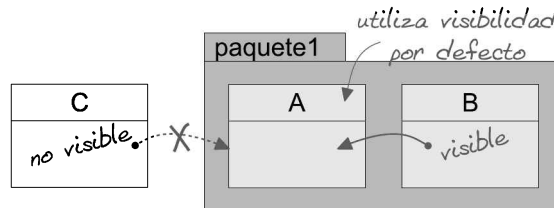


Figura 7.14. Visibilidad de la clase A

Visibilidad total

La clase A es invisible para todas las clases externas. ¿Cómo podemos hacer para que la clase A sea visible desde C? Mediante el modificador de acceso `public`, que produce el siguiente efecto: la clase marcada como pública será visible, además, desde cualquier clase externa con una sentencia de importación. El modificador `public` proporciona visibilidad total a la clase.

Vamos a redefinir A para que utilice visibilidad total,

```
package paquete1;

public class A { //clase marcada como pública
    ...
}
```

A partir de ahora, cualquier clase, vecina o externa, puede crear objetos o acceder a los miembros estáticos de A. Lo único que necesita una clase externa, como C, para utilizar A es importarla,

```
import paquete1.A; //ahora C puede utilizar la clase A

class C {
    ...
}
```

No debemos olvidar que se importan las clases, no los paquetes. Si queremos importar todas las clases públicas de un paquete en una sola sentencia de importación, se usa el asterisco.

```
import paquete1.*; //C puede utilizar cualquier clase visible en paquete1

class C {
    ...
}
```

La visibilidad entre clases puede resumirse como: una clase siempre será visible por sus clases vecinas. Que sea visible —previa importación— por clases externas dependerá de si está marcada como pública (Tabla 7.1).

Tabla 7.1. Resumen de la visibilidad entre clases

	Visible desde...	
	clases vecinas	clases externas
<i>sin modificador</i>	✓	
<i>public</i>	✓	✓

Veamos un ejemplo: necesitamos modelar el funcionamiento de una floristería, donde se venden rosas y margaritas, cortadas o plantadas en una maceta. En la floristería nunca se venden macetas vacías. Una posible solución es definir las clases **Rosa**, **Margarita** y **Maceta** dentro del paquete **floristeria**. Las dos primeras serán visibles desde fuera del paquete; y la clase **Maceta** será de uso interno al paquete. Cualquier clase externa, como la clase **Regalo**, podrá utilizar la clase **Rosa** o la clase **Margarita** (que vendrán cortadas o en una maceta) pero no será posible regalar una maceta vacía. La Figura 7.15 muestra esta solución.

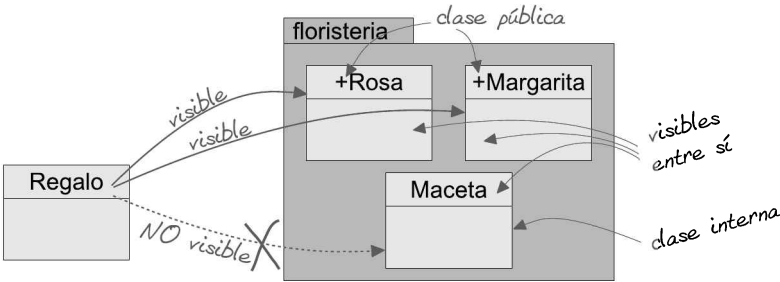


Figura 7.15. Paquete floristeria, con distinta visibilidad en sus clases

El código que modela este comportamiento es,

```
package floristeria;      package floristeria;      package floristeria;
public class Rosa {      public class Margarita {  class Maceta {
    ...                  }
}                        }
```

7.8.2. Modificadores de acceso para miembros

De igual manera que es posible modificar la visibilidad de una clase, podemos regular la visibilidad de sus miembros. Que un atributo sea visible significa que podemos acceder

a él, tanto para leer como para modificarlo. Que un método sea visible significa que puede invocarse.

Para que un miembro sea visible, es indispensable que su clase también lo sea. Es evidente que si no podemos acceder a una clase, no existe forma alguna de acceder a sus miembros.

Debemos destacar que cualquier miembro es siempre visible dentro de su propia clase, indistintamente del modificador de acceso que utilicemos. Es decir, dentro de la definición de una clase siempre tendremos acceso a todos los atributos y podremos invocar cualquiera de sus métodos. Veamos el siguiente ejemplo,

```
public class A { //clase pública
    int a; //su ámbito es toda la clase:
    ...    // a es accesible desde cualquier lugar de A
}
```

Visibilidad por defecto

Cuando queramos acceder a miembros de otra clase hay diversos grados de visibilidad. La visibilidad por defecto es aquella que se aplica a miembros declarados sin ningún modificador de acceso, como el atributo `a` en el código anterior.

La visibilidad por defecto implica que un miembro es visible desde las clases vecinas, pero invisible desde clases externas.

En nuestro ejemplo, el atributo `a` será,

- Visible por clases vecinas, lo que les permite tanto acceder a la clase `A` como acceder al atributo `a`. Acceder a una clase significa utilizar sus miembros visibles, incluidos los constructores que permiten crear objetos.
- Invisible desde clases externas. Cualquier clase externa podrá acceder a la clase `A`, pero no al atributo `a`.

Hay que notar que la clase `A` se ha definido `public`, lo que permite que sea visible desde clases externas. Si `A` no fuera visible desde el exterior, tampoco los serían sus miembros, sin importar el modificador utilizado en su declaración.

Modificador de acceso `private` y `public`

Con el modificador `private` obtenemos una visibilidad más restrictiva que por defecto, ya que impide el acceso incluso para las clases vecinas. Un miembro, ya sea un atributo o un método, declarado privado es invisible desde fuera de la clase.

Y `public` hace que un miembro sea visible incluso desde clases externas. Otorga visibilidad total.

El uso de `private` está justificado cuando queremos controlar los cambios de un atributo o cuando deseamos que no se conozca directamente su valor, o bien cuando queremos que un método solo sea invocado desde otros métodos de la clase, pero no fuera de ella. El acceso a esos miembros privados deberá hacerse a través de algún método `public` de la misma clase.

En el siguiente ejemplo se pide implementar la clase `Alumno` con los atributos `nombre` y `nota media`. Esta será un atributo privado, ya que interesa controlar el rango de valores válidos que estarán comprendidos entre 0 y 10, inclusive. El método público `asignaNota()` será el encargado de controlar el valor asignado a la nota.

```
public class Alumno {
    public String nombre;
    private double notaMedia;

    public void asignaNota(double notaMedia) {
        //nos aseguramos que esté en el rango 0..10
        if (notaMedia < 0 || notaMedia > 10) {
            System.out.println("Nota incorrecta");
        } else {
            this.notaMedia = notaMedia;
        }
    }
}
```

De este modo, solo es posible modificar la nota a través del método que la controla. La Tabla 7.2 muestra un resumen del alcance de la visibilidad de los miembros de una clase, según el modificador de acceso que se utilice.

Tabla 7.2. Alcance de la visibilidad según el modificador de acceso

	la propia clase	Visible desde. . .	
		clases vecinas	clases externas
<code>private</code>	✓		
<i><code>sin modificador</code></i>	✓	✓	
<code>public</code>	✓	✓	✓

Veamos como ejemplo la definición de la clase `Rosa`,

```
package floristeria;

public class Rosa {
    private double precio; //invisible fuera de la clase
    String color; //visible por clases vecinas
    public double tamaño; //visibilidad total
}
```

La Figura 7.16 representa la visibilidad de los atributos definidos en la clase `Rosa`.

7.8.3. Métodos `get/set`

Un atributo público puede ser modificado desde cualquier clase, lo que a veces tiene sus inconvenientes, ya que es imposible controlar los valores asignados, que pueden no tener sentido. Por ejemplo, nada impide que se asigne a un atributo `edad` un valor negativo, algo

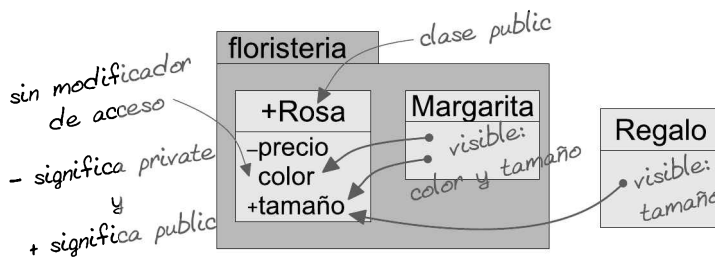


Figura 7.16. Visibilidad de private, public y por defecto

incoherente. Por este motivo, existe una convención en la comunidad de programadores que consiste en ocultar atributos públicos, y en su lugar, crear dos métodos: el primero (**set**) permite asignar un valor al atributo, controlando el rango válido de valores. Y el segundo (**get**) devuelve el atributo, lo que permite conocer su valor. Los métodos **set/get** hacen, en la práctica, que un atributo no visible se comporte como si lo fuera.

Estos métodos se identifican con **set/get** seguido del nombre del atributo. Para el atributo **edad** quedaría,

```
class Persona {
    private int edad;
    ...

    public void setEdad(int edad) {
        if (edad >= 0) //solo los valores positivos tienen sentido
            this.edad = edad;
        } // en caso contrario no se modifica la edad
    }

    public int getEdad() {
        return edad;
    }
}
```

Las ventajas de utilizar métodos **set/get** son que la implementación de la clase se encapsula, ocultando los detalles, y por otro lado permite controlar los valores asignados a los atributos. En nuestro ejemplo se ha limitado el uso de valores negativos para la edad.

7.9. Enumerados

Los tipos *enumerados* son parecidos a las clases. Sirven para definir grupos de constantes como valores posibles de una variable. Por ejemplo, **DiaDeLaSemana** sería un tipo enumerado que puede tomar solo los valores constantes: **LUNES**, **MARTES**,... **DOMINGO**. Se definiría de la forma,

```
public enum DiaDeLaSemana {
    LUNES,
```



```
MARTES,
MIERCOLES,
JUEVES,
VIERNES,
SABADO,
DOMINGO
}
```

Como puede verse, en la definición usamos la palabra clave `enum` y no `class`. En un programa se accede a los valores de la siguiente forma, `DiaDeLaSemana.LUNES`, etc.

Si queremos usar un tipo enumerado, lo podemos implementar en un archivo nuevo dentro del paquete donde está el programa principal, como si fuera una clase. En NetBeans, pulsamos con el botón derecho sobre el nombre del paquete: *New/Java Enum*.

Ahora, si queremos guardar en una variable el día de la semana que tenemos clase de inglés (los martes), crearemos una variable de tipo `DiaDeLaSemana` y le asignaremos el valor correspondiente,

```
DiaDeLaSemana claseIngles = DiaDeLaSemana.MARTES;
```

Sin embargo, cuando tengamos que introducir un valor por teclado para asignarlo a una variable de tipo enumerado, escribiremos una cadena con el valor (`LUNES`, por ejemplo) sin necesidad de poner delante el nombre del tipo. Para asignarlo a la variable correspondiente, se usa el método `valueOf()`. Por ejemplo, si queremos introducir por teclado el día de la semana de la clase de francés (el jueves) para asignarlo a la variable `claseFrances`, de tipo `DiaDeLaSemana`, escribiremos: `"JUEVES"` y no `"DiaDeLaSemana.JUEVES"`. A la hora de asignarlo a la variable pondremos,

```
Scanner sc = new Scanner(System.in);
String dia = sc.nextLine(); //introducimos JUEVES
DiaDeLaSemana claseFrances = DiaDeLaSemana.valueOf(dia);
```

En la variable `claseFrances` se ha guardado el valor `DiaDeLaSemana.JUEVES`.

Para usar un tipo enumerado solo dentro de una clase, escribimos su definición dentro del código de la clase. Por ejemplo, si queremos añadir el atributo `Sexo` a la clase `Cliente`, definimos el tipo enumerado `Sexo`, con los valores `HOMBRE` y `MUJER` y, a continuación declaramos el atributo `sexo` del tipo `Sexo`,

```
class Cliente {
    enum Sexo {HOMBRE, MUJER} //definición del tipo
    Sexo sexo; //declaración del atributo

    Cliente(String dni, String nombre, int edad, Sexo sexo) {
        ...
    }
}
```

En este caso, para acceder al tipo `Sexo` desde código escrito fuera de la clase `Cliente`, por ejemplo, desde la función `main()`, tendremos que poner delante el nombre de la clase donde está definido,

```
Cliente c = new Cliente("123", "Nuria", 41, Cliente.Sexo.MUJER);
```

Ejercicios de clases

- 7.1. Diseñar la clase `CuentaCorriente`, sabiendo que los datos necesarios son: saldo, límite de descubierto⁷, nombre y DNI del titular. Las operaciones típicas con una cuenta corriente son:

Crear la cuenta: se necesita el nombre y DNI del titular. El saldo inicial será 0 y el límite de descubierto será de -50 euros.

Sacar dinero: solo se podrá sacar dinero hasta el límite de descubierto. El método debe indicar si ha sido posible llevar a cabo la operación.

Ingresar dinero: se incrementa el saldo.

Mostrar información: muestra la información disponible de la cuenta corriente.

Clase `CuentaCorriente`

```
/*
 * Esta clase permite gestionar la cuenta corriente de un cliente. Para crear una
 * cuenta se necesita al menos el nombre y el DNI del cliente. El saldo inicial
 * será de 0 euros, y la cuenta tiene asignado un límite para descubiertos:
 * la cuenta puede estar en números rojos hasta el límite asignado.
 */
class CuentaCorriente {

    double saldo; //efectivo disponible en la cuenta
    String nombre; //del titular
    String dni; //del titular
    double limite; //límite de descubierto

    /* Los parámetros de entrada: nombre y dni, ocultan a los atributos de la
     * clase con el mismo identificador. Para acceder a ellos hay que utilizar this */
    CuentaCorriente(String nombre, String dni) { //constructor
        saldo = 0; //asignamos el saldo por defecto
        this.nombre = nombre; //nombre pasado como parámetro
        this.dni = dni; //DNI pasado como parámetro
        limite = -50; //límite del descubierto por defecto
    }

    boolean egreso(double cant) { //sacar dinero de la cuenta corriente
        boolean operacionPosible;

        if ((saldo - cant) >= limite) { //si solicitamos sacar dinero dentro
            //del límite
            saldo -= cant; //no se permite realizar la operación.
            operacionPosible = true;
        } else {
            System.out.println("No hay dinero suficiente");
            operacionPosible = false;
        }

        return (operacionPosible); //indica si ha sido posible realizar la operación
    }

    //añadimos dinero a la cuenta corriente
    void ingreso(double cant) {
        saldo += cant;
    }
}
```

⁷Cantidad de dinero que se permite sacar de una cuenta que ya está a cero.

```

void mostrarInformacion() { //muestra el estado de la cuenta corriente
    System.out.println("Nombre: " + nombre);
    System.out.println("Dni: " + dni);
    System.out.println("Saldo: " + saldo);
    System.out.println("Limite descubierto: " + limite);
}
}

```

Programa principal

```

/*
 * Creamos un objeto CuentaCorriente para probar la clase y realizar algunas
 * operaciones de ejemplo.
 */
public class Main {

    public static void main(String[] args) {
        CuentaCorriente c;

        c = new CuentaCorriente("Pepe", "12345678A"); //CuentaCorriente para "Pepe"
        //con DNI "12.345.678-A"

        c.limite = -100; // establecemos el límite máximo para un descubierto

        c.ingreso(1000); // ingresamos 1000 euros
        c.egreso(300); // sacamos 300 euros. Quedarán 700 euros
        c.mostrarInformacion(); // mostramos

        System.out.println("Puedo sacar 700 euros: " + c.egreso(700)); //quedan 0 euros
        System.out.println("Puedo sacar 500 euros: " + c.egreso(500)); //no es posible
    }
}

```

7.2. En la clase `CuentaCorriente` sobrecargar los constructores para que permitan crear objetos,

- Solo con el saldo inicial, no serán necesarios los datos del titular. Por defecto el límite de descubierto será 0 euros.
- Con un saldo inicial, con un límite de descubierto y con el DNI del titular de la cuenta.

Solución a)

Clase `CuentaCorriente`

```

/*
 * Sobrecargamos los constructores
 */
class CuentaCorriente {
    ...

    /* Los parámetros de entrada: nombre y dni, ocultan a los atributos de la
     * clase con el mismo identificador. Para acceder a ellos hay que utilizar this */
    CuentaCorriente(String nombre, String dni) { //constructor
        saldo = 0; //asignamos el saldo por defecto
        this.nombre = nombre; //nombre pasado como parámetro
        this.dni = dni; //DNI pasado como parámetro
        limite = -50; //límite del descubierto por defecto
    }
}

```

```
CuentaCorriente(double saldo) { //crea una cuenta solo con el saldo
    this.saldo = 0; //el parámetro de entrada saldo oculta el atributo. Usamos this
    this.nombre = "Sin asignar"; //aunque no es necesario, utilizamos this
    this.dni = "Sin asignar";
    this.limite = 0;
}

CuentaCorriente(double saldo, double limite, String dni) {
    this.saldo = saldo;
    this.nombre = "Sin asignar";
    this.dni = dni;
    this.limite = limite;
}
}
```

Programa principal

```
//probamos los constructores
public class Main {

    public static void main(String[] args) {
        CuentaCorriente c1, c2, c3;

        c1 = new CuentaCorriente("Pepe", "12345678A"); //nombre y dni
        c2 = new CuentaCorriente(1000); //saldo
        c3 = new CuentaCorriente(1000, 50, "12345678A"); //saldo, límite y dni

        c1.mostrarInformacion();
        c2.mostrarInformacion();
        c3.mostrarInformacion();
    }
}
```

Solución b)

Clase CuentaCorriente

```
/*
 * Desde un constructor es posible invocar otro mediante el constructor this().
 */
class CuentaCorriente {
    ...

    /* Reutilizamos el constructor: CuentaCorriente(saldo, limite, dni), que no asigna el
    nombre. Es necesario asignar el nombre a posteriori */
    CuentaCorriente(String nombre, String dni) { //constructor
        this(0, -50, dni);
        this.nombre = nombre; //nombre pasado como parámetro
    }

    //Reutilizamos el constructor CuentaCorriente(saldo, limite, dni)
    CuentaCorriente(double saldo) { //crea una cuenta solo con el saldo
        this(saldo, 0, "Sin dni"); //this() tiene que ser la primera línea del constructor
    }

    CuentaCorriente(double saldo, double limite, String dni) {
        this.saldo = saldo;
        this.nombre = "Sin asignar";
        this.dni = dni;
        this.limite = limite;
    }
}
```

- 7.3. Para la clase **CuentaCorriente** escribir un programa que compruebe el funcionamiento de sus métodos, incluidos los constructores.

Programa principal

```
/*
 * Creamos un objeto CuentaCorriente para probar la clase y realizar algunas
 * operaciones de ejemplo.
 */
public class Main {

    public static void main(String[] args) {
        CuentaCorriente c;

        c = new CuentaCorriente("Pepe", "12345678-A"); //CuentaCorriente para "Pepe"
        //con DNI "12.345.678-A"

        c.limite = -100; // establecemos el límite máximo para un descubierto

        c.ingreso(1000); // ingresamos 1000 euros
        c.egreso(300); // sacamos 300 euros. Quedarán 700 euros
        c.mostrarInformacion(); // mostramos

        System.out.println("Puedo sacar 700 euros: " + c.egreso(700)); //quedan 0 euros
        System.out.println("Puedo sacar 500 euros: " + c.egreso(500)); //no es posible

        //vamos a probar el constructor que solo utiliza el saldo
        c = new CuentaCorriente(100); //c referencia al nuevo objeto, el anterior queda
        //sin referencia a merced del recolector de basura
        c.mostrarInformacion();

        //vamos a probar el constructor que utiliza el saldo, el límite de descubierto y
        // el nombre del titular
        c = new CuentaCorriente(2000, -300, "98765432-Z");
        c.mostrarInformacion();
    }
}
```

- 7.4. Modificar la visibilidad de la clase **CuentaCorriente** para que sea visible desde clases externas y la visibilidad de sus atributos para que:

- **saldo** y **limite** no sean visibles para otras clase.
- **nombre** sea público para cualquier clase.
- **dni** solo sea visible por clases vecinas.

Realizar un programa para comprobar la visibilidad de los atributos.

Clase **CuentaCorriente**

```
/*
 * Marcamos la clase con public, para que sea visible desde clase externas. Para
 * utilizar la clase CuentaCorriente tendrán que importarla. */
public class CuentaCorriente {
    private double saldo; //invisible para cualquier clase (vecina o externa)
    public String nombre; //visibilidad total
    String dni; //sin modificador (visibil. por defecto). Solo visible por clases vecinas
    private double limite; //invisible para cualquier clase (vecina o externa)

    ...
}
```

Programa principal

```
/*
 * La clase Main es una clase vecina de CuentaCorriente
 */
public class Main {

    public static void main(String[] args) {
        CuentaCorriente c;

        c = new CuentaCorriente("Pepe", "12345678-A"); //CuentaCorriente para "Pepe"
        //con DNI "12.345.678-A"

        c.saldo = 2000; //produce un error, ya que el saldo no es visible desde fuera de
        //la clase CuentaCorriente
        c.dni = "11111111-T"; //al ser Main una clase vecina, dni es visible
        //en caso de acceder al dni desde una clae externa produciría un error
        c.nombre = "Antonio"; //nombre es visible desde cualquier clase
        c.limite = -100; //atributo privado. No accesible desde fuera de su clase.
        //Genera un error
    }
}
```

- 7.5. Todas las cuentas corrientes con las que vamos a trabajar pertenecen al mismo banco. Añadir un atributo que almacene el nombre del banco (que es único) en la clase *CuentaCorriente*. Diseñar un método que permita modificar el nombre del banco (al que pertenecen todas las cuentas corrientes).

Clase CuentaCorriente

```
/*
 * Vamos a añadir el atributo banco a la clase. Como el banco es el mismo para todas
 * la cuentas el atributo será estático. También escribiremos un método estático
 * para modificar el nombre del banco
 */
public class CuentaCorriente {
    ...
    static String nombreBanco = "International Java Bank"; //valor por defecto
    //este valor se asigna antes de crear ningún objeto

    static void cambiarNombreBanco(String nuevoNombre) {
        nombreBanco = nuevoNombre;
    }
}
```

Programa principal

```
//La clase Main es vecina de la clase CuentaCorriente
public class Main {

    public static void main(String[] args) {
        CuentaCorriente c1, c2;

        c1 = new CuentaCorriente("Pepe", "12345678-A"); //CuentaCorriente para Pepe
        c2 = new CuentaCorriente("Ana", "999999999-E"); //cuenta de Ana

        c1.mostrarInformacion();
        c2.nombreBanco = "Banco Central";
        c1.mostrarInformacion();
        CuentaCorriente.nombreBanco = "Caja de Ahorros de Do-While";
    }
}
```

```

        c1.mostrarInformacion();
        CuentaCorriente.cambiarNombreBanco("If-Else Bank");
        c1.mostrarInformacion();
    }
}

```

7.6. Diseñar la clase `Texto` que gestiona una cadena de caracteres con algunas características:

- La cadena de caracteres tendrá una longitud máxima, que se especifica en el constructor.
- Permite añadir un carácter, al principio o al final, siempre y cuando exista espacio disponible.
- Igualmente, permite añadir una cadena, al principio o al final del texto, siempre y cuando no se rebase el tamaño máximo establecido.
- Es necesario saber cuántas vocales (mayúsculas y minúsculas) hay en el texto.

Clase `Texto`

```

/*
 * Implementaremos la clase Texto utilizando un String (donde guardaremos la
 * cadena de caracteres) y un número entero que indicará la longitud o tamaño
 * máximo del texto.
 */

public class Texto {
    private String cad; //cadena de caracteres
    private final int tamMax; //tamaño máximo del texto. Una vez asignado no varía
    static final String VOCALES = "aeiouáéíóü"; //cadena constante y estática que
    //contiene todas las posibles vocales en minúsculas

    public Texto(int tamMax) {
        cad = ""; //cad referencia un objeto String con valor "", no se puede utilizar
        // cad = null, en este caso cad no referencia ningún objeto y no es posible
        //utiliza sus métodos
        this.tamMax = tamMax; //tamaño máximo permitido para el texto
    }

    //Añade un carácter al final del texto, siempre y cuando quede sitio
    public void add(char c) {
        if (tamMax > cad.length()) {
            cad = cad + c; //concatena el carácter al final
        }
    }

    //Añade una cadena al final del texto, siempre y cuando quede sitio
    public void add(String c) {
        if (tamMax >= cad.length() + c.length()) {
            cad = cad + c;
        }
    }

    //Añade un carácter al comienzo del texto, siempre y cuando quede sitio
    public void addPrincipio(char c) {
        if (tamMax > cad.length()) {
            cad = c + cad;
        }
    }
}

```

```
//Añade una cadena al comienzo del texto, siempre y cuando quede sitio
public void addPrincipio(String c) {
    if (tamMax >= cad.length() + c.length()) {
        cad = c + cad;
    }
}

public void mostrar() {
    System.out.println(cad);
}

//Devuelve el número de vocales presentes en el texto
public int numVocales() {
    int voc = 0; // número de vocales del texto
    for (int i = 0; i < cad.length(); i++) {
        if (esVocal(cad.charAt(i))) {
            voc++;
        }
    }
    return (voc);
}

//Comprueba si el carácter pasado es una vocal: mayúscula/minúscula/acentuada
private boolean esVocal(char c) {
    boolean vocal = false;

    c = Character.toLowerCase(c); //convertimos el carácter a minúscula

    if (VOCALES.indexOf(c) != -1) { //buscamos el carácter (en minúscula) entre todas
        vocal = true;                //las posibles vocales
    }
    return (vocal);
}
}
```

Programa principal

```
// Creamos un objeto Texto para probar su funcionamiento.
public class Main {

    public static void main(String[] args) {
        Texto t = new Texto(5);

        t.addPrincipio("H0");
        t.addPrincipio(';');
        t.add("Lá");
        t.add('X'); // este carácter no cabe en el texto. No se añade.

        t.mostrar();
        System.out.println("Número de vocales: " + t.numVocales());
    }
}
```

7.7. Diseñar la clase Banco de la que interesa guardar su nombre, capital y la dirección central. Los bancos tienen las siguientes restricciones:

- a) Siempre tienen que tener un nombre, que no puede ser modificado.
- b) Si no se especifica, todos los bancos tienen un capital por defecto de 5.2 millones de euros al crearse.
- c) El capital y la dirección de un banco son modificables.

Modificar la clase **CuentaCorriente** para que cada una esté vinculada a un objeto de tipo **Banco**. Escribir los métodos necesarios en la clase **CuentaCorriente** para gestionar el banco al que pertenece. Existe la posibilidad de que una cuenta corriente no esté vinculada a ningún banco.

Clase Banco

```
/*
 * Para cumplir los requisitos de los bancos:
 * - Siempre tienen un nombre: todos los constructores lo asignarán
 * - Una vez creado no se puede cambiar el nombre: el atributo nombre será privado y
 *   no existirá un método que permita modificar el nombre
 * - Capital por defecto de 5.2 millones de euros: asignaremos este valor al atributo
 */
public class Banco {

    final private String nombre; //no es visible: no es modificable desde otras clases
    //La clase no tendrá ningún método que permita cambiar el nombre, que permanecerá
    //inmutable durante toda la vida del objeto.
    public double capital = 5.2; //fijamos el capital por defecto en 5.2 millones de euros
    public String direccion = "Sin dirección"; //dirección por defecto
    //no es necesario inicializar por defecto en los constructores: capital ni direccion

    public Banco(String nombre) { //crea un banco solo con el nombre
        this.nombre = nombre;
    }

    public Banco(String nombre, double capital, String direccion) { //con todos los datos
        this.nombre = nombre;
        this.capital = capital;
        this.direccion = direccion;
    }

    public void mostrarInformacion() {
        System.out.println("Banco: " + nombre);
        System.out.println("Capital: " + capital + " millones de euros");
        System.out.println("Dirección: " + direccion);
        System.out.println(""); //una línea en blanco
    }
}
```

Clase CuentaCorriente

```
/*
 * Cada CuentaCorriente tendrá una referencia a un objeto de tipo Banco.
 */
public class CuentaCorriente {
    ... //resto de atributos y métodos.
    Banco banco; //banco al que pertenece la cuenta

    CuentaCorriente(String nombre, String dni, Banco banco) { //sobrecargamos
        this(0, -50, dni);
        this.nombre = nombre; //nombre pasado como parámetro
        this.banco = banco;
    }

    //permite asignar un nuevo objeto Banco a la cuenta
    void cambiarBanco(Banco banco) {
        this.banco = banco;
    }

    void mostrarInformacion() { //muestra el estado de la cuenta, incluido el banco
        System.out.println("Información del banco");
    }
}
```

```
// no podemos hacer directamente banco.mostrarInformacion(), ya que puede
//ocurrir que banco sea null. Al intentar acceder a los miembros de un objeto
//nulo se produciría un error.
if (banco == null) { //si la cuenta no está asignada a ningún banco
    System.out.println("Cuenta no asociada a ningún banco");
} else {
    banco.mostrarInformacion(); //no es posible mostrar directamente sus atributos,
    //ya que algunos no son visibles
}
System.out.println("Información de la cuenta");
System.out.println("Titular: " + nombre);
System.out.println("Dni: " + dni);
System.out.println("Saldo: " + saldo);
System.out.println("Limite descubierto: " + limite);
}
}
```

Programa principal

```
public class Main {

    public static void main(String[] args) {
        CuentaCorriente c1, c2, c3;
        Banco b1, b2;

        //creamos dos objetos banco
        b1 = new Banco("International Java Bank");
        b2 = new Banco("Caja de Ahorros de Do-While", 10.6, "c/Larga s/n");

        //creamos varias cuentas
        c1 = new CuentaCorriente("Pepita", "12345678-A", b1); //cuenta asociada a b1
        c2 = new CuentaCorriente("Ana", "98765432-Z", b1); //otra cuenta de b1
        c1.mostrarInformacion();
        c2.mostrarInformacion();

        c3 = new CuentaCorriente("Sancho", "11222333-B"); //cuenta sin banco
        c3.mostrarInformacion();
        c3.cambiarBanco(b2); //asignamos la cuenta a b2
        c3.mostrarInformacion();
    }
}
```

- 7.8. Se quiere definir una clase que permita controlar un sintonizador digital de emisoras FM; concretamente, se desea dotar al controlador de una interfaz que permita subir (up) o bajar (down) la frecuencia (en saltos de 0.5 MHz) y mostrar la frecuencia sintonizada en un momento dado (display). Supondremos que el rango de frecuencias a manejar oscila entre los 80 Mhz y los 108 MHz y que, al inicio, el controlador sintonice a 80 MHz. Si durante una operación de subida o bajada se sobrepasa uno de los dos límites, la frecuencia sintonizada debe pasar a ser la del extremo contrario. Escribir un pequeño programa principal para probar su funcionamiento.

Clase SintonizadorFM

```
/*
 * La clase tiene un atributo real que almacena la frecuencia a la que estamos
 * sintonizando, junto a los métodos necesarios para utilizar el sintonizador. */
public class SintonizadorFM {
    private double frecuencia;
```

```

SintonizadorFM() { //constructor
    frecuencia = 80; // MHz. Frecuencia inicial
    // otra forma es invocar al constructor que permite asignar una frecuencia.
    // La siguiente instrucción debe ser la primera en el constructor actual:
    // this(80);
}

// constructor que permite asignar una frecuencia inicial
SintonizadorFM(double frecuenciaInicial) {
    // la frecuencia inicial debe encontrarse en el rango [80 - 108]
    if (frecuenciaInicial < 80) {
        frecuencia = 80; // MHz
    } else if (frecuenciaInicial > 108) {
        frecuencia = 108; //MHz
    } else {
        frecuencia = frecuenciaInicial;
    }
}

public double down() {
    frecuencia -= 0.5; // bajamos la frecuencia 0.5 MHz
    comprobarRango(); //y comprobamos

    return (frecuencia);
}

public double up() {
    frecuencia += 0.5; //subimos la frecuencia
    comprobarRango(); //y comprobamos

    return (frecuencia);
}

public void display() {
    System.out.println("Sintonizando: " + frecuencia + " MHz"); //mostramos
}

//método de uso interno que se encarga de comprobar que la frecuencia se encuentre
//en el rango 80..108. En caso de que la frecuencia esté fuera de rango la ajusta
private void comprobarRango() {
    if (frecuencia < 80) { //si al bajar la frecuencia es menor que el límite inferior
        frecuencia = 108; //asignamos el límite superior
    } else if (frecuencia > 108) { //si sobrepasamos el límite superior
        frecuencia = 80; //colocamos la frecuencia en el valor menor
    }
}
}

```

Programa principal

```

// Probamos el uso del sintonizador de FM
public class Main {

    public static void main(String[] args) {
        SintonizadorFM a = new SintonizadorFM(107);
        a.up(); a.up(); a.up(); a.up(); // subimos un total de 2 MHz
        a.display(); // debe mostrar 80.5 MHz
        SintonizadorFM b = new SintonizadorFM(80.5);
        b.down(); b.down(); b.down(); //bajamos 1.5 MHz
        b.display(); // debe mostrar 107.5 MHz
        a = new SintonizadorFM(200); //frecuencia fuera de rango. Debe ajustarse
        a.display(); //debe mostrar 108.0 MHz
    }
}

```

- 7.9. Modelar una casa con muchas bombillas, de forma que cada bombilla se pueda encender o apagar individualmente. Para ello hacer una clase **Bombilla** con una variable privada que indique si está encendida o apagada, así como un método que nos diga el estado de una bombilla concreta. Además, queremos poner un interruptor general, de forma que si saltan los fusibles, todas las bombillas quedan apagadas. Cuando el fusible se repara, las bombillas vuelven a estar encendidas o apagadas, según estuvieran antes del percance. Cada bombilla se enciende y se apaga individualmente, pero solo responde que está encendida si su interruptor particular está activado y además hay luz general.

Clase Bombilla

```
/*
 * La clase Bombilla se implementa con un indicador (apagada/encendida) de estado, que
 * será individual para cada bombilla (para cada objeto bombilla). Además disponemos de
 * un interruptor general (que afecta a todas las bombillas). Este se implementa mediante
 * un atributo estático, cuyo valor será el mismo para todos los objetos de la clase. */
public class Bombilla {
    public static boolean interruptorGeneral = true; // atributo estático
    private boolean interruptor; //interruptor (estado) que posee cada bombilla

    public Bombilla() {
        interruptor = false; // inicialmente la nueva bombilla está apagada
    }

    public void enciende() {
        interruptor = true; // activamos el interruptor (a true)
    }

    public void apaga() {
        interruptor = false; // desactivamos el interruptor
    }

    public boolean estado() {
        boolean est;
        if (interruptorGeneral == true && interruptor == true) {
            // si el interruptor de la bombilla está activado y hay luz general
            est = true;
        } else {
            est = false;
        }
        return (est);
    }

    //devuelve una cadena con el estado de la bombilla
    public String muestraEstado() {
        String estado;

        if (estado() == true) {
            estado = "Encendida";
        } else {
            estado = "Apagada";
        }

        return (estado);
    }
}
```

Programa principal

```
public class Main {
    // vemos un ejemplo de funcionamiento
```

```

public static void main(String[] args) {
    Bombilla b1, b2;
    b1 = new Bombilla();
    b2 = new Bombilla();

    b1.enciende();
    b2.apaga();

    System.out.println("b1: " + b1.muestraEstado());
    System.out.println("b2: " + b2.muestraEstado());

    Bombilla.interruptorGeneral = false; // cortamos la luz
    System.out.println("\nCortamos la luz general");
    System.out.println("b1: " + b1.muestraEstado());
    System.out.println("b2: " + b2.muestraEstado());

    Bombilla.interruptorGeneral = true; // activamos la luz
    System.out.println("\nActivamos la luz general");
    System.out.println("b1: " + b1.muestraEstado());
    System.out.println("b2: " + b2.muestraEstado());
}
}

```

7.10. Hemos recibido un encargo para definir los paquetes y las clases necesarias (solo implementar los atributos y los constructores) para gestionar una empresa ferroviaria, en la que se distinguen dos grandes grupos: el personal y la maquinaria. En el primero se ubican todos los empleados de la empresa, que se clasifican en tres grupos: los maquinistas, los mecánicos y los jefes de estación. De cada uno de ellos es necesario guardar:

Maquinistas: su nombre completo, su documento nacional de identidad, su sueldo mensual y el rango que tienen adquirido.

Mecánicos: su nombre completo, teléfono (para contactar en caso de urgencia) y en qué especialidad desarrollan su trabajo (frenos, hidráulica, electricidad, etc.)

Jefes de estación: su nombre completo y DNI.

En la parte de maquinaria podemos encontrar trenes, locomotoras y vagones. De cada uno de ellos hay que considerar:

Vagones: tienen una capacidad máxima de carga (en kilos), una capacidad actual (de la carga que tienen en un momento dado) y el tipo de mercancía con el que están cargados.

Locomotoras: disponen de una matrícula (que las identifica), la potencia de sus motores y una antigüedad (año de fabricación). Además, cada locomotora tiene asignado un mecánico que se encarga de su mantenimiento.

Trenes: están formados por una locomotora y un máximo de 5 vagones. Cada tren tiene asignado un maquinista que es responsable de él.

Todas las clases correspondientes al personal (**Maquinista**, **Mecanico** y **JefeEstacion**) serán de uso público. Entre las clases relativas a la maquinaria solo será posible construir, desde clases externas, objetos de tipo **Tren** y de tipo **Locomotora**. La clase **Vagon** será solo visible por sus clases vecinas.

Clase Maquinista

```
package personal;

public class Maquinista {
    String nombre;
    String dni;
    double sueldo;
    String rango;

    public Maquinista(String nombre, String dni, double sueldo, String rango) {
        this.nombre = nombre;
        this.dni = dni;
        this.sueldo = sueldo;
        this.rango = rango;
    }
}
```

Clase Mecanico

```
package personal;

public class Mecanico {
    String nombre;
    String telefono;
    String especialidad;

    public Mecanico(String nombre, String telefono, String especialidad) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.especialidad = especialidad;
    }
}
```

Clase JefeEstacion

```
package personal;

public class JefeEstacion {
    String nombre;
    String dni;

    public JefeEstacion(String nombre, String dni) {
        this.nombre = nombre;
        this.dni = dni;
    }
}
```

Clase Vagon

```
package maquinaria;

/*
 * Sin modificar de acceso en la clase: visibilidad por defecto.
 */
class Vagon {
    int capacidadMax;
    int capacidadActual;
    String mercancia;
}
```

```

        public Vagon(int capacidadMax, int capacidadActual, String mercancia) {
            this.capacidadMax = capacidadMax;
            this.capacidadActual = capacidadActual;
            this.mercancia = mercancia;
        }
    }
}

```

Clase Locomotora

```

package maquinaria;

import personal.Mecanico;

public class Locomotora {
    //atributos con visibilidad por defecto. Solo visible por clases vecinas
    String matricula;
    int potencia;
    int añoFabricacion;
    Mecanico mec;

    public Locomotora(String matricula, int potencia, int añoFabricacion, Mecanico mec) {
        this.matricula = matricula;
        this.potencia = potencia;
        this.añoFabricacion = añoFabricacion;
        this.mec = mec;
    }
}

```

Clase Tren

```

package maquinaria;

import personal.Maquinista;

public class Tren {
    Locomotora locomotora;
    Vagon vagones[];
    Maquinista responsable;
    private int numVagones; //número de vagones que forman el tren

    public Tren(Locomotora locomotora, Maquinista responsable) {
        this.locomotora = locomotora;
        this.responsable = responsable;

        // creamos la tabla de tamaño 5, pero no se crea
        // ningún objeto de tipo Vagón
        vagones = new Vagon[5];
        numVagones = 0; //por ahora no hay vagones enganchados al tren
    }

    /* Al ser la clase Vagon no visible por clases externas, será la clase Tren la
     * que se encargue de construir el objeto a partir de los datos que nos pasen. */
    public void enganchaVagon(int capacidadMax, int capacidadActual, String mercancia) {
        if (numVagones < 5) {
            System.out.println("El tren no admite más vagones");
        } else {
            Vagon v = new Vagon (capacidadMax, capacidadActual, mercancia);
            vagones[numVagones] = v; //el vagón pasado ocupa el último lugar
            numVagones ++; //ahora tenemos un vagón más enganchado al tren
        }
    }
}

```

- 7.11. Escribir un programa que lea por teclado una hora cualquiera y un número **n** que representa una cantidad en segundos. El programa mostrará la hora introducida y las **n** siguientes horas que se diferencian en un segundo. Para ello hemos de diseñar previamente la clase **Hora** que dispone de los atributos **hora**, **minuto** y **segundo**. Los valores de los atributos se controlará mediante métodos **set/get**.

Clase **Hora**

```
/*
 * La clase Hora es muy simple y dispone de los atributos: hora, minuto y segundo.
 * Estos serán privados y para darles visibilidad utilizaremos métodos set/get.
 * Internamente vamos a utilizar el tipo byte para almacenar los atributos, pero desde
 * fuera de la clase nos interesa dar la sensación que los atributos son int: estamos
 * encapsulando (ocultando) la verdadera implementación.
 * No escribimos ningún constructor.
 */
public class Hora {
    private byte hora; //atributos de tipo byte: más que suficiente para los valores
    private byte minuto; //que tenemos que guardar
    private byte segundo;

    public int getHora() {
        return hora; //devuelve la hora
    }

    public void setHora(int hora) {
        if (0 <= hora && hora <= 23) { //la hora está comprendida en el rango 0..23
            this.hora = (byte) hora;
        } else {
            this.hora = 0; //si el valor está fuera de rango, lo ponemos a 0
        }
    }

    public int getMinuto() {
        return minuto; //devuelve los minutos
    }

    public void setMinuto(int minuto) { //los minutos están comprendidos de 0..59
        if (0 <= minuto && minuto <= 59) {
            this.minuto = (byte) minuto;
        } else {
            this.minuto = 0; //si el valor está fuera de rango lo ponemos a 0
        }
    }

    public byte getSegundo() {
        return segundo; //devuelve los segundos
    }

    public void setSegundo(int segundo) { //los segundos están comprendidos: 0..59
        if (0 <= segundo && segundo <= 59) {
            this.segundo = (byte) segundo;
        } else {
            this.segundo = 0; //si el valor está fuera de rango lo ponemos a 0
        }
    }

    public void incrementaSegundo() {
        segundo++; //incrementamos los segundos
        if (segundo == 60) { //si los segundos alcanza un valor de 60
            segundo = 0; //reiniciamos los segundos
            minuto++; //e incrementamos los minutos
            if (minuto == 60) { //si los minutos alcanza un valor de 60
                minuto = 0; //reiniciamos los minutos
                hora++; //e incrementamos las horas
            }
        }
    }
}
```



```

        if (hora == 24) { //si la hora alcanza un valor 24
            hora = 0; //reiniciamos las horas
        }
    }
} //del método
} //de la clase

```

Programa principal

```

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Hora h = new Hora(); //creamos un objeto Hora

        System.out.println("Hora: ");
        int valor = sc.nextInt(); //leemos un valor para la hora
        h.setHora(valor); //asignamos un valor para la hora

        System.out.println("Minuto: ");
        valor = sc.nextInt(); //leemos un valor para los minutos
        h.setMinuto(valor); //asignamos un valor a los minutos

        System.out.println("Segundo: ");
        valor = sc.nextInt(); //leemos un valor para los segundos
        h.setSegundo(valor); //asignamos un valor a los segundos

        System.out.println("Cuántos segundos quiere mostrar: ");
        int numSegundos = sc.nextInt();

        for (int i = 0; i <= numSegundos; i++) {
            //mostramos la hora
            System.out.println(h.getHora() + ":" + h.getMinuto() + ":" + h.getSegundo());
            h.incrementaSegundo(); //incrementamos la hora actual en un segundo
        }
    }
}

```

7.12. Las listas son estructuras dinámicas de datos, donde se pueden insertar o eliminar elementos de un determinado tipo sin limitación de espacio. Implementar la clase **Lista** correspondiente a una lista de números de la clase **Integer** (véase Anexo B al final del libro). Los números se guardarán en una tabla. Cuando falte espacio para un nuevo elemento, la tabla se redimensionará, incrementando la capacidad de la lista. Entre los métodos de la clase, se incluirán:

- Dos constructores, uno por defecto que cree la tabla con capacidad para 10 números, y otro al que se le pasa como argumento el tamaño inicial de la tabla.
- Obtener el número de elementos insertados en la lista.
- Insertar un número al final de la lista.
- Insertar un número al principio de la lista.
- Insertar un número en un lugar de la lista cuyo índice se pasa como parámetro.
- Añadir al final de la lista los elementos de otra lista que se pasa como parámetro.

- Eliminar un elemento cuyo índice en la lista se pasa como parámetro.
- Obtener el elemento cuyo índice se pasa como parámetro.
- Buscar un número en la lista, devolviendo el índice del primer lugar donde se encuentre. Si no está, devolver -1.
- Representar la lista con una cadena de caracteres.

Clase Lista

```
import java.util.Arrays;

/*
 * Implementamos las listas de tipo Integer con tablas, que iremos redimensionando según
 * vaya haciendo falta. Los elementos de la lista se insertan desde el principio de la
 * tabla hacia el final. El número de elementos insertados se refleja en el atributo
 * numeroElementos, que son los elementos válidos de la tabla. El índice de un elemento
 * en la lista coincide con el índice del lugar que ocupa en la tabla. */

public class Lista {
    private int numeroElementos;
    private Integer[] tabla;

    //Por defecto, la capacidad inicial de la lista (tamaño de la lista) es 10
    public Lista() {
        numeroElementos = 0;
        tabla = new Integer[10];
    }

    //Podemos hacer que la capacidad inicial sea cualquier valor pasándosela
    //al constructor
    public Lista(int capacidadInicial) {
        numeroElementos = 0;
        tabla = new Integer[capacidadInicial];
    }

    //El número de elementos de la lista puede leerse desde fuera de la clase
    //aunque no modificarse
    int numeroElementos() {
        return numeroElementos;
    }

    //Método de uso interno en la clase
    boolean listaLlena() { //No hay elementos libres en la tabla
        return numeroElementos == tabla.length;
    }

    void insertarPrincipio(Integer nuevo) {
        if (listaLena()) { //Si la tabla está llena, aumentamos en 10 su tamaño
            tabla = Arrays.copyOf(tabla, tabla.length + 10);
        }
        for (int i = numeroElementos; i >= 1; i--) { //desplazamos hacia el final
            tabla[i] = tabla[i - 1];
        }
        tabla[0] = nuevo;
        numeroElementos++;
    }

    void insertarFinal(Integer nuevo) {
        if (listaLena()) {
            tabla = Arrays.copyOf(tabla, tabla.length + 10);
        }
        tabla[numeroElementos] = nuevo;
        numeroElementos++;
    }
}
```

```

void insertarFinal(Lista otraLista) {
    //Aumentamos la capacidad hasta que quepan los elementos de las dos listas
    while (numeroElementos + otraLista.numeroElementos > tabla.length) {
        tabla = Arrays.copyOf(tabla, tabla.length + 10);
    }
    for (int i = numeroElementos, j = 0; j < otraLista.numeroElementos; i++, j++) {
        tabla[i] = otraLista.tabla[j];
        numeroElementos++;
    }
}

//El primer parámetro es el índice del lugar donde queremos insertar el nuevo
//elemento, que pasamos en el segundo parámetro
boolean insertar(int posicion, Integer nuevo) {
    boolean insertado = true;

    if (posicion < 0 || posicion > numeroElementos) { //índice no válido
        insertado = false;
    } else {
        if (listaLlena()) {
            tabla = Arrays.copyOf(tabla, tabla.length + 10);
        }
        for (int i = numeroElementos; i > posicion; i--) {
            tabla[i] = tabla[i - 1];
        }
        tabla[posicion] = nuevo;
        numeroElementos++;
    }

    return insertado;
}

//Se elimina el elemento correspondiente al parámetro índice y se devuelve
Integer eliminar(int indice) {
    Integer eliminado = null;

    if (indice >= 0 && indice < numeroElementos) {
        eliminado = tabla[indice];
        for (int i = indice; i < numeroElementos - 1; i++) {
            tabla[i] = tabla[i + 1];
        }
        numeroElementos--;
    }
    return eliminado;
}

/* Al siguiente método le pasaremos un índice y nos devolverá el elemento correspon-
diente de la tabla sin modificarla. En el caso de que el índice no sea válido,
devolverá null, con lo cual evitamos que el programa aborte. */
Integer get(int indice) {
    Integer resultado = null;
    if (indice >= 0 && indice < numeroElementos) { //índice válido
        resultado = tabla[indice];
    }

    return resultado;
}

int buscar(Integer claveBusqueda) {
    int indice = -1; //si no se encuentra devolvemos -1
    for (int i = 0; i < numeroElementos && indice == -1; i++) {
        if (tabla[i].equals(claveBusqueda)) {
            indice = i; //posición donde se encuentra claveBusqueda
        }
    }
    return indice;
}

```

```
void ordenar(){
    Arrays.sort(tabla,0,numeroElementos);
}

// El método mostrar devuelve una cadena que representa a la lista
public String mostrar() {
    String res = "";
    for (int i = 0; i < numeroElementos; i++) {
        res += tabla[i] + " ";
    }
    return res;
}
}
```

Programa principal

```
public class Main {
    //prueba de los métodos de la clase Lista
    public static void main(String[] args) {
        Lista l = new Lista();
        l.insertarFinal(3);
        l.insertarFinal(2);
        l.insertarFinal(1);
        l.insertarFinal(4);
        l.insertarFinal(5);
        System.out.println(l.mostrar());

        l.insertarPrincipio(0);
        l.insertarPrincipio(1);
        l.insertarPrincipio(2);
        l.insertarPrincipio(3);
        l.insertarPrincipio(4);
        System.out.println(l.mostrar());

        l.insertar(2, 10);
        System.out.println(l.mostrar());
        l.eliminar(2);
        System.out.println(l.mostrar());
    }
}
```

7.13. Una pila es una estructura dinámica de datos donde los elementos se insertan (se apilan) y se retiran (se desapilan) siguiendo la norma de que el último que se apila será el primero en desapilarse, como ocurre con una pila de platos: cuando vamos a retirar un plato de una pila a nadie se le ocurre tirar de uno de los de abajo; retiramos (desapilamos) el que está encima de todos, que fue el último en ser apilado. Se llama cima de la pila al último elemento apilado (o al primer elemento a desapilar). Las funciones fundamentales de una pila son, por tanto, **apilar()** y **desapilar()**.

Implementar la clase **PilaTabla** para números **Integer**, donde se usa una tabla para guardar los elementos apilados.

Clase PilaTabla

```
import java.util.Arrays;

/* Vamos a implementar una estructura pila para valores Integer utilizando una tabla,
 * donde se almacenarán los elementos apilados. Las dos funciones fundamentales de una
```

```

* pila son apilar() y desapilar(), que consisten en añadir o quitar elementos de un
* mismo punto, llamado indiceCima, de manera que se desapila siempre el último elemento
* que se apiló. La tabla se redimensionará cuando falte sitio para elementos nuevos.
* Nosotros añadiremos un par de métodos que nos permitan visualizar la pila y algunos
* más, privados, para uso interno de la pila. */
public class PilaTabla {

    private int indiceCima; //índice del último elemento apilado
    private Integer[] tabla; //tabla que contendrá los elementos de la pila

    public PilaTabla() {
        indiceCima = -1; //corresponde a una pila vacía
        tabla = new Integer[10]; //por defecto creamos una pila de capacidad 10
    }

    //constructor que crea una PilaTabla con una capacidad inicial determinada
    public PilaTabla(int capacidadInicial) {
        tabla = new Integer[capacidadInicial]; //crea la tabla con el tamaño especificado
    }

    private boolean pilaVacía() {
        return indiceCima == -1; //un valor -1 indica que no existen datos en la pila
    }

    private boolean pilaLlena() {
        return indiceCima == tabla.length - 1; //si no caben más datos en la tabla
    }

    //Devuelve el elemento correspondiente a la cima o null si la pila está vacía
    Integer cima() {
        Integer elementoCima = null;
        if (!pilaVacía()) { //si la pila contiene al menos un elemento
            elementoCima = tabla[indiceCima]; //último elemento apilado
        }
        return elementoCima;
    }

    //Apila añadiendo el elemento en el primer lugar libre de la tabla,
    //empezando por el principio (índice 0)
    void apilar(Integer elemento) {
        if (pilaLlena()) {
            tabla = Arrays.copyOf(tabla, tabla.length + 10);
        }
        indiceCima++; //siempre es el índice del último elemento apilado
        tabla[indiceCima] = elemento;
    }

    //Desapila extrayendo el elemento de la cima. Si la pila está vacía, devuelve null
    Integer desapilar() {
        Integer elemento = null;
        if (!pilaVacía()) {
            elemento = tabla[indiceCima]; //último elemento apilado
            indiceCima--; //ahora tenemos un elemento menos en la pila
        }
        return elemento;
    }

    public String mostrar() {
        String resultado = "";
        for (int i = 0; i <= indiceCima; i++) {
            resultado += tabla[i] + " ";
        }
        resultado += "(cima)";
        return resultado;
    }
}

```

Programa principal

```
public class Main {  
  
    //programa principal para probar la clase PilaTabla  
    public static void main(String[] args) {  
        PilaTabla p = new PilaTabla();  
        for (int i = 0; i < 10; i++) { //apilamos los números del 0 al 9  
            p.apilar(i);  
        }  
  
        Integer num = p.desapilar(); //desapilamos  
        while (num != null) { ///mientras la pila no esté vacía  
            System.out.println(num); //mostramos  
            num = p.desapilar(); //y volvemos a desapilar  
        }  
    }  
}
```

- 7.14. Repetir el ejercicio anterior para implementar la clase `PilaLista`, donde, en vez de usar una tabla para almacenar los valores apilados, usamos un objeto de la clase `Lista` implementada en el Ejercicio 7.12.

Clase `PilaLista`

```
/*  
 * Vamos a implementar una estructura de pila para Integer usando objetos de la clase  
 * Lista para guardar los datos que se apilan. Por razón de eficiencia, la cima será el  
 * final de la lista, evitando así mover los datos apilados previamente.  
 */  
public class PilaLista {  
    private int indiceCima; //índice del último elemento apilado  
    private Lista lista; //objeto donde almacenaremos los datos  
  
    public PilaLista() {  
        indiceCima = -1; //corresponde a una pila vacía  
        lista = new Lista(); //por defecto creamos una pila de capacidad 10  
    }  
  
    public PilaLista(int capacidadInicial) {  
        indiceCima = -1;  
        lista = new Lista(capacidadInicial); //si queremos inicializar con una  
        //capacidad inicial distinta  
    }  
  
    //devuelve true si la pila está vacía y false en caso contrario  
    private boolean pilaVacía() {  
        return indiceCima == -1; //si indiceCima es -1 significa que la pila está vacía  
    }  
  
    // la pila estará llena si la lista lo está  
    private boolean pilaLlena() {  
        return lista.listaLlena();  
    }  
  
    //devuelve el último elemento (Integer) apilado  
    Integer cima() {  
        Integer elementoCima = null;  
        if (!pilaVacía()) { //si la pila no está vacía  
            elementoCima = lista.get(indiceCima); //obtenemos el último elemento insertado  
        }  
        return elementoCima;  
    }  
}
```

```
//apilamos añadiendo el elemento al final de la lista
void apilar(Integer elemento) {
    lista.insertarFinal(elemento);
    indiceCima++; //ahora tenemos un dato más en la pila
}

//desapilamos extrayendo el elemento de la cima. Si la pila está vacía, es
//porque la lista también lo está y devuelve null
Integer desapilar() {
    Integer elemento = lista.eliminar(indiceCima);
    if (elemento != null) {
        indiceCima--; //ahora tenemos un elemento menos en la pila
    }
    return elemento;
}

//muestra por pantalla directamente la pila
public String mostrar() {
    return lista.mostrar() + "(cima)"; //llamamos a mostrar() de Lista
}
}
```

Programa principal

```
public class Main {
    //programa principal para probar la clase PilaLista
    public static void main(String[] args) {
        PilaLista p = new PilaLista();
        for (int i = 0; i < 10; i++) { //apilamos los números del 0 al 9
            p.apilar(i);
        }
        Integer num = p.desapilar(); //desapilamos
        while (num != null) { //mientras no esté la pila vacía
            System.out.println(num); //mostramos
            num = p.desapilar(); //y volvemos a leer
        }
    }
}
```

- 7.15.** Una cola es otra estructura dinámica como la pila, donde los elementos, en vez de apilarse y desapilarse, se encolan y desencolan. La diferencia es que se desencola el primer elemento encolado y no el último, como en las colas del autobús o del cine. El primero que llega es el primero que sale de la cola (vamos a suponer que nadie se cuela). Por tanto, los elementos se encolan y desencolan en extremos opuestos de la estructura, llamados primero (el que está primero y será el próximo en abandonar la cola) y último (el que llegó último). Se pide implementar la clase ColaTabla en la que los elementos `Integer` encolados se guardan en una tabla.

Clase ColaTabla

```
import java.util.Arrays;

/* Implementamos una estructura cola para Integer. A diferencia de las pilas en las colas
 * los primeros elementos que entran son los primeros en salir, como en las colas del
 * cine. Por tanto, hay dos posiciones a tener en cuenta: el principio de la cola, de
 * donde se desencola (se extraen los elementos), y el final de la cola, donde se encola
 * (se insertan los elementos que llegan nuevos). Aquí la vamos a implementar con una
 * tabla donde se guardarán los elementos encolados. El primer elemento de la cola será
```

```
* el elemento de índice 0 de la tabla, con lo que solo tendremos que guardar el índice
* del último elemento de la cola. Las operaciones básicas de una cola son encolar y
* desencolar, aunque añadiremos otras auxiliares. */
public class ColaTabla {

    private int ultimo; //índice del último elemento encolado
    private Integer[] tabla; //tabla que almacena los datos

    public ColaTabla() {
        ultimo = -1; //cola vacía
        tabla = new Integer[10]; //por defecto, la capacidad inicial es 10
    }

    public ColaTabla(int capacidadInicial) {//si queremos otra capacidad inicial
        ultimo = -1;
        tabla = new Integer[capacidadInicial];
    }

    boolean colaVacía() {
        return ultimo == -1; //un valor -1 en ultimo significa una cola vacía
    }

    boolean colaLlena() {
        return ultimo == tabla.length - 1; //cuando no quepan más elementos en la tabla
    }

    void encolar(Integer elemento) {
        if (colaLlena()) {//si falta sitio para encolar, aumento la capacidad
            tabla = Arrays.copyOf(tabla, tabla.length + 10); //redimensiona la tabla
        }
        ultimo++; //ahora tenemos un elemento más en la cola
        tabla[ultimo] = elemento; //el nuevo elemento se sitúa el último
    }

    Integer desencolar() {
        Integer elemento = null;

        if (!colaVacía()) {
            elemento = tabla[0]; //al desencolar el primero, deben desplazarse
            //un lugar hacia el principio todos los demás
            for (int i = 0; i < ultimo; i++) {
                tabla[i] = tabla[i + 1];
            }
            ultimo--;
        }
        return elemento;
    }

    public String mostrar() {
        String resultado = "(primero) ";
        for (int i = 0; i <= ultimo; i++) {
            resultado += tabla[i] + " ";
        }
        resultado += "(ultimo)";

        return resultado;
    }
}
```

Programa principal

```
public class Main {
    //programa principal para probar la clase ColaTabla
    public static void main(String[] args) {
        ColaTabla c = new ColaTabla();
    }
}
```



```

        for (int i = 0; i < 10; i++) { //encolamos los números del 0 al 9
            c.encolar(i);
        }

        Integer num = c.desencolar(); //desencolamos
        while (num != null) { //mientras la cola no esté vacía
            System.out.println(num); //mostramos
            num = c.desencolar(); //y desencolamos de nuevo
        }
    }
}

```

7.16. Repetir el problema anterior, usando una Lista para guardar los elementos encolados.

Clase ColaLista

```

/* Vamos a implementar una cola (Integer) usando las funcionalidades de la clase Lista.
 * Encolaremos al final de la lista y desencolaremos del principio. El control del
 * espacio disponible y de la cola llena o vacía se deja en manos de la clase Lista.
 */

public class ColaLista {
    private Lista lista; //objeto donde se almacenan los datos

    public ColaLista() {
        lista = new Lista(); //creamos una lista con la capacidad por defecto
    }

    public ColaLista(int capacidadInicial) { //se especifica la capacidad inicial
        lista = new Lista(capacidadInicial);
    }

    void encolar(Integer elemento) {
        lista.insertarFinal(elemento); //encolamos al final de la lista
    }

    Integer desencolar() {
        return lista.eliminar(0); //desencolamos del principio
    }

    public String mostrar() {
        return "(primero) " + lista.mostrar() + "(último)";
    }
}

```

Programa principal

```

public class Main {

    //programa principal para probar el funcionamiento de los métodos de ColaLista
    public static void main(String[] args) {
        ColaLista c = new ColaLista();
        for (int i = 0; i < 10; i++) {
            c.encolar(i);
        }
        Integer num = c.desencolar();
        while (num != null) {
            System.out.println(num);
            num = c.desencolar();
        }
    }
}

```

7.17. Un conjunto es una estructura dinámica de datos como la lista, con dos diferencias: en primer lugar, en una lista puede haber elementos repetidos, mientras que en un conjunto, no. Además, en una lista el orden de inserción de los elementos puede ser relevante, mientras que en un conjunto solo interesa si un elemento pertenece o no al conjunto y no el lugar que ocupa. Se pide implementar la clase **Conjunto** utilizando una lista para almacenar números **Integer**. En particular, implementar los siguientes métodos:

- Dos constructores, uno por defecto que cree la tabla con capacidad para 10 números, y otro al que se le pasa como argumento el tamaño inicial de la tabla.
- Obtener el número de elementos insertados en el conjunto.
- Insertar un número en el conjunto. Si ya estaba, no se inserta.
- Añadir al conjunto los elementos de otro que se pasa como parámetro.
- Eliminar un elemento que se pasa como parámetro. Si no estaba, no se hace nada.
- Eliminar del conjunto los elementos de otro que se pasa como parámetro.
- Decir si un elemento que se le pasa como parámetro pertenece o no al conjunto.
- Representar el conjunto con una cadena de caracteres.

Clase Conjunto

```
/* Esta implementación de ConjuntoLista usa una lista, en vez de una tabla, como
 * estructura para almacenar los números. Con ello tiene acceso a todas las funcionalida-
 * des de las listas, incluidas las de inserción, eliminación y búsqueda. Solo hay que
 * tener en cuenta que en los conjuntos no se insertan elementos que ya estaban y que, al
 * no importar el orden, podemos insertar los números siempre al final, donde es menos
 * costoso en términos de computación. Es un ejemplo de reutilización de código.*/
public class Conjunto {
    private Lista lista; // los números se insertarán en esta lista

    public Conjunto() {
        lista = new Lista();
    }

    public Conjunto(int capacidadInicial) {
        lista = new Lista(capacidadInicial);
    }

    int numeroElementos() {
        return lista.numeroElementos();
    }

    boolean pertenece(Integer elemento) {
        return lista.buscar(elemento) >= 0; // si no está buscar() devuelve -1
    }

    boolean insertar(Integer nuevo) {
        boolean insertado = false;
        if (!pertenece(nuevo)) {
            lista.insertarFinal(nuevo);
            insertado = true; // true si nuevo no estaba y se ha insertado
        }
        return insertado;
    }

    boolean insertar(Conjunto otroConjunto) {
        boolean modificado = false;
```

```

        for (int i = 0; i < otroConjunto.lista.numeroElementos(); i++) {
            boolean insertado = insertar(otroConjunto.lista.get(i));
            if (insertado) {
                modificado = true; //true si el conjunto ha experimentado
            } //alguna modificación por inserción
        }
        return modificado;
    }

    boolean eliminarElemento(Integer elemento) {
        boolean eliminado = false;
        int indice = lista.buscar(elemento);
        if (indice != -1) {
            lista.eliminar(indice);
            eliminado = true; //true si elemento estaba y se ha eliminado
        }
        return eliminado;
    }

    boolean eliminarConjunto(Conjunto otroConjunto) {
        boolean modificado = false;
        for (int i = 0; i < otroConjunto.lista.numeroElementos(); i++) {
            boolean eliminado = eliminarElemento(otroConjunto.lista.get(i));
            if (eliminado) {
                modificado = true; //true si el conjunto ha experimentado
            } //alguna modificación por eliminación
        }
        return modificado;
    }

    public String mostrar() {
        lista.ordenar(); //se ordena para que sea más cómodo de ver
        return lista.mostrar();
    }
}

```

Programa principal

```

public class Main {

    public static void main(String[] args) {
        Conjunto c1 = new Conjunto();
        for (int i = 0; i < 10; i++) {
            c1.insertar(i);
        }
        System.out.println("c1: " + c1.mostrar());
        Conjunto c2 = new Conjunto();
        for (int i = 0; i < 10; i++) {
            c2.insertar(i + 5);
        }
        System.out.println("c2: " + c2.mostrar());
        c1.eliminarConjunto(c2);
        System.out.println(c1.mostrar());
    }
}

```

7.18. Añadir a la clase `Conjunto` los siguientes métodos estáticos:

`static boolean incluido(Conjunto c1, Conjunto c2)`, que devuelve `true` si `c1` está incluido en `c2`, es decir, si todos los elementos de `c1` están también en `c2`.

`static Conjunto union(Conjunto c1, Conjunto c2)`, que devuelve un nuevo conjunto con todos los elementos que están en `c1` o en `c2` (elementos comunes y no comunes).

`static interseccion(Conjunto c1, Conjunto c2)`, que devuelve un nuevo conjunto con todos los elementos que están en `c1` y en `c2` a la vez (elementos comunes).

`static diferencia(Conjunto c1, Conjunto c2)`, que devuelve un nuevo conjunto con todos los elementos que están en `c1`, pero no en `c2`.

Clase Conjunto

```
/*
 * Añadimos los métodos requeridos en el ejercicio. */
public class Conjunto {

    ... //implementación de Conjunto del ejercicio 7.17

    //nos dice si todos los elementos de c1 están en c2
    static boolean incluido(Conjunto c1, Conjunto c2) {
        boolean incluido = true;
        for (int i = 0; i < c1.numeroElementos() && incluido; i++) {
            if (!c2.pertenece(c1.lista.get(i))) {
                incluido = false;
            }
        }
        return incluido;
    }

    //devuelve todos los elementos, comunes y no comunes, de c1 y c2
    static Conjunto union(Conjunto c1, Conjunto c2) {
        Conjunto nuevo = new Conjunto();
        nuevo.insertar(c1);
        nuevo.insertar(c2);
        return nuevo;
    }

    //devuelve los elementos comunes de c1 y c2
    static Conjunto interseccion(Conjunto c1, Conjunto c2) {
        Conjunto nuevo = new Conjunto();
        for (int i = 0; i < c1.numeroElementos(); i++) {
            if (c2.pertenece(c1.lista.get(i))) {
                nuevo.insertar(c1.lista.get(i));
            }
        }
        return nuevo;
    }

    //devuelve los elementos de c1 que no están en c2
    static Conjunto diferencia(Conjunto c1, Conjunto c2) {
        Conjunto nuevo = new Conjunto();
        for (int i = 0; i < c1.numeroElementos(); i++) {
            if (!c2.pertenece(c1.lista.get(i))) {
                nuevo.insertar(c1.lista.get(i));
            }
        }
        return nuevo;
    }
}
```

Programa principal

```
public class Main {

    public static void main(String[] args) {
        Conjunto c1 = new Conjunto();
        for (int i = 0; i < 10; i++) {
            c1.insertar(i);
        }
        System.out.println("c1: " + c1.mostrar());
        Conjunto c2 = new Conjunto();
        for (int i = 0; i < 10; i++) {
            c2.insertar(i + 5);
        }
        System.out.println("c2: " + c2.mostrar());
        System.out.println("c1 U c2: " + Conjunto.union(c1, c2).mostrar());
        System.out.println("c1 Inter c2: " + Conjunto.interseccion(c1, c2).mostrar());
        System.out.println("c1 - c2: " + Conjunto.diferencia(c1, c2).mostrar());
    }
}
```

Ejercicios propuestos

7.1. En un puesto fronterizo recogemos diariamente la información referente al transito de personas. Para una persona nos interesa almacenar su DNI y nombre completo.

Se pide diseñar una aplicación que presente las siguientes opciones:

1. Paso de frontera.
2. Mostrar todas la personas.
3. Búsqueda por nombre.
4. Búsqueda por DNI.
6. Salir.

Mediante la opción 1 se recogen los datos de la persona que transita a través de la frontera. La opción 2 muestra la información de todas las personas que han pasado por la frontera. Mediante la opción 3 se solicita un nombre por teclado y se muestra la información de todas las persona cuyo nombre coincide con el introducido. Y por último, en la opción 4 introducimos el DNI de una persona de la que obtendremos toda su información.

7.2. Diseñar la clase **Calendario** que representa un fecha concreta (año, mes y día). La clase debe disponer de los métodos:

Calendario(int año, int mes, int dia: que crea un objeto con los datos pasados como parámetros, siempre y cuando, la fecha que representen sea correcta.

void incrementarDia(int cantidad): que incrementa, si resulta correcto, la fecha del calendario en el número de días especificados.

void incrementarMes(int cantidad): que incrementa la fecha del calendario en el número de meses especificados, siempre que la fecha resultante sea correcta.

void incrementarAño(int cantidad): que incrementa la fecha del calendario en el número de años especificados, salvo que el año resultante sea el 0 (que no existió).

void mostrar(): muestra la fecha por consola.

boolean iguales(Calendario otraFecha): que determina si la fecha invocante y la que se pasa como parámetro son iguales o distintas.

- 7.3.** Escribir la clase **Punto** que representa un punto de dos dimensiones (con una componente x y una componente y), con los métodos,

Punto(double x, double y): que construye un objeto con los datos pasados por parámetros.]

void desplaza(double dx): que desplaza la componente x la cantidad dx .

void desplaza(double dx, double dy): que desplaza ambas componentes las cantidades dx en el eje x y dy en la componente y .

double distanciaEuclidea(Punto otro): calcula y devuelve la distancia euclídea entre el punto invocante y el punto **otro**.

void muestra(): muestra por consola la información relativa al punto.

Capítulo 8

Herencia

La *herencia* es una de las grandes cualidades de la POO, que permite, al igual que en la vida real, que las características pasen de padres a hijos. Cuando una clase hereda de otra, adquiere sus atributos y métodos visibles, permitiendo reutilizar el código y las funcionalidades, que en las clases que heredan se pueden ampliar o extender.

La clase de la que se hereda se denomina clase *padre* o *superclase*, y la clase que hereda es conocida como clase *hija* o *subclase*. El diagrama de clases de la Figura 8.1 representa el concepto de herencia.

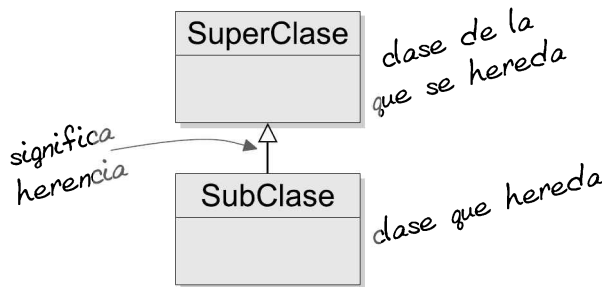


Figura 8.1. Herencia entre clases

Una subclase dispone de los miembros heredados de la superclase y, habitualmente, se amplía añadiéndole nuevos atributos y métodos. Esto aumenta su funcionalidad, a la vez que evita la repetición innecesaria de código. En la API, por ejemplo, la mayoría de las clases no se definen desde cero. Por el contrario, se construyen heredando de otras, lo que simplifica su desarrollo.

8.1. Superclase

La forma de expresar cuál es la superclase de la que heredamos es mediante la palabra reservada **extends**, de la forma,

```
class SubClase extends SuperClase {  
    ...  
}
```

Veamos un ejemplo: supongamos que disponemos de la clase **Persona** —nombre, edad y estatura— y necesitamos construir la clase **Empleado**. Un empleado no es más que una persona —nombre, edad y estatura— con un salario¹. Vamos a definir **Empleado** haciendo que herede de **Persona**, lo que hará que adquiera todos sus miembros y, acto seguido, ampliaremos la clase **Empleado** añadiendo el atributo **salario**.

```
class Empleado extends Persona {  
    double salario;  
    ...  
}
```

Al crear un objeto de **Empleado** disponemos de los siguientes atributos,

```
Empleado e = new Empleado();  
e.nombre = "Sancho"; //atributos heredados  
e.estatura = 1.80;  
e.edad = 25;  
e.salario = 1725.49; //atributo propio
```

El proceso de herencia puede continuar ampliando la biblioteca de clases a partir de las existentes. En nuestro ejemplo, podemos definir a partir de **Empleado** la clase **Jefe**, que no es más que un empleado con unas propiedades añadidas.

Existen lenguajes de programación, como **C++**, que permiten que una clase herede de otras muchas, lo que se conoce como *herencia múltiple*. Java solo permite *herencia simple*, donde cada clase tiene como padre una única superclase, cosa no impide que, a su vez, tenga varias clases hijas.

Es posible, aunque no tiene mucho sentido, dejar la definición de una subclase vacía, con lo cual será una copia exacta de la superclase de la que hereda.

8.2. Modificador de acceso para herencia

Con la aparición de la herencia podemos plantearnos algunas cuestiones, ¿se heredan todos los miembros de una clase?, si no es así, ¿cuáles son los miembros que se heredan? Se heredan todos, aunque los **private** no son visibles en la subclase. No obstante, se puede acceder a ellos con un método no privado.

Por otra parte, con los tipos de visibilidad citados hasta ahora, para que un miembro sea visible desde una subclase, con objeto de obtener una mayor flexibilidad, podemos hacer uso de un nuevo modificador de acceso, **protected**, que está pensado para facilitar la herencia. Funciona de forma muy similar a la visibilidad por defecto, con la diferencia de que los miembros protegidos serán siempre visibles para las clases que hereden, indistintamente de si la superclase y la subclase son vecinas o externas.

¹Un empleado, en realidad, es muchas más cosas pero, para nosotros, será lo que requiera la aplicación que estamos desarrollando.

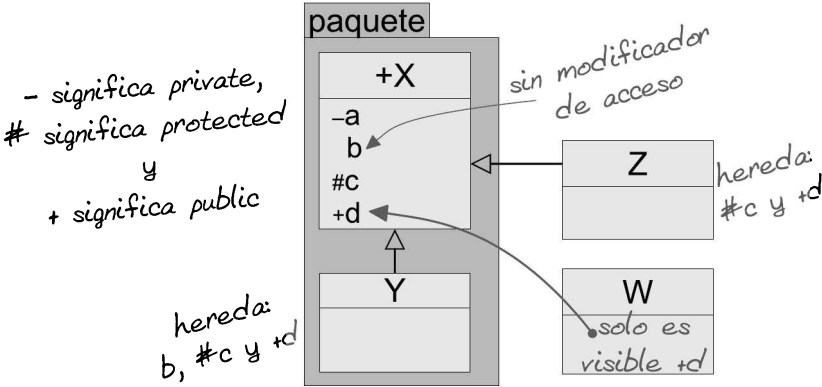


Figura 8.2. Visibilidad de un miembro protected

En resumen, un miembro **protected** es visible en las clases vecinas, no es visible para las clases externas, pero siempre es visible, indistintamente del paquete, desde una clase hija.

La Tabla 8.1 muestra la visibilidad de un miembro **protected** junto al resto de los modificadores.

Tabla 8.1. Modificadores de acceso

	Visible desde...			
	la propia clase	clases vecinas	subclases	clases externas
private	✓			
sin modificador	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

Veamos cómo se define la clase X utilizada en la figura anterior,

```
public class X {
    private int a;    //invisible fuera de la clase
    int b;            //visibilidad por defecto: visible en el paquete
    protected int c; //visibilidad en el paquete y por las subclases
    public int d;     //visibilidad total
}
```

El atributo **a** es invisible desde fuera de la clase; el atributo **b** visible solo desde el mismo paquete, solo será visible por subclases vecinas; **c** solo es accesible desde el mismo paquete, pero en caso de herencia, siempre es visible desde las subclases; y por último **d** es visible desde cualquier lugar.

8.3. Redefinición de miembros heredados

Cuando una clase hereda una serie de miembros, en alguna ocasión puede ocurrir que interese modificar el tipo de algún atributo o redefinir un método. Este mecanismo se conoce como *ocultación* cuando es un atributo y *sustitución* u *overriding* para un método. Consiste en declarar un miembro con igual nombre que uno heredado, lo que hace que este sea ocultado o sustituido por el nuevo.

Veamos como sustituir un método. Partimos de la superclase,

```
class Persona {  
    String nombre;  
    byte edad;  
    double estatura;  
  
    void mostrarDatos() {  
        System.out.println(nombre);  
        System.out.println(edad);  
        System.out.println(estatura);  
    }  
}
```

Vamos a definir una nueva clase,

```
class Empleado extends Persona { //Empleado hereda de Persona  
    double salario; //atributo propio  
}
```

Nos encontramos que la clase `Empleado` dispone, mediante herencia, del método `mostrarDatos()`, pero, en la práctica, este método no basta para mostrar la información de un empleado, ya que no muestra su salario. Una solución es redefinir el método en la clase `Empleado`. Aunque es opcional, los métodos sustituidos en las subclases se suelen marcar con la anotación `@Override`, que indica que el método es una sustitución u overriding de un método de la superclase. Veamos como redefinir el método `mostrarDatos()` en la subclase,

```
class Empleado extends Persona {  
    double salario;  
  
    @Override //significa: sustituye un método de la superclase  
    void mostrarDatos() {  
        System.out.println(nombre);  
        System.out.println(edad);  
        System.out.println(estatura);  
        System.out.println(salario);  
    }  
}
```

El método `mostrarDatos()` definido en `Empleado` sustituye al método, con el mismo nombre y los mismos parámetros de entrada, de `Persona`. Sea el código,

```
Empleado e = new Empleado();  
e.mostrarDatos(); //ejecuta el método de Empleado, no el heredado
```

Ahora veamos un ejemplo de ocultación de un atributo. La estatura de un empleado, como una longitud, no es un dato relevante para la empresa. Pero sí es interesante conocer la estatura como talla del uniforme. En este caso, vamos a redefinir el atributo como una cadena para que contenga la talla del uniforme, con los valores: «XXL», «XL», «L», etc.

La redefinición de la clase `Empleado` quedará,

```
class Empleado extends Persona {
    double salario;
    String estatura;

    @Override
    void mostrarDatos() {
        ...
    }
}
```

En la clase `Empleado` se oculta el atributo `estatura`, de tipo `double`, por otro de tipo `String`. El código de la Figura 8.3 muestra qué miembro es el que se utiliza en cada caso.

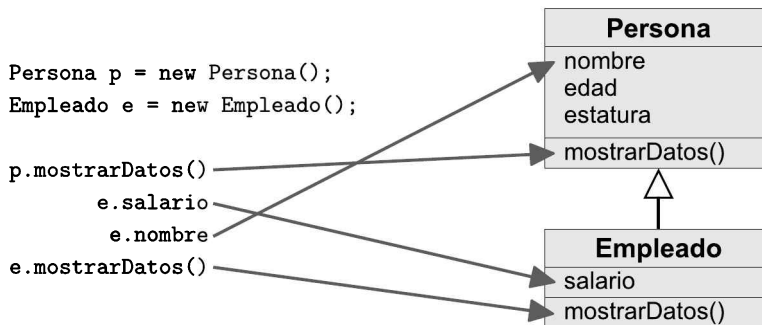


Figura 8.3. Overriding de `Empleado`

8.3.1. `super` y `super()`

La palabra reservada `this` se utiliza para indicar la propia clase. De forma análoga, disponemos de `super`, que hace referencia a la superclase de la clase donde se utiliza.

Sean las siguientes clases:

```
class SuperClase {
    int a;
    int b;
}

class SubClase extends SuperClase {
    String b;
}
```

Como puede apreciarse, en `SubClase` se ha redefinido el atributo `b`. Cada vez que se utiliza dicho atributo en `SubClase` estaremos utilizando un `String`, pero si deseamos

utilizar el atributo `b` de tipo entero, de la superclase desde la subclase, no tendremos más que escribir `super.b`.

Algo habitual es tener un método que muestre los datos de una clase. En caso de que la clase haya heredado de otra, dicho método tiene que redefinirse casi obligatoriamente, para mostrar también los datos de la clase hija. Si utilizamos el método `mostrarDatos()` de la superclase, dejaremos sin mostrar los nuevos atributos definidos en la subclase. En `mostrarDatos()` de la subclase no hace falta copiar el código que muestra los atributos de la superclase; se puede invocar el método `mostrarDatos()` de la superclase, utilizando `super.mostrarDatos()`. El método `mostrarDatos()` de la subclase quedará:

```
void mostrarDatos() {
    super.mostrarDatos(); //muestra los atributos heredados de la superclase
    System.out.println(atributoSubClase);
    ...
}
```

Vamos a escribir de nuevo la clase `Empleado`,

```
class Empleado extends Persona {
    double salario;

    @Override
    void mostrarDatos() {
        super.mostrarDatos(); //método de la superclase
        System.out.println(salario); //muestra el atributo propio
    }
}
```

Algo análogo ocurre con los constructores de la superclase. Para ellos disponemos del método `super()`, que invoca un constructor de la superclase. En caso de estar estos sobrecargados, podemos variar sus parámetros de entrada en número y en tipo. Una restricción a tener en cuenta con `super()` es que si lo utilizamos, tiene que ser forzosamente la primera instrucción de un constructor.

Un ejemplo,

```
public class Persona {
    String nombre;
    byte edad;
    double estatura;

    Persona (String nombre, byte edad, double estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
    }
}
```

Escribimos `Empleado`,

```
class Empleado extends Persona {
    double salario;
```

```
Empleado (String nombre, byte edad, double estatura, double salario) {
    super(nombre, edad, estatura); //constructor de Persona
    this.salario = salario;
}
}
```

8.3.2. Selección dinámica de métodos

Cuando definimos una clase como subclase de otra, los objetos de la subclase son también objetos de la superclase. Por ejemplo, un objeto **Empleado** será, al mismo tiempo, un objeto de **Persona**, ya que posee todos los miembros de **Persona** —además de otros específicos de **Empleado**—. Esto no debe extrañar; ocurre lo mismo en el mundo real: todo empleado es una persona. Por tanto, podemos referenciar un objeto **Empleado** usando una variable **Persona**. Por ejemplo (véase Figura 8.4),

```
Empleado e = new Empleado();
Persona p = e;
```

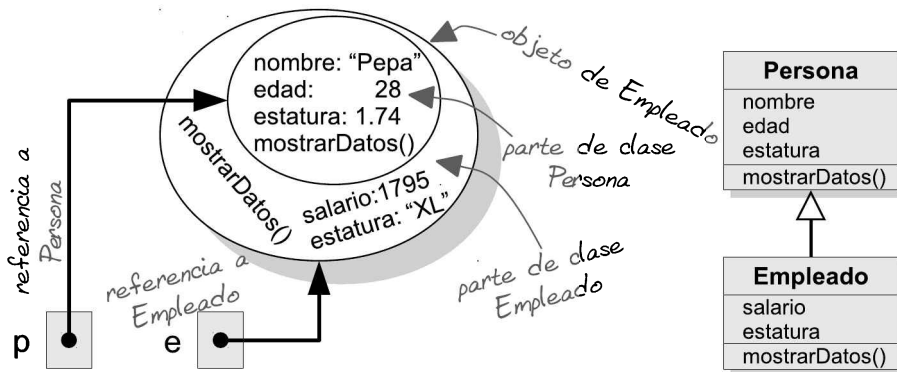


Figura 8.4. Objeto de Empleado

¿Es lo mismo una variable **Empleado** que una variable **Persona** para referenciar un objeto **Empleado**? No. Hay una sutil, pero importante diferencia. En primer lugar, los atributos accesibles son los definidos en la clase de la variable. Por tanto, no se produce la ocultación.

```
p.estatura //atributo de Persona de tipo double
e.estatura //atributo de Empleado de tipo String
```

Pero, en cambio, con los métodos ocurre lo contrario. Se ejecuta la versión del objeto referenciado, es decir, de la subclase **Empleado**. Por tanto, sí funciona el *overriding*.

```
p.mostrarDatos(); //método de Empleado
e.mostrarDatos(); //método de Empleado
```

Esto proporciona una de las herramientas más potentes de que dispone Java para ejercer el polimorfismo: la selección de métodos en tiempo de ejecución. Por ejemplo, supongamos que una tercera clase **Cliente** hereda de **Persona**,

```
class Cliente extends Persona {  
    ...  
    @Override  
    void mostrarDatos() {  
        ...  
    }  
}
```

Si creamos una variable de tipo **Persona**, con ella podemos referenciar tanto objetos de clase **Empleado**, como **Cliente**, como **Persona**. Para todos ellos disponemos del método `mostrarDatos()`, pero se ejecutará una u otra versión, según el objeto referenciado, que puede cambiar en tiempo de ejecución.

```
Persona p;  
p = new Persona();  
p.mostrarDatos(); //se ejecuta el método de Persona  
p = new Empleado();  
p.mostrarDatos(); //se ejecuta el método de Empleado  
p = new Cliente();  
p.mostrarDatos(); //se ejecuta el método de Cliente
```

Así, la misma línea de código, `p.mostrarDatos()`, ejecutará métodos distintos, según el tipo de objeto referenciado.

8.4. La clase **Object**

La clase **Object** del paquete `java.lang` es una clase especial de la que heredan, directa o indirectamente, todas las clases de Java. Es la *superclase* por excelencia, ya que se sitúa en la cúspide de la estructura de herencias entre clases.

Todas las clases que componen la API descenden de la clase **Object**. Incluso, cualquier clase que implementemos nosotros, hereda de **Object**. Esta herencia se realiza por defecto, sin necesidad de especificar nada. Por ejemplo, la definición de la clase **persona**,

```
class Persona {  
    ...  
}
```

es en realidad, equivalente a,

```
class Persona extends Object {  
    ...  
}
```

Y cualquier clase que herede de **Persona**, está heredando, a su vez, de **Object**.

¿Cuál es el objetivo de que todas las clases hereden de **Object**? Haciendo esto se consigue:

- Que todas las clases implementen un conjunto de métodos —en `Object` solo se han definido métodos— que son de uso universal en Java, como realizar comparaciones entre objetos, clonarlos o representar un objeto como una cadena.
- Como se ha visto en el Apartado 8.3.2, poder referenciar cualquier objeto, de cualquier tipo, mediante una variable de tipo `Object`.

Si queremos ver los métodos de `Object` que ha heredado `Persona`, escribiremos en NetBeans una variable de tipo `Persona`, seguida de un punto (`.`) Aparecerán todos los atributos y métodos disponibles: los propios más los heredados de `Object`.

8.4.1. Método `toString()`

Este método devuelve una cadena que representa al objeto desde el que se llama. Tiene el prototipo,

```
public String toString()
```

La implementación de `toString()` en `Object` consiste en devolver el nombre cualificado de la clase a la que pertenece el objeto, seguida de una arroba (`@`) junto a la referencia del objeto. Para una objeto `Persona` devuelve algo similar a,

```
"paquete.Persona@2a139a55"
```

Esta implementación *por defecto* no es útil para representar la mayoría de los objetos, por lo que estamos obligados a realizar un *overriding* de `toString()` en cada clase (que es la única que conoce cómo será la cadena que represente a sus objetos).

Vamos a reimplementar `toString()` en `Persona`: podemos elegir de diversas formas cómo queremos representar una persona, pero en este caso hemos decidido que una representación adecuada consistirá en el nombre junto a la edad, omitiendo la estatura.

```
class Persona {
    ...

    @Override
    public String toString() { //siempre utilizar public
        String cad;
        cad = "Persona: " + nombre + " (" + edad + ")";

        return cad;
    }
}
```

Ahora podemos mostrar en consola la información de un objeto `Persona`,

```
Persona p = new Persona("Claudia", 8, 1.20);
System.out.println(p.toString());
```

`System.out.println()` sabe que todos los objetos tienen un método `toString()` y lo invoca por defecto. Para mostrar los datos de un objeto solo será necesario escribir,

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

8.4.2. Método equals()

Compara dos objetos y decide si son iguales, devolviendo **true** en caso afirmativo y **false** en caso contrario. Es importante destacar que el operador **==** es válido para comparar tipos primitivos, pero no sirve para comparar objetos, ya que solo examina sus referencias, sin fijarse en su contenido. Por ejemplo,

```
Persona a = new Persona("Claudia", 8, 1.20);
Persona b = new persona("Claudia", 8, 1.20);
System.out.println(a == b); //false
```

ya que la comparación se hace atendiendo a las referencias de los objetos, que son distintas, al ocupar lugares distintos en la memoria, no al contenido de sus atributos.

Vamos a reimplementar **equals()** para la clase **Persona**. Tenemos que decidir qué significa que dos personas sean iguales. Para este ejemplo, vamos a considerar dos personas iguales si tienen el mismo nombre y la misma edad²,

```
@Override
public boolean equals(Object otraPersona) { //compara this con otraPersona
    Persona otra = (Persona) otraPersona; //hay que hacer un cast
    boolean iguales;

    if (this.nombre.equals(otra.nombre) &&
        this.edad == otra.edad) {
        iguales = true;
    } else {
        iguales = false;
    }

    return iguales;
}
```

En la condición de **if** hemos utilizado **equals()** para comparar los nombres, ya que son objetos **String**. Pero hemos utilizado **==** para comparar la edad, ya que es un tipo primitivo. Ahora podemos comparar,

```
Persona a = new Persona("Claudia", 8, 1.20);
Persona b = new persona("Claudia", 8, 0.0);
Persona c = new Persona("Pepe", 24, 1.89);

System.out.println(a.equals(b)); //true: compara atributos, no referencias
//equals() compara nombre y edad. Observar que la estatura, a pesar de
//ser distinta, no influye en el resultado
System.out.println(a.equals(c)); //false
```

La mayoría de las clases de la API tienen su propia implementación (overriding) de **equals()**, que permite comparar sus objetos entre sí. Las tablas, a pesar de ser objetos, no hacen overriding de este método, teniendo que recorrer, uno a uno, sus elementos. También podemos utilizar el método estático **equals()** de la clase **Arrays**.

²En la práctica, a la hora de comparar es muy útil utilizar atributos que identifiquen a cada objeto, como el DNI, el número de socio de una biblioteca, etc.

8.5. Clases abstractas

En la jerarquía de herencia de clases, cuanto más abajo, más específica, más particular es la implementación de los métodos. Asimismo, cuanto más arriba, más general, más abstracta. Si subimos lo suficiente, podemos encontrarnos con un método que sabemos que deben tener todas las clases y subclases que heredan de la que estamos considerando, pero que solo podemos implementar conociendo el tipo específico, es decir, la subclase a la que pertenece. Un método definido en una clase, pero cuya implementación se delega en las subclases, se conoce como *abstracto*. Para declarar un método *abstracto* se le antepone el modificador **abstract** y se declara el prototipo, sin escribir el cuerpo de la función. Por ejemplo, para declarar un método abstracto que muestra información del objeto,

```
abstract void mostrarDatos();
```

Las subclases, deberán implementar el método `mostrarDatos()`, cada una con las particularidades específicas de la clase, que no se conocen al nivel de la superclase.

Toda clase que tiene un método abstracto debe ser declarada, a su vez, *abstract*. Las clases abstractas no son instanciables, es decir, no se pueden crear objetos de esa clase. Es lógico, ya que un objeto requiere la implementación de todos los métodos para poder ejecutarlos.

De hecho, las clases abstractas existen para ser heredadas por otras, y no para ser instanciadas. Si una clase hereda de una abstracta, pero deja alguno de sus métodos abstractos sin implementar, a su vez, será abstracta. Una clase abstracta puede tener algún método implementado y algunos atributos definidos, que serán heredados por las subclases.

Vamos a ver todo esto por medio de un ejemplo. Definimos una clase abstracta `A`, donde declaramos e inicializamos una variable `a` entera. Asimismo, definimos e implementamos un método `metodo1`. Tanto la variable como el método serán heredados tal cual por las subclases de `A`. Por otra parte, declaramos un método abstracto `metodo2`,

```
//clase abstracta
abstract class A {
    int a = 1;

    void metodo1() { //método implementado
        System.out.println("método1 definido en A");
    }

    abstract void metodo2(); //método abstracto para ser implementado
                           //por las subclases
}
```

A continuación, definimos las clases `B` y `C` que heredan de `A`, e implementan el método `metodo2`. Ambas clases heredan tanto la variable `a` como el método `metodo1`,

```
class B extends A {
    void metodo2() {
        System.out.println("método2 definido en B");
    }
}
```

```
class C extends A {  
    void metodo2() {  
        System.out.println("método2 definido en C");  
    }  
}
```

En el programa principal creamos sendos objetos de clase B y C (de la clase A no es posible, puesto que es abstracta) y ejecutamos los métodos `metodo1` y `metodo2` de cada uno de los dos objetos,

```
B b = new B();  
C c = new C();  
System.out.println("Valor de a en la clase B: " + b.a); //heredado de A  
b.metodo1();  
b.metodo2();  
c.metodo1();  
c.metodo2();
```

El resultado mostrado por consola será,

```
Valor de a en la clase B: 1  
método1 definido en A  
método2 definido en B  
método1 definido en A  
método2 definido en C
```

El atributo `a` mostrado desde el objeto `b` de clase B es el valor heredado de la clase A. Asimismo, desde los objetos `b` (de clase B) y `c` (de clase C) ejecutamos `metodo1`, heredado de A. En cambio, al ejecutar `metodo2` desde `b` y `c` se ejecuta la versión implementada en las clases B y C, respectivamente. El que no se puedan crear objetos de clase A no significa que no puedan existir variables de dicha clase. Una variable de clase A puede hacer referencia a cualquier objeto de una subclase de A que no sea abstracta, por ejemplo, de tipo B o C. Al código anterior le podemos añadir las líneas,

```
A a;  
a = b;  
a.metodo2();
```

Como el objeto referenciado es de clase B, la versión de `metodo2` ejecutada será la implementada en B. Si ahora asignamos a `a` la referencia de `c` de tipo C, se ejecutará la versión de `metodo2` implementada en C,

```
a = c;  
a.metodo2();
```

Como vemos, con la misma línea de código `a.metodo2()`, se ejecutan implementaciones distintas, es decir, código distinto. Estamos ante un ejemplo de *selección dinámica de métodos*.

Ejercicios de herencia

8.1. Diseñar la clase `Hora` que representa un instante de tiempo compuesto por una hora (de 0 a 23) y los minutos.

Dispone de los métodos:

- `Hora(hora, minuto)`, que construye un objeto con los datos pasados como parámetros.
- `inc()`, que incrementa la hora en un minuto.
- `setMinutos(valor)`, que asigna un valor, si tiene sentido, a los minutos.
- `setHora(valor)`, que asigna un valor, si tiene sentido, a la hora.
- `toString()`, que devuelve un `String` con la representación del reloj.

Clase `Hora`

```
public class Hora {

    protected int hora, minutos; //atributos protegidos, pensados para heredar

    Hora(int hora, int minutos) { //constructor
        this.hora = 0; //valores por defecto
        this.minutos = 0;
        setHora(hora); //utilizamos métodos de asignación, que comprueban valores válidos
        setMinutos(minutos);
    }

    public void inc() { //incrementa la hora +1 minuto
        minutos++;
        if (minutos > 59) { //comprobamos si los minutos sobrepasan 59
            minutos = 0; //reiniciamos los minutos a 0
            hora++; //e incrementamos la hora
            if (hora > 23) { //si la hora es mayor a 23 (algo que no tiene sentido)
                hora = 0; //reiniciamos la hora a 0
            }
        }
    }

    public void setMinutos(int minutos) {
        if (0 <= minutos && minutos < 60) { //solo modificamos si valor está en 0..59
            this.minutos = minutos;
        }
    }

    public void setHora(int hora) {
        if (0 <= hora && hora < 24) { //solo modificamos si valor está en 0..23
            this.hora = hora;
        }
    }

    @Override //indica que estamos sustituyendo (overriding) el método
    public String toString() {
        String result;
        result = hora + ":" + minutos;
        return result;
    }
}
```

Programa Principal

```
//vamos a probar la clase Hora
public class Main {
    static public void main(String args[]) {
        Hora r = new Hora(11, 30); //las 11:30
        System.out.println(r);
        for (int i = 1; i <= 61; i++) { //incrementamos 61 minutos
            r.inc();
        }
        System.out.println(r); //mostramos
        r.setHora(20); //cambiamos la hora a las 20
        System.out.println(r);
    }
}
```

- 8.2. Escribir la clase Hora12, que funciona de forma similar a la clase Hora, con la diferencia de que las horas solo pueden tomar un valor entre la 1 y las 12; y se distingue la mañana de la tarde mediante “am” y “pm”.

Clase Hora12

```
/*
 * Hereda de la clase Hora. Tendremos que añadir un atributos para saber si la hora es
 * pm o am. Habrá que añadir algunos métodos y sustituir otros.
 * Aprovechamos el código de la superclase con super y super();
 */
public class Hora12 extends Hora {
    public enum Meridiano { AM, PM} //tipo enumerado para distinguir el meridiano
    protected Meridiano mer; //indica si la hora es am (mañana) o pm (tarde)

    public Hora12(int hora, int minutos, Meridiano mer) {
        super(hora, minutos); //constructor superclase. OJO!: permite horas de 0..23
        setHora(hora); //utilizamos nuestro propio método para asignar hora (de 1 a 12)
        this.mer = mer; //asignamos el meridiano
    }

    @Override //estamos sustituyendo un método de la superclase
    public void setHora(int hora) { //las horas pueden estar en el intervalo 1..12
        if (1 <= hora && hora <= 12) {
            this.hora = hora;
        }
    }

    @Override //sustituimos el método incrementar de la superclase
    public void inc() {
        super.inc(); //incrementamos un minutos con el método inc() de la superclase
        //la hora puede llegar a ser las 12:00 (que aquí no tiene sentido)
        if (hora > 12) {
            hora = 1; //inicializamos la hora
            mer = mer == Meridiano.AM ? Meridiano.PM : Meridiano.AM; //cambia el meridiano
        }
    }

    @Override
    public String toString() {
        String result; //cadena con el resultado a devolver
        result = super.toString(); //aprovechamos toString() de la superclase
        result += " " + mer; //añadimos a la hora (h:m) pm o am
        return result;
    }
}
```

Programa Principal

```

public class Main {
    static public void main(String args[]) {
        Hora12 r = new Hora12 (12, 10, Hora12.Meridiano.AM);
        System.out.println(r);
        for (int i = 1; i <= 61; i++) {
            r.inc();
        }
        System.out.println(r);

        r.setHora(20);
        System.out.println(r);
    }
}

```

8.3. A partir de la clase *Hora* implementar la clase *HoraExacta*, que incluye en la hora los segundos. Además de los métodos visibles de *Hora* dispondrá de:

- *HoraExacta(hora, minuto, segundo)*, que construye un objeto con los datos pasados como parámetros.
- *setSegundo(valor)*, que asigna, cuando es posible, el valor indicado a los segundos.
- *inc()*, que incrementa la hora en un segundo.

Clase *HoraExacta*

```

public class HoraExacta extends Hora { //heredamos de la clas Hora
    protected int segundos; //añadimos un atributo para los segundos

    public HoraExacta(int hora, int minutos, int segundos) {
        super(hora, minutos); //aprovechamos el constructor de la superclase
        //this.segundos = segundos; permitiría asignar cualquier valor a los segundos
        setSegundos(segundos); //mejor utilizar el método para asignar y comprobar valores
    }

    //añadimos un método que asigna los segundos
    public void setSegundos(int segundos) {
        if (0 <= segundos && segundos < 60) { //si está en un rango válido
            this.segundos = segundos; //modificamos los segundos
        }
    }

    @Override //sustituimos el método para incrementar segundos en lugar de minutos
    public void inc() {
        segundos++;
        if (segundos > 60) { //si los segundos son mayores que 60
            segundos = 0; // inicializamos los segundos
            super.inc(); //+1 minuto con el método inc() de la superclase
        }
    }

    @Override //sustituimos toString() para mostrar los segundos
    public String toString() {
        String result = super.toString(); //utilizamos toString() de la superclase
        result += ":" + segundos; //añadimos los segundos
        return result;
    }
}

```

Programa Principal

```
public class Main {
    static public void main(String args[]) {
        HoraExacta r = new HoraExacta (25, 60, 31);
        System.out.println(r);
        for (int i = 1; i <= 61; i++) {
            r.inc();
        }
        System.out.println(r);

        r.setHora(20);
        System.out.println(r);
    }
}
```

- 8.4. Añadir a la clase `HoraExacta` un método que compare si dos horas (la invocante y otra pasada como parámetro de entrada al método) son iguales o distintas.

Clase `HoraExacta`

```
public class HoraExacta extends Hora {
    ... //resto de implementación de la clase

    //Implementaremos (overriding) el método equals() de la clase Object, para comparar
    //dos horas, que serán iguales si sus horas, minutos y segundos son iguales.
    //La hora con la que tenemos que comparar se pasa como un objeto de la clase Object,
    //que tendremos que convertir (cast) a HoraExacta.
    @Override
    public boolean equals(Object o) {
        HoraExacta otroReloj = (HoraExacta) o; //el mismo objeto está referenciado
        //como Object (con el parámetro o) y como HoraExacta (con la variable otroReloj).
        boolean iguales;
        if (this.hora == otroReloj.hora //si las horas son iguales
            && this.minutos == otroReloj.minutos // y los minutos son iguales
            && this.segundos == otroReloj.segundos) {//y los segundos son iguales
            iguales = true; //son iguales
        } else {
            iguales = false; //no son iguales
        }

        return iguales;
    }
}
```

Programa Principal

```
public class Main {
    static public void main(String args[]) {
        HoraExacta a = new HoraExacta (1, 2, 3);
        HoraExacta b = new HoraExacta (1, 2, 3);
        HoraExacta c = new HoraExacta (10, 20, 30);

        System.out.println(a.equals(b));
        System.out.println(a.equals(c));
    }
}
```

- 8.5. Crear la clase **Instrumento** que es una clase abstracta que almacena un máximo de 100 notas musicales. Mientras haya sitio es posible añadir nuevas notas musicales, al final, con el método `add()`. La clase también dispone del método abstracto `interpretar()` que en cada subclase que herede de **Instrumento**, mostrará por consola las notas musicales según las interprete. Utilizar enumerados para definir las notas musicales.

Clase **Instrumento**

```
/*
 * La clase abstracta Instrumento, básicamente contiene una tabla con una serie de notas.
 * Cada clase que herede de Instrumento, tendrá que implementar el método interpretar()
 * donde deciden de qué forma suenan las notas. La forma que tendremos de distinguir un
 * timbre u otro, será mediante la forma en que escribamos las notas, por ejemplo: do,
 * Do, Doloon, dooooooooooooo, etc. */
public abstract class Instrumento {
    public enum Nota {DO, RE, MI, FA, SOL, LA, SI} //tipo enumerado con las notas

    protected Nota notas[]; //tabla que almacena las notas a interpretar
    protected int numNotas; //número de notas que se guardan en la tabla
    static protected int TAM_MAX_TABLA = 100; //la tabla contendrá un máximo de 100 notas

    public Instrumento() {
        notas = new Nota[TAM_MAX_TABLA]; //creamos la tabla
        numNotas = 0; //por ahora, no hay ninguna nota a interpretar
    }

    //Añade, si hay sitio, la nota n al final de la tabla
    void add(Nota n) {
        if (numNotas < notas.length) { //si existe sitio para otra nota
            notas[numNotas] = n; //la insertamos al final
            numNotas++; //ahora tenemos una nota más para tocar
        }
    }

    abstract void interpretar(); //a implementar en cada subclase
}
```

- 8.6. Crear la clase **Piano** y la clase **Campana** que heredan de **Instrumento**.

Clase **Campana**

```
/*
 * Una campana es un instrumento musical que interpreta las notas con un
 * timbre determinado. */
public class Campana extends Instrumento {
    ... //podemos añadir todos los atributos y métodos que necesitemos a la clase

    public Campana() {
        super(); //utilizamos el constructor de la superclase
    }

    @Override //implementamos el método abstracto
    //recorreremos las notas y las interpretaremos de una forma característica.
    public void interpretar() {
        for (int i = 0; i < numNotas; i++) {
            switch (notas[i]) {
                case DO:
                    System.out.print("Dolonnann ");
                    break;
                case RE:
                    System.out.print("Reeeeen ");
                    break;
            }
        }
    }
}
```

```
        case MI:
            System.out.print("Miiiiii ");
            break;
        case FA:
            System.out.print("Faaaaaa ");
            break;
        case SOL:
            System.out.print("Sooooool ");
            break;
        case LA:
            System.out.print("Laaaaaaa ");
            break;
        case SI:
            System.out.print("Siiiiiii ");
            break;
    }
}
System.out.println("");
}
```

Clase Piano

```
/*
 * Un piano es un instrumento que interpreta las notas con un timbre muy característico.
 */
public class Piano extends Instrumento {
    ...//podemos añadir tantos atributos y métodos como necesitemos

    public Piano() {
        super(); //constructor de la superclase
    }

    @Override //implementamos el método abstracto
    //recorreremos las notas y las interpretaremos de una forma característica.
    public void interpretar() {
        for (int i = 0; i < numNotas; i++) {
            switch (notas[i]) {
                case D0:
                    System.out.print("do ");
                    break;
                case RE:
                    System.out.print("re ");
                    break;
                case MI:
                    System.out.print("mi ");
                    break;
                case FA:
                    System.out.print("fa ");
                    break;
                case SOL:
                    System.out.print("sol ");
                    break;
                case LA:
                    System.out.print("la ");
                    break;
                case SI:
                    System.out.print("si ");
                    break;
            }
        }
        System.out.println("");
    }
}
```


Programa Principal

```
public class Main {

    public static void main(String[] args) {
        Campana c = new Campana();
        c.add(Instrumento.Nota.DO);
        c.add(Instrumento.Nota.SI);
        c.add(Instrumento.Nota.SOL);
        c.add(Instrumento.Nota.RE);
        c.interpretar();

        Piano p = new Piano();
        p.add(Instrumento.Nota.DO);
        p.add(Instrumento.Nota.SI);
        p.add(Instrumento.Nota.MI);
        p.add(Instrumento.Nota.FA);
        p.add(Instrumento.Nota.DO);
        p.interpretar();
    }
}
```

8.7. Las empresas de transportes, para evitar daños en los paquetes, embalan todas sus mercancías en cajas con el tamaño adecuado. Una caja se crea expresamente con un ancho, un alto y un fondo y, una vez creada, se mantiene inmutable. Cada caja lleva pegada una etiqueta con información útil como el nombre del destinatario, dirección, etc. Se pide implementar la clase **Caja** con los métodos,

- **Caja(int ancho, int alto, int fondo, Unidades u)**: que construye una caja con las dimensiones especificadas, que pueden encontrarse en “cm” (centímetros) o “m” (metros).
- **double getVolumen()**: que devuelve el volumen de la caja en metros cuadrados.
- **String toString()**: que devuelva una cadena con la representación de la caja.

Clase Caja

```
/* Las cajas una vez creadas no pueden modificar sus dimensiones, lo único que se
 * puede cambiar es el texto de la etiqueta que lleva pegada.
 * Siempre guardaremos las dimensiones en metros. */
public class Caja {

    public enum Unidad {CM, M} //centímetros y metros como posibles unidades de medida

    protected final double ancho, alto, fondo; //dimensiones
    protected final Unidad unid; //unidades de medida
    protected double volumen; //el volumen lo calculamos siempre en metros cúbicos
    public String etiqueta; //permitimos que la etiqueta se modifique libremente

    public Caja(double ancho, double alto, double fondo, Unidad unid) {
        this.ancho = ancho;
        this.alto = alto;
        this.fondo = fondo;
        this.unid = unid;

        switch (unid) { //el volumen se calcula siempre en metros cúbicos
            case CM:
                volumen = (ancho / 100) * (alto / 100) * (fondo / 100); //pasamos a metros
                break;
        }
    }
}
```

```
        case M:
            volumen = ancho * alto * fondo; //las medidas ya están en metros
            break;
    }
}

//devuelve la capacidad de la caja, siempre en metros cúbicos
public double getVolumen() {
    return volumen;
}

@Override
public String toString() {
    return ancho + "x" + alto + "x" + fondo + " " + unid.toString() + "\n" + etiqueta;
}
}
```

Programa Principal

```
//probamos la clase caja
public class Main {

    public static void main(String[] args) {
        Caja a, b;
        a = new Caja(100, 200, 200, Caja.Unidad.CM);
        a.etiqueta = "Antonio Pérez\nCalle Larga n 8";
        b = new Caja(1.2, 0.9, 1.45, Caja.Unidad.M);
        b.etiqueta = "Pepa González. Entregar antes de las 10:00h";

        System.out.println(a);
        System.out.println("Volumen: " + a.getVolumen());
        System.out.println(b);
        System.out.println("Volumen: " + b.getVolumen());
    }
}
```

8.8. La empresa de mensajería BiciExpress que reparte en bicicleta, para disminuir el peso a transportar por sus empleados solo utiliza cajas de cartón. Por motivos de privacidad, las etiquetas (con texto) que se pegan en las cajas normales se sustituyen por una etiqueta con un número (que determina el cliente, la dirección del envío, etc). Además las cajas de cartón se caracterizan porque su volumen se calcula como el 80 % del volumen real, ya que si las cajas se llenan mucho, se deforman y se rompen. Otra característica de las cajas de cartón es que sus medidas siempre están en centímetros. Por último, la empresa, para controlar costes, necesita saber cuál es la superficie total de cartón utilizado para construir todas las cajas enviadas. Se pide implementar a partir de la clase `Caja` la clase `CajaCarton`.

Clase `CajaCarton`

```
/*
 * Una caja de cartón no es más que una caja, con algunas características, como:
 * -Las cajas de cartón utilizan como unidad de medida estándar el cm
 * -Es necesario controlar (guardar) el área de la superficie de la caja
 * -Y tenemos que acumular, para todas las cajas de cartón, el cartón total utilizado
 */
```

```
public class CajaCarton extends Caja {
    //añadimos nuevos atributos
    static double cartonTotal = 0; //cartón utilizado(cm2) para construir todas las cajas
    protected double area; //area de la superficie de la caja de cartón, en cm2

    //oculto el atributo etiqueta (que en la superclase es una cadena) con un entero
    int etiqueta; //ahora la etiqueta de la caja es un número

    CajaCarton(double ancho, double alto, double fondo) {
        super(ancho, alto, fondo, CajaCarton.Unidad.CM); //reutilizamos constructor
        area = 2 * (ancho * alto + ancho * fondo + alto * fondo); //calculamos el área
        cartonTotal += area; //acumulamos el área total de cartón utilizado en las
        //contrucción de todas las cajas
    }

    double getArea() {
        return area; //devolvemos el área
    }

    @Override //sustituimos el método de la superclase
    public double getVolumen() {
        //Aunque el volumen de una caja de cartón coincida con el volumen de una caja,
        //en la práctica, para evitar roturas, se calcula el volumen "práctico" como un
        // 80% del volumen real:
        return volumen * 0.8; //devolvemos el 80% del volumen
    }

    @Override
    public String toString() {
        String result = "Cartón total " + cartonTotal + "cm2\n";
        result += "Área: " + area + "cm2\n";
        result += super.toString(); //aprovechamos el método de la superclase
        return result;
    }
}
```

Programa Principal

```
public class Main {
    //probamos la clase CajaCarton
    public static void main(String[] args) {
        Caja a = new CajaCarton(100, 200, 200); //variable de tipo Caja
        a.etiqueta = "Dirección envío";
        System.out.println(a);
        System.out.println("Volumen: " + a.getVolumen());

        CajaCarton b = new CajaCarton(50.6, 75.5, 100);
        b.etiqueta = 23;
        System.out.println(b);
        System.out.println("Volumen: " + b.getVolumen());
    }
}
```

8.9. Modificar la clase Lista del Ejercicio 7.12 sustituyendo el método `mostrar()` por el método `toString()`.

Clase Lista

```
/*
 * Sustituimos el método mostrar() por un overriding de toString() (heredado de
 * la clase Object). * /
```

```
import java.util.Arrays;

public class Lista {

    ... // implementación de la clase Lista en el ejercicio 7.12

    @Override
    public String toString() {
        String res = "";
        for (int i = 0; i < numeroElementos; i++) {
            res += tabla[i] + " ";
        }

        return res;
    }
}
```

8.10. Implementar la clase Pila heredando de Lista (implementada en el ejercicio anterior).

Clase Pila

```
/*
 * Otra forma de implementar la clase Pila es heredar de Lista, aprovechando así todas
 * sus funcionalidades. Solo tenemos que definir las operaciones propias de las pilas:
 * apilar y desapilar. Apilaremos y desapilaremos del mismo lugar, que será el final de
 * la lista, que es menos costoso en computación, ya que no hay que mover de sitio
 * ningún dato previamente insertado.
 */

public class Pila extends Lista { //herede de Lista (ejercicio resuelto 12 de clases)

    void apilar(Integer elemento) {
        insertarFinal(elemento);
    }

    Integer desapilar() {
        return eliminar(numeroElementos() - 1); //final de la lista
    }
}
```

Programa Principal

```
public class Main {
    //programa principal para probar la clase Pila
    public static void main(String[] args) {
        Pila p = new Pila();

        for (int i = 0; i < 10; i++) {
            p.apilar(i);
        }

        Integer num = p.desapilar();
        while (num != null) {
            System.out.println(num);
            num = p.desapilar();
        }
    }
}
```

8.11. Escribir la clase Cola heredando de la clase Lista.

Clase Cola

```

/*
 * Otra forma de implementar la clase Cola es heredar de Lista, aprovechando así todas
 * sus funcionalidades. Solo tenemos que definir las operaciones propias de las colas:
 * encolar y desencolar. Encolaremos al final de la lista y desencolaremos del principio.
 */
public class Cola extends Lista {

    void encolar(Integer elemento) {
        insertarFinal(elemento);
    }

    Integer desencolar() {
        return eliminar(0);
    }
}

```

Programa Principal

```

public class Main {
    //probamos la clase Cola
    public static void main(String[] args) {
        Cola c = new Cola();

        for (int i = 0; i < 10; i++) {
            c.encolar(i);
        }

        Integer num = c.desencolar();

        while (num != null) {
            System.out.println(num);
            num = c.desencolar();
        }
    }
}

```

8.12. Diseñar la clase Conjunto heredando de Lista.

Clase Conjunto

```

/*
 * Otra forma de implementar un conjunto es heredar de Lista, aprovechando todas sus
 * funcionalidades, para implementar los métodos propios de los conjuntos, teniendo en
 * cuenta que no puede haber elementos repetidos. Además, como el orden no importa,
 * haremos las inserciones al final, ya que nuestra clase Lista se basa en tablas, donde
 * la inserción de un nuevo dato al final evita tener que mover los datos ya insertados
 * previamente.
 */
public class Conjunto extends Lista {

    boolean pertenece(Integer elemento) {
        return buscar(elemento) != -1; //si elemento pertenece, no devuelve -1
    }

    //Utilizamos el método de inserción al final de la lista
    boolean insertar(Integer nuevo) {
        boolean insertado = false;
    }
}

```

```
        if (!pertenece(nuevo)) {
            insertarFinal(nuevo);
            insertado = true;
        }
        return insertado;
    }

    boolean insertar(Conjunto otroConjunto) {
        boolean modificado = false;
        //Recorremos el conjunto, utilizando el método de acceso posicional get(), de Lista
        for (int i = 0; i < otroConjunto.numeroElementos(); i++) {
            if (!pertenece(otroConjunto.get(i))) {
                modificado = insertar(otroConjunto.get(i));
            }
        }
        return modificado;
    }

    //Devuelve el elemento eliminado
    Integer eliminarElemento(Integer elemento) {
        Integer eliminado = null;
        int i = buscar(elemento);
        //Si elemento está en el conjunto. No usamos el método pertenece()
        //porque necesitamos el índice para eliminarlo
        if (i != -1) {
            eliminado = eliminar(i); //llamamos al método de la superclase
        }

        return eliminado;
    }

    //Eliminamos todos los elementos que pertenecen a otroConjunto. Si se elimina
    //algún elemento, el conjunto this se modifica y devuelve true
    boolean eliminarConjunto(Conjunto otroConjunto) {
        boolean modificado = false;
        for (int i = 0; i < otroConjunto.numeroElementos(); i++) {
            Integer eliminado = eliminarElemento(otroConjunto.get(i));
            if (eliminado != null) {
                modificado = true;
            }
        }
        return modificado;
    }

    @Override
    public String toString() {
        String res = "";
        //En un conjunto, en principio, el orden de los elementos no importa.
        //Ordenamos la tabla para hacer más cómoda su lectura por la pantalla.
        ordenar();
        for (int i = 0; i < numeroElementos(); i++) {
            res += get(i) + " ";
        }

        return res;
    }

    //Nos dice si todos los elementos de c1 están en c2
    static boolean incluido(Conjunto c1, Conjunto c2) {
        boolean incluido = true;
        for (int i = 0; i < c1.numeroElementos() && incluido; i++) {
            if (!c2.pertenece(c1.get(i))) {
                incluido = false;
            }
        }

        return incluido;
    }
}
```

```

//Devuelve todos los elementos, comunes y no comunes, de c1 y c2
static Conjunto union(Conjunto c1, Conjunto c2) {
    Conjunto nuevo = new Conjunto();
    nuevo.insertar(c1);
    nuevo.insertar(c2);
    return nuevo;
}

//Devuelve los elementos comunes de c1 y c2
static Conjunto interseccion(Conjunto c1, Conjunto c2) {
    Conjunto nuevo = new Conjunto();
    for (int i = 0; i < c1.numeroElementos(); i++) {
        if (c2.pertenece(c1.get(i))) {
            nuevo.insertar(c1.get(i));
        }
    }
    return nuevo;
}

//Devuelve los elementos de c1 que no están en c2
static Conjunto diferencia(Conjunto c1, Conjunto c2) {
    Conjunto nuevo = new Conjunto();
    for (int i = 0; i < c1.numeroElementos(); i++) {
        if (!c2.pertenece(c1.get(i))) {
            nuevo.insertar(c1.get(i));
        }
    }
    return nuevo;
}
}

```

Programa Principal

```

public class Main {
    //programa principal para probar la clase Conjunto
    public static void main(String[] args) {
        Conjunto c1 = new Conjunto();
        for (int i = 0; i < 10; i++) {
            c1.insertar(i);
        }
        Conjunto c2 = new Conjunto();
        for (int i = 5; i < 15; i++) {
            c2.insertar(i);
        }
        System.out.println("c1: " + c1);
        System.out.println("c2: " + c2);

        System.out.println("c1 U c2: " + Conjunto.union(c1, c2));
        System.out.println("c1 Inter c2: " + Conjunto.interseccion(c1, c2));
        System.out.println("c1 - c2: " + Conjunto.diferencia(c1, c2));
    }
}

```

Ejercicios propuestos

- 8.1. A partir de la clase `Punto` implementada en el Ejercicio propuesto 7.3, se pide diseñar, mediante herencia, la clase `Punto3D` que representa un punto en tres dimensiones (con tres componentes x , y y z).

- 8.2.** A partir de la clase `Calendario`, implementada en el ejercicio propuesto 7.2, se pide escribir la clase `CalendarioExacto` que determina un instante de tiempo exacto formado por un año, un mes, un día, una hora y un minuto. Implementar los métodos `toString()`, `equals()` y aquellos necesarios para manejar la clase.

Capítulo 9

Interfaces

Supongamos que vamos a trabajar con clases de animales. De ellos, unos tienen la capacidad de emitir un sonido (por ejemplo, los perros, los gatos o los lobos) y otros no. Los primeros, por tanto, tendrán el método,

```
void voz()
```

9.1. Interfaces

Sin embargo, la forma en que se ejecuta dicho método es distinta, según la clase. En un objeto de la clase **Gato**, la ejecución de `voz()` hará que aparezca en la pantalla la cadena «¡Miau!». En cambio, en un objeto **Perro** mostrará «¡Guau!». Pero todos ellos tienen en común que implementan el método `voz()`.

Hay una forma de expresar en Java esa capacidad común de algunas clases, en este caso aquellas que tienen implementado el método `voz()`. A dichas funcionalidades comunes las definimos en lo que llamamos *interfaces*. En nuestro ejemplo hablaríamos de la interfaz **Sonido**, que consiste en implementar el método `voz()`. Diríamos que las clases **Perro**, **Gato** y **Lobo** implementan la interfaz **Sonido** ya que, entre sus métodos está `voz()`, mientras que las clases **Caracol** y **Lagartija** no.

La definición de la interfaz tendría la forma,

```
interface Sonido {  
    //métodos de la interfaz  
    void voz();  
}
```

En ella se define en qué consiste la interfaz **Sonido**, que en este caso es el método `voz()`. En la definición aparece el nombre del método, la lista de parámetros de entrada (aquí está vacía) y el tipo devuelto (`void`). Naturalmente, no se implementa el cuerpo de la función, ya que este depende de la clase concreta que implementa **Sonido**. Es en la definición de cada clase donde se decide qué hace exactamente `voz()`. Por ejemplo, la clase **Perro** se definiría,

```
class Perro implements Sonido {
    public void voz() {
        System.out.println("¡Guau!");
    }
    ... //resto de la implementación de Perro
}
```

La implementación en una clase de un método de una interfaz tiene que declararse **public**, como hemos hecho con el método `voz()`.

Después del nombre de la clase —**Perro**— hemos añadido la palabra clave **implements** y el nombre de la interfaz **Sonido**. Con ello estamos declarando que la clase **Perro** tiene implementada la interfaz **Sonido** y que, por tanto, tiene entre sus métodos `voz()`. La definición de la clase **Gato** sería,

```
class Gato implements Sonido {
    public void voz(){
        System.out.println("¡Miau!");
    }
    ... //resto de la implementación de Gato
}
```

En cambio, la clase **Caracol** no implementa la interfaz **Sonido**, y entre sus métodos no debemos esperar la implementación de `voz()`.

Una interfaz puede consistir en más de un método, e incluso puede contener atributos. Diremos que una clase implementa una interfaz si implementa todos sus métodos.

En general, una interfaz es una declaración de funcionalidades, una especie de compromiso asumido por una clase. Cuando una clase declara que tiene implementada una interfaz concreta, se está comprometiendo a tener implementadas en su definición todas las funcionalidades de esa interfaz.

Desde el punto de vista sintáctico, una interfaz se parece mucho a una clase: tiene atributos y métodos. Pero las interfaces no son instanciables, es decir, no se pueden crear objetos de una interfaz. En la definición de una interfaz hay métodos (llamados métodos abstractos) que son declarados, pero no implementados, igual que ocurre en las clases abstractas. En nuestro ejemplo, `voz()` es el único método de la interfaz, y además es abstracto. Junto a ellos pueden aparecer otros métodos, llamados métodos por defecto, y métodos estáticos, que sí están implementados en la propia interfaz. Además pueden declararse atributos. Los métodos abstractos de una interfaz deberán ser implementados por las clases que implementan dicha interfaz, pero los métodos por defecto y los estáticos, no.

Vamos a ver un ejemplo de atributo definido en una interfaz: si queremos llevar la cuenta de las sucesivas versiones de nuestra interfaz **Sonido**, podemos definir el atributo entero `version`. La definición de la interfaz tendría la forma,

```
interface Sonido {
    int version = 1;
    void voz();
}
```

El atributo `version` se convertirá automáticamente en un atributo de las clases **Gato** y **Perro**, y solo podrá cambiarse en la definición de **Sonido**, pero no en las clases que la

implementan, ya que los atributos definidos en una interfaz son **static** y **final** por defecto, sin necesidad de escribirlo explícitamente.

Para ser exactos, se dice que una clase implementa una interfaz cuando implementa sus métodos abstractos. Como ya hemos mencionado, los métodos por defecto y los estáticos son implementados en la definición de la interfaz, donde también se declaran e inicializan los atributos. Tanto unos como otros son incorporados por las clases de forma automática (por defecto), aunque los métodos por defecto pueden ser reimplementados por las clases, haciendo overriding de ellos.

Los métodos no abstractos de una interfaz (por defecto y estáticos) se llaman *de extensión*. Los métodos por defecto se deben declarar como **default**. Vamos a añadir uno a nuestro ejemplo de los animales. Supongamos que, entre los sonidos de los animales, queremos incluir los ruidos que hacen durmiendo, y que todos emiten el mismo sonido cuando duermen. En este caso, podemos incluir e implementar la función `void vozDurmiendo()` en la interfaz, sin tener que esperar a la definición de ninguna clase particular,

```
interface Sonido {
    int version = 1;
    void voz();

    default void vozDurmiendo() {
        System.out.println("Zzzzzzzzz");
    }
}
```

Este método será incorporado por **Perro** y **Gato**, tal como está implementado en la interfaz **Sonido** y será accesible por cualquier objeto de una de esas clases,

```
Gato g = new Gato();
g.vozDurmiendo();
Perro p = new Perro();
p.vozDurmiendo();
```

En ambos casos, se muestra por pantalla el mensaje: «Zzzzzzzzz».

No obstante, `vozDurmiendo()` se puede reimplementar en cada clase haciendo overriding de la implementación que se ha hecho en la interfaz.

Supongamos que descubrimos que los leones son una excepción, y rugen hasta cuando duermen. Entonces la clase **Leon** sería,

```
class Leon implements Sonido {
    public void voz() {
        System.out.println("¡Grrrr!");
    }

    @Overriding //de la implementación en Sonido, debe ser public
    public void vozDurmiendo() {
        System.out.println("¡Grrrr!");
    }
    ... //resto de la implementación de Leon
}
```

En este caso, el trozo de código,

```
Leon le = new Leon();  
le.vozDurmiendo();
```

mostraría por consola «¡Grrrr!».

No se debe olvidar el modificador `public` para hacer el overriding.

Si implementamos un método estático en una interfaz, pertenecerá a la interfaz, y no a las clases que la implementan y mucho menos a los objetos instanciados. Por ejemplo, supongamos que todos los animales, sin excepción, emitieran el mismo sonido al bostezar. Entonces podríamos implementar un método estático para los bostezos,

```
interface Sonido {  
    void voz();  
    static void bostezo(){  
        System.out.println("¡Aaaauuuh!");  
    }  
}
```

Este método será accesible directamente desde la interfaz `Sonido`. Para invocarlo, tendremos que escribir,

```
Sonido.bostezo();
```

Las interfaces pueden heredar unas de otras y se pueden crear variables cuyo tipo es una interfaz, con las que podremos referenciar cualquier objeto de cualquier clase que la implemente. Por ejemplo, podremos crear la variable de tipo `Sonido`,

```
Sonido son;
```

Con ella podremos referenciar cualquier objeto de cualquier clase que tenga implementada la interfaz `Sonido`,

```
son = new Gato();
```

La sentencia,

```
son.voz();
```

escribirá en la pantalla el mensaje «¡Miau!», ya que se ejecuta el método `voz()`, tal como está implementado en la clase `Gato`. Esa misma variable puede referenciar a un objeto `Perro`,

```
son = new Perro();
```

con lo cual, la misma sentencia,

```
son.voz();
```

escribirá «¡Guau!», que es el mensaje implementado en la clase `Perro`.

Este es un ejemplo de una de las formas de polimorfismo de Java: la selección dinámica de métodos. La misma línea de código produce efectos distintos (ejecuta métodos distintos), dependiendo del objeto referenciado por la variable `son`. Se decide en tiempo de ejecución qué implementación concreta del método se va a ejecutar. Es importante saber que cuando usemos una variable de tipo `Sonido` para referenciar un objeto, solo tendremos acceso a los

métodos de ese objeto que pertenezcan a la interfaz **Sonido**. En este contexto, el objeto se manifiesta exclusivamente como un objeto que emite un sonido. De ahí el nombre de interfaz.

Como puede verse a partir de su definición, una interfaz es semejante a una clase abstracta. La diferencia es que una clase solo puede heredar de una clase abstracta, mientras que puede implementar más de una interfaz. Las clases abstractas están para ser heredadas por otras clases, mientras que las interfaces son implementadas por las clases. Por otra parte, mientras que una clase solo puede heredar de una clase, una interfaz puede heredar de más de una interfaz (entre las interfaces es posible la herencia múltiple).

La sintaxis general de la definición de una interfaz tiene la forma¹,

```
tipoDeAcceso interface NombreInterfaz {
    //atributos, son public, static y final por defecto:
    tipo atributo1 = valor1;
    ... //otros atributos

    //métodos abstractos, sin implementar:
    tipo metodo1(listaPar1);
    ... //otros métodos abstractos

    //métodos static, implementados:
    static tipo metodoEstatico1(listaPar1) {
        //cuerpo del método estático 1
    }
    ... //otros métodos estáticos

    //métodos por defecto, implementados:
    default tipo metodoDefault1(listaPar1) {
        //cuerpo de metodoDefault1
    }
    ... //otros métodos por defecto
}
```

donde *tipoDeAcceso* puede omitirse, en cuyo caso el acceso está restringido al paquete en el que está incluida, o ser **public** para que la interfaz pueda ser importada desde otro paquete mediante una sentencia **import**.

Los atributos, por defecto, son **public**, **final** y **static**, sin que haya que especificarlo de forma explícita en la definición. Se accede a ellos a través del nombre de la interfaz o de una clase que la implemente, pero no desde una instancia. Por ejemplo, si queremos mostrar la versión actual de la interfaz **Sonido**, podemos poner,

```
System.out.println(Sonido.version);
```

o bien,

```
System.out.println(Perro.version);
```

¹Igual que ocurre con las clases, se debe definir cada interfaz en un archivo propio dentro del paquete donde se va a usar.

Para que una clase implemente una interfaz, debe declararla en el encabezamiento, usando la palabra clave `implements`, e implementar todos los métodos abstractos de la interfaz en el cuerpo de definición de la clase.

```
tipoDeAcceso class NombreClase implements nombreInterfaz {  
    ...  
    public tipo metodoAbstracto1(listaPar1) {  
        ... //cuerpo del método 1  
    }  
  
    ... //otros métodos abstractos  
}
```

Una clase puede implementar más de una interfaz, para lo cual deberá escribirlas, separadas por comas, en el encabezamiento, e implementar todos los métodos abstractos de cada una de ellas.

```
tipoDeAcceso class nombreClase implements Interfaz1, Interfaz2,... {  
    ...  
}
```

A su vez, una interfaz puede ser implementada por más de una clase, como ya vimos con las clases `Gato` y `Perro`. Eso significa que todas ellas van a ofrecer el conjunto de funcionalidades declaradas en la interfaz, aunque cada una las implemente de forma distinta, según sus propias características.

9.2. Clases anónimas

Lo más común es que las interfaces sean implementadas por distintas clases de las que, por otra parte, se crearán diversos objetos. Sin embargo, hay ocasiones en que una determinada implementación de una interfaz es necesaria en un solo lugar. En ese caso no merece la pena definir una clase nueva para crear un solo objeto.

En su lugar, podemos crear un objeto de una clase sin nombre, es decir, anónima, donde se implementan los métodos de la interfaz. Como no dispondremos de nombre para la clase, tampoco lo tendremos para el constructor. Por eso usaremos un constructor con el nombre de la interfaz. Asimismo, el objeto será referenciado por una variable del tipo de la interfaz. Como ejemplo, crearemos una clase anónima que implemente la interfaz `Sonido`. Imaginemos que alguien ha encontrado un animal de una especie desconocida, que emite un extraño ruido. Mientras se identifica o no, para describir su sonido crearemos un objeto de una clase anónima que implementa la interfaz `Sonido`,

```
Sonido son = new Sonido() {  
    public void voz() {  
        System.out.println("¡Jajejijojuuuu!");  
    }  
};  
son.voz();
```

Por pantalla obtendremos el mensaje «Jajejijojuuuu!».

En realidad, no hemos definido ninguna clase. Lo que hemos hecho es crear un objeto sin clase, aunque es costumbre hablar de clases anónimas.

9.3. Interfaz Comparable

En programación hay operaciones tan necesarias y tan frecuentes, que merece la pena definir una interfaz que las declare. Una de esas operaciones es la de comparar dos valores para hacer búsquedas u ordenarlos. Con este fin, los desarrolladores de Java han implementado un par de interfaces, **Comparable** y **Comparator**. Aquí las vamos a estudiar a modo de ejemplo real que ilustre todo lo visto hasta ahora en el capítulo. Junto a ellas, para completar el tema de las comparaciones, también veremos el método `equals()`, aunque no pertenezca a ninguna interfaz especial, sino a la clase raíz **Object**, de la que heredan todas las clases de Java.

La interfaz **Comparable** consta de un único método abstracto,

```
int compareTo(Object ob);
```

Para que una clase implemente la interfaz **Comparable**, deberá tener implementado este método, que se comportará de la siguiente forma. Supongamos que queremos comparar dos objetos `ob1` y `ob2` de una determinada clase. Deberemos ejecutar la sentencia,

```
ob1.compareTo(ob2);
```

Si en una ordenación el objeto `ob1`, que invoca el método, va antes que el objeto `ob2` que se le pasa como parámetro, el método devolverá un número negativo; si va después, devolverá un número positivo y si se consideran iguales a efectos de ordenación, devolverá un cero.

- `ob1.compareTo(ob2) < 0` si `ob1` va antes que `ob2`.
- `ob1.compareTo(ob2) > 0` si `ob1` va después que `ob2`.
- `ob1.compareTo(ob2) = 0` si `ob1` es igual que `ob2`.

Veámoslo con un ejemplo. Supongamos que queremos ordenar una tabla de objetos de la clase **Persona**. Naturalmente, tendremos que escoger un criterio de ordenación. En este caso queremos ordenar por los números de identificación —`id`— crecientes,

```
//la clase Persona tendrá que implementar los métodos de Comparable
class Persona implements Comparable {
    int id;
    String nombre;
    int edad;

    Persona(int id, String nombre, int edad) { //constructor
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

```
@Override //la implementación debe declararse public
public int compareTo(Object ob) {
    int resultado;
    if (id < ((Persona) ob).id){ //this va antes que ob
        resultado = -1; //o cualquier número negativo
    } else if (id > ((Persona) ob).id) { //this va después que ob
        resultado = 1; //o cualquier número positivo
    } else { //this es igual que ob
        resultado = 0;
    }
    return resultado;
}

public String toString() {
    return "Id: " + id + " Nombre: " + nombre + " Edad: " + edad + "\n";
}
}
```

Si observamos la implementación de `compareTo()`, llama la atención el cast `Persona` aplicado al parámetro `ob`. La razón es que, para que la interfaz sea útil para cualquier clase de Java, es necesario que su parámetro de entrada sea de la clase `Object`, que es la más general posible. Esto obliga a que en el cuerpo de la función haya que aplicarle un cast que le diga al compilador que, en realidad, es un objeto `Persona`, pudiendo acceder así al atributo `id`. En la implementación del método, el `id` del objeto que llama a `compareTo()` es accesible directamente, ya que estamos accediendo al atributo desde el código del propio objeto (`this`) que hace la llamada. En cambio, `ob` es una referencia a un objeto externo que se le pasa como parámetro, y para acceder a su `id` deberemos poner el nombre `ob` del objeto, además del cast.

En casos como este, en que se usa un atributo de tipo numérico para comparar, hay una implementación mucho más sencilla para calcular el resultado de la comparación,

```
int compareTo(Object ob){
    return id - ((Persona)ob).id;
}
```

Téngase en cuenta que el número positivo (o negativo) devuelto no tiene por qué ser 1 (o -1).

Se deja al lector como ejercicio que verifique que es equivalente a la otra implementación.

Vamos a aplicar todo esto a la comparación de dos objetos `Persona`.

```
Persona p1 = new Persona(3, "Anselmo", 14);
Persona p2 = new Persona(1, "Josefa", 15);

int resultado = p1.compareTo(p2);
System.out.println(resultado);
```

se mostrará por pantalla un número positivo, ya que `p1` iría después de `p2` en una ordenación por número `id`. En cambio, si la tercera línea fuera,


```
int resultado=p2.compareTo(p1);
```

aparecería un número negativo.

La interfaz **Comparable** ha sido creada por los desarrolladores de Java y es reconocida por otras clases de la API. Entre ellas, la clase **Arrays**, donde se implementa el método **sort()**, que sirve para ordenar una tabla. Por ejemplo, si declaramos e inicializamos la tabla de números enteros,

```
int[] t = {5, 3, 6, 9, 3, 4, 1, 0, 10};
```

la sentencia,

```
Arrays.sort(t);
```

ordena la tabla **t** por el orden natural de sus elementos que, en el caso de los números enteros, es el orden creciente.

El método **sort()** no devuelve una copia ordenada de **t**, sino que ordena la tabla original. Si queremos mostrarla, podemos recurrir a otra función de la clase **Arrays**,

```
System.out.println(Arrays.toString(t));
```

que mostrará por pantalla,

```
[0, 1, 3, 3, 4, 5, 6, 9, 10]
```

Aquí hemos llamado al método **Arrays.toString()**, que devuelve una cadena de caracteres donde se muestran los elementos de la tabla entre corchetes y separados por comas.

El método **sort()** también sirve para ordenar una tabla de objetos de cualquier clase, con tal de que esta tenga implementada la interfaz **Comparable**. Utilizará el orden natural de la clase que, por definición, será el establecido por el método **compareTo()**. Por tanto, para usar la utilidad **sort()** con objetos de una determinada clase, esta deberá tener implementada la interfaz **Comparable**, que será la que establezca el criterio de ordenación natural de la clase.

Por ejemplo, vamos a ordenar la tabla de objetos **Persona**²,

```
Persona[] t = new Persona[] {
    new Persona(2, "Ana", 19),
    new Persona(5, "Jorge", 12),
    new Persona(1, "Juan", 10)
};
```

Para ordenarlos utilizaremos el método **sort()**, ya que **Persona** tiene implementada la interfaz **Comparable**, que determina que la ordenación natural es por id creciente,

```
t.sort();
```

Podemos mostrar la tabla ya ordenada, usando la sentencia,

```
System.out.println(Arrays.deepToString(t));
```

²Esta es una forma nueva de declarar una tabla de objetos, en este caso de la clase **Persona**, a la vez que se inicializa. Aquí es una tabla de tamaño tres, de la clase **Persona**, que inicializamos con los tres objetos creados con las sentencias **new**.

donde, en lugar de `toString()`, válido para tablas de tipos primitivos, hemos utilizado `deepToString()`, ya que nuestra tabla es de objetos `Persona`. Este método llamará al método `toString()` que hayamos implementado en la clase `Persona`, para mostrar los objetos individuales. Se obtendrá por pantalla,

```
[Id: 1 Nombre: Juan Edad: 10
,Id: 2 Nombre: Ana Edad: 19
,Id: 5 Nombre: Jorge Edad: 12
]
```

Muchas clases implementadas en la API de Java, como la clase `String`, implementan la interfaz `Comparable`, con lo cual podemos comparar sus objetos y ordenarlos por medio de `sort()`. En particular, las cadenas se comparan por medio de `compareTo()` siguiendo el orden alfabético creciente.

9.4. Interfaz Comparator

La interfaz `Comparable` proporciona un criterio de ordenación a la clase que la implemente, llamada ordenación natural, que es la que usan por defecto los distintos métodos de la API, como `sort()`. Pero es frecuente que tengamos que ordenar los objetos de la misma clase con distintos criterios, según las circunstancias. Por ejemplo, puede ser que necesitemos un listado de objetos `Persona` por orden alfabético de nombres, o por edades, en sentido creciente o decreciente. Para resolver este problema existe la interfaz `Comparator`. Para usarla habrá que importarla con la sentencia,

```
import java.util.Comparator;
```

Esta interfaz tiene un único método abstracto,

```
int compare(Object ob1, Object ob2)
```

Recibe como parámetros dos objetos que queremos comparar para determinar cuál va antes y cuál después en un proceso de ordenación. Devuelve un entero que será negativo si `ob1` va antes de `ob2`, positivo si va después y cero si son iguales. Llamamos comparador a cualquier objeto de una clase que implemente la interfaz `Comparator`. Necesitaremos una clase de comparadores distinta para cada criterio de comparación que queramos emplear para objetos de una clase determinada. Por ejemplo, si queremos ordenar una tabla de objetos de la clase `Persona` por orden de edad,

Implementaremos una clase comparadora para el atributo edad,

```
import java.util.Comparator;

public class ComparaEdades implements Comparator {
    @Override
    public int compare(Object ob1, Object ob2) {
        Persona p1 = (Persona) ob1;
        Persona p2 = (Persona) ob2;
        return p1.edad - p2.edad;
    }
}
```

No se debe olvidar la sentencia de importación. Para que la interfaz `Comparator` sirva para comparar objetos de cualquier clase, los parámetros de entrada tienen que ser lo más generales posible. Por eso son de tipo `Object`. Esto obliga a usar un cast en cada implementación, en este caso (`Persona`). Si la edad de `p1` es menor que la de `p2`, `p1` irá antes que `p2` en la ordenación y se devuelve un número negativo, y viceversa.

Los métodos de ordenación de la API de Java para distintas estructuras de datos, como las tablas de objetos, por ejemplo el método `sort()`, están sobrecargados y admiten como parámetro adicional un comparador que les diga el criterio de ordenación que deben emplear.

Como ejemplo vamos a crear una lista de tres personas, que vamos a ordenar por edades,

```
Persona[] t = new Persona[] {
    new Persona(2, "Ana", 19),
    new Persona(5, "Jorge", 12),
    new Persona(1, "Juan", 10)
};
```

Para ordenar por edades, necesitaremos un objeto comparador de edades,

```
ComparaEdades c = new ComparaEdades();
```

que pasaremos al método `sort()` como argumento, junto con la tabla que queremos ordenar,

```
Arrays.sort(t, c);
```

Para mostrar la tabla ordenada,

```
System.out.println(Arrays.deepToString(t));
```

mostrándose por pantalla,

```
[Id: 1 Nombre: Juan Edad: 10
, Id: 5 Nombre: Jorge Edad: 12
, Id: 2 Nombre: Ana Edad: 19]
```

En vez de declarar la variable `c`, podríamos haber creado el comparador en la propia llamada a la función `sort()`,

```
Arrays.sort(t, new ComparaEdades());
```

Otra opción, en caso de que se vaya a hacer la ordenación una sola vez, es crear una clase anónima en la llamada al método `sort()`,

```
Arrays.sort(t, new Comparator() {
    public int compare(Object ob1, Object ob2) {
        return ((Persona) ob1).edad - ((Persona) ob2).edad;
    }
});
```

```
System.out.println(Arrays.deepToString(t));
```

Si, en algún otro lugar del programa tuviéramos que mostrarlas ordenadas por orden alfabético de nombres, tendríamos que definir otra clase comparadora,

```
public class ComparaNombres implements Comparator {
    public int compare(Object ob1, Object ob2) {
        int resultado;
        String nombre1 = ((Persona) ob1).nombre;
        String nombre2 = ((Persona) ob2).nombre;
        resultado = nombre1.compareTo(nombre2);
        return resultado;
    }
}
```

Como puede observarse, para obtener el resultado, se llama al método `compareTo()` de la clase `String`, ya que se están comparando `nombre1` y `nombre2`, que son cadenas.

En clases más complejas, con más atributos, podrá haber más criterios posibles de ordenación. Para cada uno de ellos, si se va a usar, habría que definir una clase comparadora apropiada.

Ejercicios de interfaces

9.1. Implementar la interfaz `Pila`. Declararla e implementarla en las clases `PilaTabla` y `PilaLista`. Utilizando una referencia del tipo `Pila` con una cualquiera de las implementaciones, mostrar por pantalla, en orden inverso, 10 números introducidos por teclado.

Interfaz `Pila`

```
/*
 * La interfaz Pila contiene dos métodos que corresponden, cada uno, a sus operaciones
 * más importantes: apilar y desapilar. */
public interface Pila {
    void apilar(Integer elemento); //apila un elemento Integer
    Integer desapilar(); //desapila un elemento y lo devuelve
}
```

Clase `PilaTabla`

```
import java.util.Arrays;

/*
 * Vamos a implementar una estructura pila para valores Integer utilizando una tabla,
 * donde se almacenarán los elementos apilados. Las dos funciones fundamentales de una
 * pila son apilar y desapilar, que consisten en añadir o quitar elementos de un mismo
 * punto, llamado índiceCima, de manera que se desapila siempre el último elemento que
 * se apiló. La tabla se redimensionará cuando falte sitio para elementos nuevos.
 * Nosotros añadiremos un par de métodos que nos permitan visualizar la pila y algunos
 * más, privados, para uso interno de la pila.*/
public class PilaTabla implements Pila {

    private int indiceCima;
    private Integer[] tabla;

    public PilaTabla() {
        indiceCima = -1; //corresponde a una pila vacía
        tabla = new Integer[10]; //por defecto creamos una pila de capacidad 10
    }
}
```

```

public PilaTabla(int capacidadInicial) {
    tabla = new Integer[capacidadInicial];//si queremos inicializar con una
    //capacidad inicial distinta
}

private boolean pilaVacía() {
    return indiceCima == -1;
}

private boolean pilaLlena() {
    return indiceCima == tabla.length - 1;
}

Integer cima() {
    Integer elementoCima = null;
    if (!pilaVacía()) {
        elementoCima = tabla[indiceCima];
    }
    return elementoCima;
}

//apilamos añadiendo el elemento en el primer lugar libre de la tabla,
//empezando por el principio (índice 0)
@Override
public void apilar(Integer elemento) {
    if (pilaLlena()) {
        tabla = Arrays.copyOf(tabla, tabla.length + 10);
    }
    indiceCima++;//siempre es el índice del último elemento apilado
    tabla[indiceCima] = elemento;
}

//desapilamos extrayendo el elemento de la cima. Si la pila está vacía,
//devuelve null
@Override
public Integer desapilar() {
    Integer elemento = null;
    if (!pilaVacía()) {
        elemento = tabla[indiceCima];
        indiceCima--;
    }

    return elemento;
}

//implementamos toString() para visualizar el estado de la pila, aunque
//no es una función propia de pilas
@Override
public String toString() {
    String resultado = "";
    for (int i = 0; i <= indiceCima; i++) {
        resultado += tabla[i] + " ";
    }
    resultado += "(cima)";

    return resultado;
}
}

```

Clase PilaLista

```

/*
 * Vamos a implementar una estructura de pila para Integer usando objetos de la clase
 * Lista para guardar los datos que se apilan. Por razón de eficiencia, la cima será el
 * final de la lista, evitando así mover los datos apilados previamente. */

```

```
public class PilaLista implements Pila {
    private int indiceCima;
    private Lista lista;

    public PilaLista() {
        indiceCima = -1; //corresponde a una pila vacía
        lista = new Lista(); //por defecto creamos una pila de capacidad 10
    }

    public PilaLista(int capacidadInicial) {
        lista = new Lista(capacidadInicial); //si queremos inicializar con una
        //capacidad inicial distinta
    }

    private boolean pilaVacía() {
        return indiceCima == -1;
    }

    private boolean pilaLlena() {
        return lista.listaLlena();
    }

    int cima() {
        return lista.numeroElementos() - 1;
    }

    //apilamos añadiendo el elemento al final de la lista
    @Override
    public void apilar(Integer elemento) {
        lista.insertarFinal(elemento);
        indiceCima++; //siempre es el índice del último elemento apilado
    }

    //desapilamos extrayendo el elemento de la cima. Si la pila está vacía, es
    //porque la lista también lo está y devuelve null
    @Override
    public Integer desapilar() {
        Integer elemento = lista.eliminar(indiceCima);
        if (elemento != null) {
            indiceCima--;
        }
        return elemento;
    }

    //implementamos toString() para visualizar el estado de la pila, aunque
    //no es una función propia de pilas
    @Override
    public String toString() {
        return lista.toString() + "(cima)";
    }
}
```

Programa principal

```
/*
 * Debemos incluir las clases PilaTabla, PilaLista y Lista. */
public class Main {

    public static void main(String[] args) {
        //optamos por la implementación de pilas con tablas
        Pila p = new PilaTabla();
        for (int i = 0; i < 10; i++) {
            p.apilar(i);
        }
        Integer num = p.desapilar();
    }
}
```

```

        while (num != null) {
            System.out.println(num);
            num = p.desapilar();
        }
    }
}

```

9.2. Implementar la interfaz Cola e implementarla en las clases ColaTabla y ColaLista. Utilizando una referencia del tipo Cola y una de las implementaciones, encolar 10 números entre 1 y 100, generados aleatoriamente y desencolarlos para mostrarlos por pantalla.

Interfaz Cola

```

/* La interfaz Cola tiene dos métodos, encolar y desencolar. */
public interface Cola {

    void encolar(Integer elemento);
    Integer desencolar();
}

```

Clase ColaTabla

```

import java.util.Arrays;

/*
 * Implementamos una estructura cola para Integer. A diferencia de las pilas en las colas
 * los primeros elementos que entran son los primeros en salir, como en las colas del
 * cine. Por tanto, hay dos posiciones a tener en cuenta: el principio de la cola, de
 * donde se desencola (se extraen los elementos), y el final de la cola, donde se encola
 * (se insertan los elementos que llegan nuevos). Aquí la vamos a implementar con una
 * tabla donde se guardarán los elementos encolados. El primer elemento de la cola será
 * el elemento de índice 0 de la tabla, con lo que solo tendremos que guardar el índice
 * del último elemento de la cola. Las operaciones básicas de una cola son encolar y
 * desencolar, aunque añadiremos otras auxiliares.
 */

public class ColaTabla implements Cola {

    private int ultimo;
    private Integer[] tabla;

    public ColaTabla() {
        ultimo = -1; //cola vacía
        tabla = new Integer[10]; //por defecto, la capacidad inicial es 10
    }

    //por si queremos otra capacidad inicial
    public ColaTabla(int capacidadInicial) {
        ultimo = -1;
        tabla = new Integer[capacidadInicial];
    }

    boolean colaVacía() {
        return ultimo == -1;
    }

    boolean colaLlena() {
        return ultimo == tabla.length - 1;
    }
}

```

```
@Override
public void encolar(Integer elemento) {
    if (colaLlena()) {//si falta sitio para encolar, aumento la capacidad
        tabla = Arrays.copyOf(tabla, tabla.length + 10);
    }
    ultimo++;
    tabla[ultimo] = elemento;
}

@Override
public Integer desencolar() {
    Integer elemento = null;
    if (!colaVacia()) {
        elemento = tabla[0];//al desencolar el primero, deben desplazarse
        //un lugar hacia el principio todos los demás
        for (int i = 0; i < ultimo; i++) {
            tabla[i] = tabla[i + 1];
        }
        ultimo--;
    }
    return elemento;
}

//implementamos toString() para visualizar el estado de la pila, aunque
//no es una función propia de pilas
@Override
public String toString() {
    String resultado = "(primero) ";
    for (int i = 0; i <= ultimo; i++) {
        resultado += tabla[i] + " ";
    }
    resultado += "(ultimo)";
    return resultado;
}
}
```

Clase ColaLista

```
/*
 * Vamos a implementar una cola para Integer usando la clase Lista, con todas sus funcio-
 * nalidades. Encolaremos al final de la lista y desencolaremos del principio. El control
 * del espacio disponible y de la cola llena o vacía se deja en manos de la Lista
 */
public class ColaLista implements Cola {
    private Lista lista;

    public ColaLista() {
        lista = new Lista();
    }

    public ColaLista(int capacidadInicial) {
        lista = new Lista(capacidadInicial);
    }

    //encolamos al final de la lista
    @Override
    public void encolar(Integer elemento) {
        lista.insertarFinal(elemento);
    }

    //desencolamos del principio
    @Override
    public Integer desencolar() {
        return lista.eliminar(0);
    }
}
```



```

@Override
public String toString() {
    return "(primero) " + lista.toString() + "(último)";
}
}

```

Programa principal

```

/*
 * Debemos incluir las clases PilaTabla, PilaLista y Lista. */
public class Main {

    public static void main(String[] args) {
        //optamos por la implementación de pilas con tablas
        Pila p = new PilaTabla();
        for (int i = 0; i < 10; i++) {
            p.apilar(i);
        }
        Integer num = p.desapilar();
        while (num != null) {
            System.out.println(num);
            num = p.desapilar();
        }
    }
}

```

9.3. Diseñar la clase **Cliente** con los siguientes atributos: DNI, nombre, edad y saldo. Implementar un constructor y los métodos **toString()** y **equals()** (este último basado en el DNI). Implementar la interfaz **Comparable** con un criterio de ordenación basado también en el DNI. Implementar, asimismo, un comparador para hacer ordenaciones basadas en el nombre y otro basado en la edad. Crear una tabla con 5 clientes y mostrarlos ordenados por DNI, por nombre y por edad.

Clase **Cliente**

```

public class Cliente implements Comparable {
    String dni;
    String nombre;
    int edad;
    double saldo;

    //constructor
    public Cliente(String dni, String nombre, int edad, double saldo) {
        this.dni = dni;
        this.nombre = nombre;
        this.edad = edad;
        this.saldo = saldo;
    }

    @Override
    public boolean equals(Object otro) {
        return dni.equals(((Cliente) otro).dni);
    }

    @Override
    public String toString() {
        return "\nDni: " + dni + " Nombre: " + nombre + " Edad: " + edad
            + " Saldo: " + saldo;
    }
}

```

```
//remite al criterio de comparación natural de String, que es el de los dni
@Override
public int compareTo(Object otro) {
    return dni.compareTo(((Cliente) otro).dni);
}
}
```

Clase ComparaEdades

```
import java.util.Comparator;

/*
 * El método compare() tiene que devolver un número positivo si el primer argumento va
 * antes que el segundo, negativo si va después y cero si son iguales. Este resultado se
 * consigue restando el primero menos el segundo.
 */
public class ComparaEdades implements Comparator {

    @Override
    public int compare(Object o1, Object o2) {
        return ((Cliente) o1).edad - ((Cliente) o2).edad;
    }
}
```

Clase ComparaNombres

```
import java.util.Comparator;

public class ComparaNombres implements Comparator {

    @Override
    public int compare(Object o1, Object o2) {
        return ((Cliente) o1).nombre.compareTo(((Cliente) o2).nombre);
    }
}
```

Programa principal

```
import java.util.Arrays;

public class Main {

    //programa principal para comprobar
    public static void main(String[] args) {
        Cliente[] tablaClientes = new Cliente[5];

        //insertamos algunos los datos de 5 clientes
        tablaClientes[0] = new Cliente("435", "Jorge", 34, 6200);
        tablaClientes[1] = new Cliente("235", "Ana", 20, 3200);
        tablaClientes[2] = new Cliente("125", "Julio", 20, 4100);
        tablaClientes[3] = new Cliente("465", "Hector", 30, 5200);
        tablaClientes[4] = new Cliente("115", "Anne", 28, 1200);

        //Ordenamos por el criterio de orden natural, que es alfabético de DNI,
        Arrays.sort(tablaClientes);
        System.out.println("Por DNI: " + Arrays.toString(tablaClientes));

        //ordenamos por orden alfabético de nombres
        Arrays.sort(tablaClientes, new ComparaNombres());
        System.out.println("Por nombres: " + Arrays.toString(tablaClientes));
    }
}
```

```
        //ordenamos por orden creciente de edades
        Arrays.sort(tablaClientes, new ComparaEdades());
        System.out.println("Por edades: " + Arrays.toString(tablaClientes));
    }
}
```

Ejercicios propuestos

- 9.1. Crear una tabla de 20 números **Integer** aleatorios entre 1 y 100. Implementar un comparador que ordene los números en orden decreciente. Utilizarlo para ordenar en sentido decreciente la tabla de enteros.

Capítulo 10

Ficheros de texto

En la mayoría de los programas que se implementan, en un momento u otro hay que interaccionar con alguna fuente de datos, como un archivo del disco duro, un DVD o un dispositivo de red, ya sea para guardar información o para recuperarla. Java implementa una serie de clases llamadas flujos, encargadas de comunicarse con los dispositivos de almacenamiento. El funcionamiento de estos flujos, desde el punto de vista del programador, no depende del tipo de dispositivo hardware con el que está asociado, lo que nos liberará del trabajo que supone tener en cuenta las características físicas de cada dispositivo.

Los flujos pueden ser de entrada o de salida, según sean para guardar o recuperar información. Por otra parte, atendiendo al tipo de datos que se transmiten, los flujos son de dos tipos:

- **Carácter:** si se asocia a archivos u otras fuentes de tipo texto.
- **Binarios:** si transmiten bytes, enteros entre 0 y 255. Esto, en realidad, permite manipular cualquier tipo de datos.

Vamos a empezar por los flujos de tipo texto y en otro capítulo, más adelante, nos dedicaremos a los archivos —flujos— de tipo binario, algo más complejos, pero antes vamos a estudiar las excepciones.

Cuando se intenta acceder a un fichero ocurre a veces que se produce un error debido a una ruta de acceso mal escrita o alguna otra causa. En Java los errores se llaman *excepciones*.

10.1. Excepciones

Los errores en los programas son prácticamente inevitables, ya sean originados por códigos deficientes, entrada de datos o parámetros incorrectos, archivos inexistentes, discos defectuosos, etc.

En la mayoría de los lenguajes de programación, la manipulación de los errores suele ser complicada y confusa. Su código se mezcla con el del resto del programa, haciéndolo poco claro. Suele estar destinado solo a evitar el error y no a controlar la situación una vez que el error se ha producido.

Java, un lenguaje que se adapta a las condiciones de internet, donde los programas se ejecutarán en máquinas remotas, casi siempre manipuladas por personal no cualificado, aporta un enfoque nuevo, tratando de evitar, en lo posible, que el programa se interrumpa a causa del error. Para ello, no basta con evitar que se produzcan los errores, sino que hay que implementar los medios para que un programa se recupere de las condiciones generadas por un error que no se ha podido evitar. Eso depende del tipo de error, y no siempre es posible.

Cuando, en la ejecución de un programa, se produce una situación anormal —un error—, que interrumpe el flujo normal de ejecución, el método que se esté ejecutando genera un objeto de la clase **Throwable** («arrojable»), que contiene información del error, de su causa y del contexto del programa en el momento en que se produce, y lo entrega al sistema de tiempo de ejecución —la máquina virtual—. Este objeto es susceptible de ser capturado —ya veremos cómo— por el programa y analizado para dar una respuesta, si procede. En caso contrario, el programa se interrumpe y el sistema de tiempo de ejecución muestra una serie de mensajes que describen el error, como ocurre en otros lenguajes de programación.

Hay errores lo bastante graves para que sea preferible que el programa se interrumpa. Por ejemplo, errores derivados de problemas de hardware. Si intentamos leer de un disco defectuoso, se producirán errores de los que es difícil, si no imposible, recuperarse. Estos y otros errores relacionados directamente con la máquina virtual y no con el código de nuestro programa, arrojan objetos de la clase **Error**, una subclase de **Throwable**, de los que no nos vamos a ocupar, ya que poco se puede hacer con ellos a la hora de programar.

Pero hay otra clase de errores más habituales y menos graves, como entradas de datos de tipo equivocado, aperturas de ficheros con ruta de acceso errónea o las operaciones aritméticas no permitidas. A estos errores se les llama *excepciones*, y producen un objeto de la clase **Exception**, otra subclase de **Throwable**. De estas excepciones vamos a tratar aquí, ya que son manipulables a través del código. Para ello se usan los bloques **try**, **catch** y **finally**.

Cuando sabemos que en un determinado fragmento de código se puede producir una excepción, lo encerramos dentro de un bloque **try**. Por ejemplo, si en una división entre las variables **a** y **b** sospechamos que el divisor **b** podría ser cero, escribimos,

```
try {  
    int c;  
    c = a / b;  
}
```

Con esta estructura estamos sometiendo a observación al bloque de código encerrado entre llaves. Si salta una excepción en ese bloque, deberá ser capturada por un bloque **catch** de la siguiente forma,

```
catch(ArithmeticException e) {  
    System.out.println("Error: división por cero");  
}
```

Cuando se produce la excepción y es capturada, se interrumpe la ejecución del código del bloque **try**, saltando a la primera línea del bloque **catch**. Cuando se termina de ejecutar dicho bloque, continúa en la línea inmediatamente posterior a la estructura **try-catch**.

La palabra clave `catch` va seguida de unos paréntesis donde se encierra un parámetro de la clase de excepción que puede atrapar; en este caso, una excepción aritmética. El parámetro `e` se puede usar como variable local dentro del bloque, como en los métodos. Hace referencia al objeto de la excepción generada, que contiene toda la información sobre el error producido. Entre sus métodos está `getMessage()`, que nos muestra un mensaje descriptivo del error. Podríamos haber puesto,

```
catch(ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

o incluso,

```
catch(ArithmeticException e) {
    System.out.println(e);
}
```

que hace una llamada a `toString` de la clase de la excepción, donde también se describe el error (en inglés).

Los bloques `try` y `catch` deben ir uno a continuación del otro, sin ningún código en medio,

```
try {
    ... //bloque try
} catch(TipoExcepción nombreParámetro) {
    ... //bloque catch
}
```

Es importante señalar que en el bloque `catch` solo se recogerán las excepciones del tipo declarado entre los paréntesis o de una subclase. En el ejemplo anterior podríamos haber escrito,

```
try {
    c = a / b;
} catch(Exception e) {
    System.out.println("Error: división por cero");
}
```

ya que `ArithmeticException` es una subclase de `Exception`.

Esto nos permitiría recoger otros tipos de excepción en el mismo `catch` pero, para ese caso, es mejor escribir más de un bloque `catch`. Con un mismo bloque `try` se pueden poner tantos bloques `catch` como deseemos, siempre que vayan seguidos,

```
try {
    ... //bloque try
} catch(tipoExcepción1 nombreParámetro1) {
    ... //bloque catch
} catch(tipoExcepción2 nombreParámetro2) {
    ... //bloque catch
} ...
```

Cuando salta una excepción en el bloque `try`, se compara con el tipo de primer bloque `catch`. Si coincide con él o es una subclase, se ejecuta dicho bloque y continúa el programa después del último bloque `catch`. Si no coincide, se compara con el tipo del segundo bloque, y así sucesivamente, hasta que se encuentra un bloque cuyo parámetro coincida o sea una superclase de la excepción generada, de forma que solo se ejecuta un bloque `catch`, el primero cuyo tipo sea compatible.

Aquí hay que tener cuidado de no poner antes un `catch` con una excepción que sea superclase de otra que vaya más abajo, pues el bloque de esta última nunca se ejecutará. Por ejemplo,

```
try {  
    c = a / b;  
} catch (Exception e) {  
    System.out.println("Estoy en el primer catch");  
} catch (ArithmeticException e) {  
    System.out.println("Estoy en el segundo catch");  
}
```

Si `b` vale cero se producirá una excepción de tipo `ArithmeticException`, pero al ser un subtipo de `Exception`, dará afirmativo la comparación con el primer `catch`, cuyo bloque será el que se ejecute, apareciendo el mensaje: «Estoy en el primer `catch`». A continuación, la ejecución seguirá después del último bloque, de modo que el segundo bloque `catch` es inútil, ya que jamás se va a ejecutar. De hecho, en nuestro ejemplo, como toda excepción es subclase de `Exception`, ningún bloque `catch` que añadamos al final se va a ejecutar nunca.

Por otra parte, es posible que procesemos de igual manera distintas excepciones; en este caso existe la posibilidad de capturar más de un tipo de excepción con un único bloque `catch`,

```
catch (tipoExcepción1 | tipoExcepción2 | ... e) {  
    ...  
}
```

Aquí, la barra vertical equivale a una disyunción lógica *O*. Se pueden añadir tantos tipos de excepción como se quiera, separados por barras verticales. El significado es: «si salta una excepción del tipo `tipoExcepción1` o `tipoExcepción2` o ..., ejecutar este bloque `catch`, donde la excepción será referenciada con la variable `e`».

Todo código en Java forma parte de un método que, en última instancia, puede ser el método `main`. Cuando, dentro de un método, se produce una excepción, podemos manipularla por medio de una estructura `try-catch` como hemos hecho hasta ahora, pero hay otro enfoque posible. Todo método, salvo `main`, se escribe para ser llamado desde alguna línea de código de otro método. Durante la ejecución de ese otro método, la excepción saltará cuando se haga la llamada a nuestro primer método. Entonces, cabe la posibilidad de atrapar la excepción desde el método que hace la llamada, encerrando la línea de código en cuestión dentro de su correspondiente bloque `try-catch`. Para ello, debemos eliminar el bloque `try-catch` que escribimos en la implementación del método invocado y especificar, en su declaración, que contiene código donde se puede producir la excepción. Veámoslo con un ejemplo. Supongamos que en el método `metodo1` se puede dar una división por cero. Por

otra parte, un segundo método `metodo2` llama a `metodo1`. Lo que hemos hecho hasta ahora es,

```
void metodo1(int a, int b) {
    int c;
    try {
        c = a / b;
    } catch(ArithmeticException e) {
        System.out.println("División por cero");
    }
    System.out.println("a/b = " + c);
}
```

El `metodo2`, que llama a `metodo1`, podría ser,

```
void metodo2() {
    int x, y;
    ...
    metodo1(x, y);
    ...
}
```

Si la variable `y` es cero, la excepción producida es capturada y manipulada en origen (en `metodo1`) antes de que la ejecución vuelva al método que llama (`metodo2`).

Pero podríamos hacerlo de otra forma. Primero, eliminamos el bloque `try-catch` de `metodo1` y lo declaramos como susceptible de producir una `ArithmeticException`. Esto se implementa por medio de la palabra clave `throws` en su encabezamiento,

```
void metodo1(int a, int b) throws ArithmeticException {
    int c;
    c = a / b;
    System.out.println("a/b = " + c);
}
```

Con esto estamos declarando que, dentro de la función, puede producirse una excepción aritmética y que deberá ser manipulada por código externo, implementado en el método (en nuestro caso `metodo2`) que llame a `metodo1`. Además, esta particularidad, que forma parte de la definición de `metodo1`, deberá constar en la documentación que la acompaña para que los usuarios del método sepan a qué atenerse. La implementación de `metodo2`, por tanto, deberá hacerse cargo de la excepción,

```
void metodo2() {
    int x, y;
    ...
    try {
        metodo1(x, y);
    } catch(ArithmeticException e) {
        System.out.println("División por cero");
    }
    ...
}
```

Por supuesto, `metodo2` podría «pasar» la excepción a un tercer método que la invoque, etcétera.

Una estructura **try-catch** supone una bifurcación en el programa. A menudo estamos interesados en que una serie de líneas de código se ejecuten, tanto si se produce una excepción como si no, sin tener que duplicarlas; por ejemplo, para cerrar un archivo en el que estábamos escribiendo. A veces, un bloque **try** termina con un **return**. Si quisiéramos cerrar un archivo antes de volver del método, tendríamos que escribir el código correspondiente dentro del bloque **try** antes del **return** y repetirlo fuera. Esto se evita con un bloque **finally**, que se coloca al final de los bloques **catch**, si los hay (está permitida una estructura **try-finally**, sin **catch**). El código dentro de **finally** se ejecuta, tanto si se ha producido una excepción dentro del bloque **try**, como si no. No importa si hay un **return** dentro del bloque **try**; el bloque **finally** se ejecutará antes de retornar del método donde se encuentra.

```
try {  
    ... //bloque trabajo con archivos  
    return  
  
} catch(IOException e) {  
    ... //bloque si salta excepción  
  
} finally {  
    ... //código para cerrar archivos  
}  
...  
return;
```

El bloque **finally** se ejecuta incluso si no salta la excepción, antes de ejecutarse el **return** del bloque **try**, a pesar de que figura después de dicha sentencia.

Al principio distinguimos entre las excepciones (clase **Exception**) y los errores propiamente dichos —clase **Error**—. Pero, entre las excepciones, hay un grupo especialmente importante, ya que son predecibles a partir del código y es fácil recuperarse de ellas. Tanto es así que el propio compilador sabe dónde se pueden producir y nos obliga a manipularlas, ya sea por medio de estructuras **try-catch** o declarándolas en el encabezamiento de los métodos (mediante **throws**).

Este grupo de excepciones, llamadas *excepciones comprobadas* (*checked exceptions*), entre las que se encuentran las más comunes, generalmente con un origen fuera del programa (entradas de datos, nombres de ficheros incorrectos, etc.), se dice que están sometidas al requisito de «atrapar o especificar» (*catch or specify*), es decir, o implementamos el bloque **try-catch**, o especificamos la excepción en la declaración del método por medio de **throws**. Como el compilador nos exige su tratamiento (si no lo hacemos, genera un error de compilación), no tenemos que preocuparnos por saber cuáles son exactamente aunque, con la práctica, acabamos familiarizándonos con las más importantes: **IOException**, **FileNotFoundException**, **NumberFormatException**, **ClassCastException**, etc.

Junto a ellas se encuentran las «no comprobadas» (*unchecked exceptions*), como las que hemos usado en nuestro ejemplo, **ArithmeticException**, por errores aritméticos, o **ArrayIndexOutOfBoundsException**, que se produce cuando intentamos salirnos de los límites de una tabla. Dichas excepciones suelen estar asociadas a malas prácticas de programa-

ción. Por tanto, más que tratarlas con una estructura **try-catch** para recuperar la ejecución del programa, hay que repasar y corregir el código para evitar que se vuelvan a producir.

10.2. Flujos de entrada

Los flujos de entrada de tipo texto heredan de la clase **InputStreamReader**. Las clases de entrada de texto tienen siempre un nombre que termina en **Reader**. Nosotros usaremos flujos del tipo **FileReader**. El constructor es,

```
FileReader(String nombreArchivo)
```

al que se le pasa, como parámetro, el nombre del archivo al que se quiere asociar el flujo para su lectura. Este nombre puede llevar incluida la ruta de acceso si el archivo no está en la carpeta de trabajo. Por ejemplo,

```
FileReader in = new FileReader("C:\\programas\\prueba.txt");
```

crea un flujo de texto asociado al archivo *prueba.txt*, que se halla en la carpeta *programa* de la unidad *C*. No hay que olvidar que, para imprimir la barra invertida, hay que escribirla doble (\\), ya que escrita de forma simple se emplea para las secuencias de escape (p. ej., '\n' significa un cambio de línea). Cuando estamos trabajando en una plataforma Linux, las rutas de acceso son distintas. Por ejemplo,

```
FileReader in = new FileReader("/home/pedro/programas/prueba.txt");  
//la barra hacia delante (/) se puede escribir simple
```

La apertura de un fichero puede arrojar una excepción del tipo **IOException** cuando el archivo no se abre por alguna razón¹. Por ejemplo, puede que el archivo que queremos abrir para lectura no exista, o que no tenga la ruta de acceso especificada. Por ello, dicha operación siempre deberá ir dentro de la estructura **try-catch** correspondiente. Cuando se abre un archivo, el cursor se posiciona al principio, apuntando al primer carácter.

Una vez que se ha abierto el fichero para lectura, podremos leer del flujo asociado, usando el siguiente método,

int read(): que lee y devuelve del fichero. El carácter (en *Unicode*) lo devuelve como un entero, de forma que cada carácter ocupa los dos bytes menos significativos del entero devuelto (los enteros ocupan 4 bytes). Si queremos recuperar el carácter que lleva dentro cada entero, debemos aplicarle un *cast*, ya que la conversión, al ser de estrechamiento, no es automática. Sabremos que hemos llegado al final del archivo cuando **read()** devuelva **-1**, que no corresponde a ningún carácter.

A medida que vamos llamando al método **read()**, el cursor va avanzando dentro del archivo, apuntando al siguiente carácter. Una vez que terminemos de leer del flujo, hay que cerrarlo con el método,

void close(): cierra el flujo de entrada con objeto de completar las lecturas pendientes y liberar el archivo.

¹En realidad, el constructor de **FileReader** puede arrojar una excepción **FileNotFoundException**, que hereda de **IOException**, y que podría usar para la apertura del fichero, separándola del resto del código.

Pongamos un ejemplo: queremos leer el archivo *Main.java*, que es de texto, de uno de los proyectos que ya hemos terminado. Lo copiamos de su ubicación original, en su carpeta correspondiente y lo pegamos en la carpeta del proyecto actual que estamos implementando en NetBeans (la carpeta que lleva el mismo nombre que nuestro proyecto), junto a los archivos *build.xml*, *manifest.mf*, etc. Con esta ubicación, al crear el flujo de entrada, basta con poner el nombre del archivo, sin ruta de acceso:

```
try {
    FileReader in = new FileReader("Main.java");
    ...
} catch(IOException ex) {
    System.out.println(ex.getMessage());
}
```

Una vez abierto el flujo, podemos leer de él. Por ejemplo, si queremos construir una cadena con todo el texto del archivo y mostrarla por pantalla,

```
try {
    FileReader in = new FileReader("Main.java");
    String texto = "";
    int c = in.read();
    while(c != -1) { //mientras no lleguemos al final del archivo
        texto = texto + (char)c; //concatenamos convirtiendo c a char
        c = in.read(); //entero leído a carácter (char)
    }
    in.close(); //siempre hay que cerrar el flujo
} catch(IOException ex) {
    System.out.println(ex.getMessage());
}
System.out.println(texto); //mostramos
```

La variable `texto` contiene todo el texto de *Main.java*, incluidos los cambios de línea. `FileReader` nos permite leer cualquier archivo de texto plano creado con un editor de texto (no con un procesador de texto, como Word). Sin embargo, por razones de eficiencia, no se suele usar tal cual. Normalmente se usan flujos de la clase `BufferedReader`, que no es más que un `FileReader` filtrado para asociarle un *búfer* (espacio reservado para almacenamiento temporal) en memoria. Esto permite hacer lecturas en el dispositivo físico (el disco, por ejemplo) de grupos de caracteres, en vez de caracteres individuales, que son colocados en cola en el *búfer*, a la espera de que el programa los vaya reclamando. Con ello se reduce el número de accesos al disco, que es una operación extremadamente lenta. Para crear un `BufferedReader`, basta pasarle al constructor un objeto `FileReader`,

```
BufferedReader in = new BufferedReader(new FileReader("Main.java"));
```

Ahora, el flujo `in` dispone del método `read()` para hacer las lecturas de caracteres individuales pero, además, al tener un *búfer* asociado, puede hacer lecturas de líneas completas con el método,

String readLine(): que devuelve una cadena con la línea leída (que concluye en el siguiente carácter fin de línea, que descarta). Al llegar al final del fichero, `readLine()` devuelve `null`.

El programa anterior lo podríamos implementar ahora con `readLine()`,

```
try {
    BufferedReader in = new BufferedReader(new FileReader("Main.java"));
    String texto="";
    String linea = in.readLine();

    while (linea != null) { //mientras no llegue al final del archivo
        texto = texto + linea + '\n'; //linea no incluye '\n'
        linea = in.readLine();
    }
    in.close();

} catch(IOException ex) {
    System.out.println(ex.getMessage());
}
System.out.println(texto);
```

10.3. Flujos de salida

Si en vez de leer de un archivo de texto queremos escribir en él, necesitaremos un flujo de salida de texto. Para ello, crearemos un objeto de la clase `FileWriter`, que hereda de `OutputStreamWriter`. Las clases de salida de texto tienen un nombre que acaba en `Writer`. Los constructores de `FileWriter` son,

```
FileWriter(String nombreArchivo)
FileWriter(String nombreArchivo, boolean append)
```

donde *nombreArchivo*, como ya ocurría en `FileReader`, puede contener la ruta de acceso. El primer constructor destruye la versión anterior del archivo y escribe en él desde el principio. Sin embargo, el booleano `append`, si vale `true`, nos permite añadir texto al final del archivo, respetando el contenido anterior.

La apertura de un `FileWriter` puede generar una excepción del tipo `IOException`, que habrá que tratar con el `try-catch` correspondiente.

Como hicimos con `FileReader`, para mejorar el rendimiento usaremos una versión con *búfer*, `BufferedWriter`, a cuyo constructor se le pasa como parámetro un flujo de salida. Por ejemplo,

```
BufferedWriter out = new BufferedWriter(new FileWriter("salida.txt"));
```

Los métodos de que disponemos son:

void write(int caracter): escribe un carácter en el archivo.

void write(String cadena): escribe una cadena en el archivo.

void newLine(): escribe un salto de línea en el fichero. Se deben evitar los saltos de línea escribiendo secuencias de escape como `'\n'`, que son sensibles a la plataforma utilizada.

void flush(): vacía el *búfer* de salida, escribiendo en el fichero los caracteres pendientes.

void close(): cierra el flujo de salida, vaciando el *búfer* y liberando el recurso correspondiente.

Veamos un ejemplo de un archivo donde guardamos un par de líneas del Quijote, la primera carácter a carácter y la segunda, en una sola sentencia,

```
try {
    BufferedWriter out = new BufferedWriter(new FileWriter("quijote.txt"));

    String cad = "En un lugar de la mancha,"; //primera línea
    for (int i = 0; i < cad.length(); i++) {
        out.write(cad.charAt(i));
    }

    cad = "de cuyo nombre no quiero acordarme."; //segunda línea
    out.newLine(); //cambio de línea en el archivo
    out.write(cad);
    out.close(); //hacemos que se vacíe el búfer y se escriba en el archivo

} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```

Es muy común olvidarse de cerrar el flujo. El resultado puede ser encontrarse un archivo vacío, es decir, sin ningún carácter escrito. Esto es porque los caracteres se han guardado en el *búfer*, pero no han llegado a escribirse en el archivo antes de que el programa termine.

A partir de la versión 7 de Java, disponemos de una estructura para cerrar archivos o liberar cualquier recurso, sin necesidad de usar **finally**. Se trata de la estructura **try-catch-resources**. Por ejemplo, si deseamos abrir un archivo de texto llamado *nombres.txt*, para lectura, y queremos estar seguros de que cuando terminemos de trabajar con él se cerrará siempre (en cualquier caso, ocurra lo que ocurra),

```
BufferedReader in;
try(in = new BufferedReader(new FileReader("nombres.txt")) {
    ... //código que manipula los archivos
} catch(IOException e) {
    System.out.println(e);
}
```

Tanto si se produce la excepción como si no, el flujo **in** se cierra automáticamente al terminar de ejecutarse la estructura **try-catch**.

Ejercicios de ficheros de texto

- 10.1. Realizar un programa que solicite al usuario el nombre de un fichero de texto y muestre su contenido en pantalla. Si no se proporciona ningún nombre de fichero, la aplicación utilizará por defecto *prueba.txt*.

```
import java.io.*;
import java.util.Scanner;

/* Vamos a leer el fichero, carácter a carácter. Mientras no alcancemos
 * el final iremos leyendo y mostrando.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final String POR_DEFECTO = "prueba.txt"; //constante con el fichero por defecto

        System.out.println("Escriba nombre del fichero (por defecto prueba.txt):");
        String nombreFichero = sc.nextLine();
        if (nombreFichero.isEmpty()) { //si nombreFichero está vacío
            nombreFichero = POR_DEFECTO; //asignamos el valor por defecto
        }

        try {
            BufferedReader f = new BufferedReader(new FileReader(nombreFichero));

            int c = f.read(); // leemos un carácter
            while (c != -1) { // mientras no lleguemos al final del fichero
                System.out.print((char) c); // mostramos
                c = f.read(); // volvemos a leer
            }
            f.close(); // una vez utilizado el fichero lo cerramos
        } catch (EOFException eof) {
            System.out.println("Fichero no se pudo abrir.");
        }
    }
}
```

10.2. Diseñar una aplicación que pida al usuario su nombre y edad. Estos datos deben guardarse en el fichero *datos.txt*. Si este fichero existe, debe borrarse su contenido, y en caso de que no existir, debe crearse.

```
import java.io.*;
import java.util.Scanner;

/*
 * Para sobrescribir o crear el fichero, si no existe, utilizaremos el constructor
 * adecuado de FileWriter.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Escriba su nombre: ");
        String nombre = sc.nextLine();
        System.out.print("Escriba su edad: ");
        int edad = sc.nextInt();

        try {
            BufferedWriter f = new BufferedWriter(new FileWriter("datos.txt"));

            f.write("Nombre: " + nombre); // escribimos los datos
            f.newLine(); //escribimos en el fichero una nueva línea
            f.write("Edad: " + edad);

            f.close(); // una vez utilizado el fichero lo cerramos
        }
    }
}
```

```
        } catch (EOFException eof) {
            System.out.println("Fichero no se pudo abrir.");
        }
    }
}
```

10.3. Crear un programa que duplique el contenido de un fichero. Realizar dos versiones:

- a) Duplicaremos el fichero *original.txt* en uno que se llame *copia.txt*.
- b) Pedir el nombre del fichero fuente y duplicarlo en un fichero con el mismo nombre con el prefijo «*copia_de_*».

Solución a)

```
import java.io.*;

/*
 * Leeremos, carácter a carácter, del fichero original y escribiremos en el fichero
 * destino. Utilizaremos dos flujos, uno de entrada y otro de salida. */
public class Main {

    public static void main(String[] args) throws Exception {
        try { //abrimos los ficheros (de entrada y de salida)
            BufferedReader f1 = new BufferedReader(new FileReader("original.txt"));
            BufferedWriter f2 = new BufferedWriter(new FileWriter("copia.txt"));

            int c = f1.read(); // leemos del original
            while (c != -1) { // mientras no lleguemos al final del fichero
                f2.write(c); // escribimos en la copia
                c = f1.read(); //y volvemos a leer
            }

            f1.close(); //cerramos los ficheros
            f2.close();
        } catch (EOFException eof) {
            System.out.println("Error de fichero");
        }
    }
}
```

Solución b)

```
import java.io.*;
import java.util.Scanner;

/*
 * El algoritmo es idéntico a la primera versión, salvo que solicitamos al usuario el
 * nombre del fichero y, a partir de él, creamos el nombre del fichero de salida. */
public class Main {

    public static void main(String[] args) throws Exception {
        Scanner sc = new Scanner(System.in);

        System.out.print("Fichero fuente: ");
        String fuente = sc.nextLine(); //nombre del fichero a copiar
        String destino = nombreFichero(fuente); //función que añade el prefijo "copia_de_"

        try {
            BufferedReader f1 = new BufferedReader(new FileReader(fuente));
```



```

        BufferedWriter f2 = new BufferedWriter(new FileWriter(destino));

        int c = f1.read(); // leemos del original
        while (c != -1) { // mientras no terminemos
            f2.write(c); // escribimos en la copia
            c = f1.read();
        }
        f1.close(); //cerramos
        f2.close();
    } catch (EOFException eof) {
        System.out.println("Error de fichero.");
    }
}

// Devuelve el nombre del fichero con el prefijo "copia_de_". Tendremos cuidado con
// el path del fichero. Un ejemplo:
// si el nombre del fichero es "carta.txt", quedará "copia_de_carta.txt". Pero
// si el nombre es "../documentos/carta.txt" el prefijo hay que insertarlo después
// del último '/', quedando "../documentos/copia_de_carta.txt"
static String nombreFichero(String nombre) {
    String nuevoNombre;

    int pos = nombre.lastIndexOf('/'); // buscamos la última ocurrencia de '/'
    if (pos == -1) { // si no contiene ningún '/'
        nuevoNombre = "copia_de_" + nombre;
    } else {
        nuevoNombre = nombre.substring(0, pos + 1) + //parte del path antes del último /
            "copia_de_" + //prefijo
            nombre.substring(pos + 1); //parte del path después del último /
    }

    return (nuevoNombre);
}
}
}

```

- 10.4.** Realizar un programa que lea un fichero de texto llamado *carta.txt*, tenemos que contar los caracteres, las líneas y las palabras. Para facilitar el procesamiento supondremos que cada palabra está separada de otra por un único espacio en blanco.

```

import java.io.*;
/*
 * Recorreremos el fichero carácter a carácter, contando los caracteres, palabras (espacios en blanco y nueva línea) y las líneas (carácter nueva línea). */
public class Main {

    public static void main(String[] args) throws Exception {
        try {
            BufferedReader f = new BufferedReader(new FileReader("carta.txt"));
            int contCar = 0, // contador de caracteres
                contPal = 0, // contador de palabras
                contLinea = 0; // contador de líneas

            int c = f.read(); // leemos un carácter

            while (c != -1) { //mientras no hayamos procesado todo el fichero
                if ((char) c == ' ') { // suponemos solo un espacio entre palabras
                    contCar++; //un carácter más
                    contPal++; //una palabra más
                } else {
                    if ((char) c == '\n') { // encontramos una línea
                        contPal++; //una palabra más
                        contLinea++; //una línea más
                    } else {

```

```
        contCar++; //es otro carácter
    }
    c = f.read(); //volvemos a leer
}
f.close(); // cerramos el fichero, ya que lo hemos procesado entero

if (contPal > 0) { //la ultima palabra no se ha contabilizado
    contPal++;
}
if (contLinea > 0) { //la última línea no se ha contabilizado
    contLinea++;
}

System.out.println("Caracteres: " + (contCar - 1)); //contamos un carácter más
System.out.println("Palabras: " + contPal);
System.out.println("Lineas: " + contLinea);
} catch (EOFException eof) {
    System.out.println("Error");
}
}
}
```

- 10.5. En el archivo *numeros.txt* disponemos de una serie de números (uno por cada línea). Diseñar un programa que procese el fichero y nos muestre el menor y el mayor.

```
import java.io.*;
/*
 * Mientras leemos los números del fichero vamos guardando el menor y el mayor de los
 * que se han leído hasta el momento.
 */
public class Main {

    public static void main(String[] args) throws Exception {
        BufferedReader entri = new BufferedReader(new FileReader("numeros.txt"));
        int num, max, min; //números leídos, máximo y mínimo

        String cifra = entri.readLine(); //leemos el primer número (como un String)
        num = Integer.valueOf(cifra); // convertimos el String a un int
        max = num; // el primer número que leamos será el máximo
        min = num; // y el mínimo

        cifra = entri.readLine(); //volvemos a leer
        while (cifra != null) { //mientras podamos leer cifras
            num = Integer.valueOf(cifra); //convertimos de String a int

            if (num > max) { // si es más grande que el max, será el nuevo máximo
                max = num;
            }
            if (num < min) { // si es menor que el min, será el nuevo mínimo
                min = num;
            }
            cifra = entri.readLine(); //leemos de nuevo
        }

        entri.close(); // cerramos el fichero

        System.out.println("Mayor: " + max); // mostramos los datos
        System.out.println("Menor: " + min);
    }
}
```

- 10.6. Un libro de firmas es útil para recoger todas las personas que han pasado por un determinado lugar. Crear una aplicación que permita mostrar el libro de firmas o insertar un nuevo nombre (comprobando que no se encuentre repetido). Llamaremos al fichero *firmas.txt*.

```
import java.io.*;
import java.util.Scanner;

/* La aplicación tendrá dos opciones: mostrar e insertar (comprobando que no existan
 * nombres repetidos). */
public class Main {

    public static void main(String[] args) throws Exception {
        Scanner sc = new Scanner(System.in);
        System.out.println("1. Mostrar libro de firmas."); //menú de la aplicación
        System.out.println("2. Añadir nombre.");
        System.out.print("¿Qué desea hacer? ");
        int opc = sc.nextInt(); //lee un entero, pero deja en el búfer el \n
        sc.nextLine(); //esta lectura lee el \n y lo limpia del búfer. Esto hay que
        //hacerlo siempre que leamos un número y a continuación una cadena
        switch (opc) {
            case 1:
                muestraFicheroFirmas();
                break;

            case 2:
                System.out.print("Introduzca el nombre que desea insertar: ");
                String nombre = sc.nextLine(); //nombre a insertar en el fichero
                insertaNuevaFirma(nombre); //inserta nombre, si no está repetido
                break;
        }
    }

    //Recorre el fichero, línea a línea, mostrando su contenido por consola
    static void muestraFicheroFirmas() throws IOException {
        try { //abrimos el fichero, leemos línea a línea y mostramos
            BufferedReader entr = new BufferedReader(new FileReader("firmas.txt"));
            String linea = entr.readLine(); //lee una línea y no el \n (leído antes)
            while (linea != null) { //mientras no lleguemos al final del fichero
                System.out.println(linea); //mostramos por consola
                linea = entr.readLine(); //volvemos a leer.
            }

            entr.close(); //cerramos
        } catch (EOFException eof) {
            System.out.println("Error de fichero.");
        } catch (FileNotFoundException fnf) {
            System.out.println("No se encuentra el fichero");
        }
    }

    //Recorre el fichero buscando el nombre pasado como parámetro, en el caso en que no
    //lo encuentre lo inserta al final
    static void insertaNuevaFirma(String nuevo) throws IOException {
        try {
            BufferedReader entr = new BufferedReader(new FileReader("firmas.txt"));
            String nombre = entr.readLine(); //lee el primer nombre del fichero
            boolean encontrado = false; //suponemos que no se ha encontrado

            //mientras quede por leer del fichero y no encontremos el nombre buscado
            while (nombre != null && encontrado == false) {
                if (nombre.equals(nuevo)) {
                    encontrado = true;
                }
            }
        }
    }
}
```

```
        nombre = entr.readLine(); //leemos otro nombre del fichero
    }
    entr.close(); //cerramos

    if (encontrado == false) { //si el nombre nuevo no está en el fichero
        BufferedWriter firma = new BufferedWriter(
            new FileWriter("firmas.txt", true)); //abrimos para añadir
        firma.newLine(); //insertamos una nueva línea en el fichero
        firma.write(nuevo); //e insertamos el nuevo nombre
        System.out.println("\nNuevo nombre insertado.");
        firma.close(); //cerramos
    } else {
        System.out.println("\nYa había firmado");
    }
} catch (EOFException eof) {
    System.out.println("Error de fichero");
} catch (FileNotFoundException fnf) {
    System.out.println("No se encuentra el fichero");
}
}
}
```

10.7. En Linux disponemos del comando *more*, al que se le pasa un fichero y lo muestra poco a poco: cada 24 líneas. Implementar un programa que funcione de forma similar.

```
import java.io.*;
import java.util.Scanner;

/*
 * La idea es mostrar el fichero, línea a línea, utilizando un contador y haciendo una
 * pausa cuando llegamos a un número determinado de líneas mostradas. */
public class Main {

    public static void main(String[] args) throws Exception {
        Scanner sc = new Scanner(System.in);
        final int PARAR_CADA = 24; //indica cada cuántas líneas hacer la pausa
        int cont = 0; //contador de líneas mostradas

        try {
            BufferedReader entr = new BufferedReader(new FileReader("texto.txt"));
            String linea = entr.readLine(); //leemos la primera línea
            while (linea != null) {
                System.out.println(linea);
                cont++; //incrementamos el número de líneas mostradas
                if (cont == PARAR_CADA) { //si hemos mostrado el número de líneas PARAR_CADA
                    System.out.println();
                    System.out.print("Pulse Intro...");
                    sc.nextLine(); //leemos pero no hace falta guardar nada
                    System.out.println();
                    cont = 0; //inicializamos el contador
                }
                linea = entr.readLine(); //volvemos a leer
            }
            entr.close(); //cerramos
        } catch (EOFException eof) {
            System.out.println("Error de fichero.");
        } catch (FileNotFoundException fnf) {
            System.out.println("Fichero no encontrado.");
        }
    }
}
```

- 10.8. Disponemos de dos ficheros, *perso1.txt* y *perso2.txt* con nombres de personas (ambos ordenados). Realizar un programa que lea ambos ficheros y que cree un tercer fichero (*todos.txt*) con todos los nombres ordenados alfabéticamente.

```
import java.io.*;
/*
 * Vamos a implementar un algoritmo de fusión, que ya hemos implementado en alguna
 * ocasión. La diferencia es que aquí los datos proviene de un fichero. */
public class Main {

    public static void main(String[] args) throws Exception {
        try {
            // abrimos perso1.txt y perso2.txt para lectura
            BufferedReader perso1 = new BufferedReader(new FileReader("perso1.txt"));
            BufferedReader perso2 = new BufferedReader(new FileReader("perso2.txt"));
            // abrimos todos para escritura
            BufferedWriter todos = new BufferedWriter(new FileWriter("todos.txt"));

            String nombre1 = perso1.readLine(); //primer nombre del fichero perso1.txt
            String nombre2 = perso2.readLine(); //primer nombre del fichero perso2.txt

            while (nombre1 != null && nombre2 != null) {
                if (nombre1.compareToIgnoreCase(nombre2) < 0) { //si nombre1 < nombre2
                    todos.write(nombre1); // guardamos nombre1 y leemos de perso1.txt
                    todos.newLine();
                    nombre1 = perso1.readLine();
                } else { //en caso contrario
                    todos.write(nombre2); // guardamos nombre2 y leemos de perso2.txt
                    todos.newLine();
                    nombre2 = perso2.readLine();
                }
            }
            // en este punto, hemos terminado de procesar uno de los ficheros y falta por
            // copiar el resto del otro en el fichero de salida
            if (nombre1 == null) { // falta copiar el resto de perso2.txt
                while (nombre2 != null) {
                    todos.write(nombre2);
                    todos.newLine();
                    nombre2 = perso2.readLine();
                }
            } else { // falta copiar el resto de perso1.txt
                while (nombre1 != null) {
                    todos.write(nombre1);
                    todos.newLine();
                    nombre1 = perso1.readLine();
                }
            }
            perso1.close(); // cerramos los ficheros
            perso2.close();
            todos.close();
        } // del try
        catch (EOFException eof) {
            System.out.println("Error de fichero.");
        } catch (FileNotFoundException fnf) {
            System.out.println("Fichero no encontrado.");
        }
    } // del main
} // de la clase Main
```

- 10.9. Un encriptador es una aplicación que transforma un texto haciendolo ilegible para aquellos que desconocen el código. Diseñar un programa que lea un fichero de texto,

lo codifique y cree un nuevo archivo con el mensaje cifrado. El alfabeto de codificación se encontrará en el fichero *codec.txt*. Un ejemplo de alfabeto de codificación es:

Alfabeto	a	b	c	d	e	f	g	h	i	j	k	l	m	n	→
Cifrado	e	m	s	r	c	y	j	n	f	x	i	w	t	a	→

→	o	p	q	r	s	t	u	v	w	x	y	z
→	k	o	z	d	l	q	v	b	h	u	p	g

```
import java.io.*;

/* Este ejercicio se realizó en el capítulo de cadenas, con la diferencia que aquí
 * leemos los datos de un fichero.
 */
public class Main {

    public static void main(String[] args) {
        String cad;
        char codec[][] = null; //tabla bidimensional con el alfabeto
        int c; // la función que lee, devuelve el carácter como un entero; trabajaremos
        // con c como entero y haremos un casting a char cuando nos haga falta

        try { //primero cargamos el alfabeto
            BufferedReader entr = new BufferedReader(new FileReader("codec.txt"));
            codec = new char[2][26]; //tabla de 2x26
            cad = entr.readLine();
            codec[0] = cad.toCharArray(); //letras sin codificar
            cad = entr.readLine();
            codec[1] = cad.toCharArray(); //letras codificadas
            entr.close();
        } catch (IOException eof) {
            System.out.println("Error de fichero");
        }

        // ahora abrimos el texto a codificar y el resultado
        // iremos leyendo carácter a carácter, codificando y escribiendo
        try {
            // abrimos los ficheros
            BufferedReader lect = new BufferedReader(new FileReader("texto.txt"));
            BufferedWriter escr = new BufferedWriter(new FileWriter("codificado.txt"));

            c = lect.read(); // leemos el carácter a codificar
            //read() devuelve el carácter leído como un entero o -1 si se termina el fichero

            while (c != -1) { // mientras no terminemos
                escr.write(traduce((char) c, codec)); //traducimos y escribimos
                c = lect.read(); // leemos el carácter a codificar
            }

            lect.close(); //cerramos
            escr.close();
        } catch (IOException eof) {
            System.out.println("Error de fichero");
        }
    }

    // Devuelve el carácter codificado, según el alfabeto pasado como parámetro
    static char traduce(char c, char codec[][]) {
        char cod;
        int i = 0;

        while (i < codec[0].length && c != codec[0][i]) { //búsqueda secuencial
            i++;
        }
    }
}
```

```

        if (i < codec[0].length) { //si encontramos el elemento buscado
            cod = codec[1][i]; //codificamos
        } else {
            cod = c; //no codificamos el carácter
        }
        return (cod);
    }
}

```

10.10. Utilizando el fichero *codec.txt* del ejercicio anterior, diseñar un decodificador.

```

// La solución de este ejercicio es idéntica a la planteada en el Ejercicio 10.9,
// con la diferencia de que intercambiamos el alfabeto sin codificar con el codificado
// y viceversa.
// El resto del algoritmo es exactamente igual.
// Veamos cómo se intercambian los alfabetos:
...
try {
    BufferedReader entr = new BufferedReader(new FileReader("codec.txt"));
    codec = new char[2][26];
    cad = entr.readLine();
    codec[1] = cad.toCharArray(); // codec[1] alfabeto sin codificar
    cad = entr.readLine();
    codec[0] = cad.toCharArray(); // codec[0] alfabeto codificado
    entr.close();
} catch (IOException eof) {
    ...
}

```

10.11. El fichero *matriz.txt*, contiene los datos de una matriz cuadrada. Leer dicha matriz y mostrarla transpuesta en pantalla.

```

import java.io.*;

public class Main {

    public static void main(String[] args) throws Exception {
        int fila[], m[][] = null; //tabla para almacenar la matriz

        try { // abrimos matriz.txt
            BufferedReader matriz = new BufferedReader(new FileReader("matriz.txt"));
            int filaActual = 0; //número de fila con la que trabajamos

            String linea = matriz.readLine(); // leemos la primera fila (línea)
            while (linea != null && !linea.isEmpty()) {
                System.out.println(linea);
                // línea es una cadena, obtenemos los datos
                fila = extraeFila(linea);
                m = insertaNuevaFila(m, fila);
                filaActual++;
                linea = matriz.readLine();
            }
            matriz.close(); // cerramos el fichero

            muestraTranspuesta(m); ///mostramos la matriz transpuesta
        } // del try
        catch (EOFException eof) {
            System.out.println("Error de fichero.");
        } catch (FileNotFoundException fnf) {
            System.out.println("Fichero no encontrado.");
        }
    } // del main
}

```

```
// Saca la información de una serie de números de una cadena en una tabla.
// Suponemos que los números están separados por un único espacio en blanco.
// Esta función extrae de una cadena del tipo "12 81 2 0 765" la tabla:
// [12, 81, 2, 0, 765]
static int[] extraeFila(String l) {
    int t[];
    String aux[];

    aux = l.split(" "); //separa la cadena "12 34 56" en una tabla ["12", "34", "56"]
    t = new int[aux.length]; // creamos una tabla de enteros de idéntico tamaño a aux

    for (int i = 0; i < aux.length; i++) { // convertimos a entero cada elemento de aux
        t[i] = (int) Integer.parseInt(aux[i]);
    }
    return (t);
}

//redimensiona la tabla m, añadiendo la información de la fila f
static int[][] insertaNuevaFila(int m[][], int f[]) {
    int filas;
    int aux[][] = null;

    // averiguamos el número de filas actual de la matriz
    if (m == null) {
        filas = 0;
    } else {
        filas = m.length;
        aux = new int[m.length][f.length]; // copiamos las filas de m en aux;
        for (int i = 0; i < m.length; i++) {
            aux[i] = m[i];
        }
    }

    filas++; // como añadiremos una fila, incrementamos en 1
    m = new int[filas][f.length];
    if (filas != 1) {
        // restauramos los valores (filas) de aux a m
        for (int i = 0; i < aux.length; i++) {
            m[i] = aux[i];
        }
    }
    m[m.length - 1] = f; // añadimos la nueva fila justo al final de m

    return (m);
}

//muestra la matriz transpuesta de m[][] (pasada como parámetro)
static void muestraTranspuesta(int m[][]) {
    for (int i = 0; i < m.length; i++) {
        for (int j = 0; j < m[0].length; j++) {
            //en lugar de [i][j] utilizamos [j][i]
            System.out.print(m[j][i] + " ");
        }

        System.out.println(); //nueva línea al final de la fila
    }
}
} // de la clase Main
```

10.12. Algunos sistemas operativos disponen del comando *comp*, que compara dos archivos y nos dice si son iguales o distintos. Diseñar este comando de forma que, además, nos diga en qué línea y carácter se encuentra la primera diferencia. Utilizar los ficheros *texto1.txt* y *texto2.txt*.


```
import java.io.*;
/*
 * Iremos leyendo conjuntamente de ambos ficheros, carácter a carácter. Llevaremos
 * la cuenta del número de caracteres leídos en cada línea, y del número de líneas,
 * mediante '\n'. La primera diferencia la encontramos cuando los caracteres
 * leídos de ambos ficheros sean distintos. */
public class Main {

    public static void main(String[] args) throws Exception {
        int contLinea = 1, // inicializamos el contador de línea
            contCar = 1; // inicializamos el contador de caracteres

        try {
            BufferedReader entr1 = new BufferedReader(new FileReader("texto1.txt"));
            BufferedReader entr2 = new BufferedReader(new FileReader("texto2.txt"));

            int c1 = entr1.read(); // leemos el primer carácter de cada fichero
            int c2 = entr2.read();
            while (c1 != -1 && c2 != -1 && c1 == c2) { //mientras sean iguales
                contCar++; //tenemos un carácter más
                if ((char) c1 == '\n') { //si encuentro un \n incremento contLinea
                    contLinea++;
                    contCar = 1; // e inicializamos el carácter a 1
                }
                c1 = entr1.read();
                c2 = entr2.read();
            }

            // Al salir del bucle puede ocurrir:
            if (c1 != c2) { //si son distintos
                System.out.println("Distinto:");
                System.out.println("Línea: " + contLinea);
                System.out.println("Carácter: " + contCar);
            } else {
                System.out.println("Los ficheros son iguales.");
            }

            entr1.close(); // cerramos los ficheros
            entr2.close();
        } catch (EOFException eof) {
            System.out.println("Error de fichero");
        }
    }
}
```

10.13. Diseñar una pequeña agenda, que muestre el siguiente funcionamiento:

1. Nuevo contacto.
2. Buscar por nombre.
3. Mostrar todos.
4. Salir.

En la agenda, guardaremos el nombre y el teléfono de un máximo de 20 personas.

La opción 1 nos permitirá introducir un nuevo contacto siempre y cuando la agenda no esté llena, comprobando que el nombre no se encuentra insertado ya.

La opción 2 muestra todos los teléfonos que coinciden con la cadena a buscar. Por ejemplo, si tecleamos «Pe», mostrará el teléfono de Pedro, de Pepe y de Petunia.

La opción 3 mostrará un listado con toda la información (nombres y teléfonos) ordenados alfabéticamente por el nombre.

Por último, la opción 4 guarda todos los datos de la agenda en el archivo *agenda.txt*.

La próxima vez que se ejecute la agenda, se debe comprobar si hay datos guardados y cargarlos.

Programa principal

```
import java.util.Scanner;

public class Main {

    //Número máximo de contactos en la agenda
    final static int NUM_MAX_CONTACTOS = 20;
    public static void main(String[] args) throws Exception {
        Scanner sc = new Scanner(System.in);
        Agenda age; // agenda para almacenar toda la información
        int opc;     // opción del menú
        String nombre, tlf; //variables auxiliares

        age = new Agenda(NUM_MAX_CONTACTOS); // creamos la agenda de contactos

        age.desdeFichero(); // cargamos la agenda desde el fichero de texto

        // manipulación de la agenda
        do {
            opc = menu();
            switch (opc) {
                case 1:
                    if (age.agendaLlena() == true) { //si no caben nuevos contactos
                        System.out.println("La agenda está llena");
                    } else {
                        // leemos los datos del nuevo contacto
                        System.out.print("Nombre?: ");
                        nombre = sc.nextLine();
                        System.out.print("Teléfono?: ");
                        tlf = sc.nextLine(); //el teléfono se trata como una cadena
                        age.nuevoContacto(nombre, tlf); //insertamos el nuevo contacto
                    }
                    break;

                case 2:
                    System.out.print("Nombre a buscar?: ");
                    nombre = sc.nextLine();
                    age.buscaNombre(nombre); //muestra los contactos coincidentes con "nombre"
                    break;

                case 3:
                    age.muestraTodos(); // mostramos toda la agenda ordenada
                    break;
            }
        } while (opc != 4);

        age.aFichero(); //antes de salir volcamos la agenda en el fichero de texto
    } // de main

    static int menu() {
        Scanner sc = new Scanner(System.in);
        int opc;

        System.out.println("1. Nuevo contacto");
        System.out.println("2. Buscar por nombre");
        System.out.println("3. Mostrar todos");
        System.out.println("4. Salir");
    }
}
```

```

        do {
            System.out.print("opción?: ");
            opc = sc.nextInt();
        } while (opc <= 0 || opc >= 5);

        return (opc);
    }
} // de la clase Main

```

Clase Agenda

```

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

/*
 * La clase Agenda incluire una serie de contactos. Al crear un objeto de tipo Agenda,
 * tenemos que indicar cuántos contactos podrá almacenar como máximo. */
public class Agenda {
    private Contacto agenda[]; // tabla para guardar los contactos
    private int numContactos; //indicador del número de contactos en la agenda

    Agenda(int tam) {
        agenda = new Contacto[tam]; // creamos la tabla del tamaño indicado
        numContactos = 0; //inicialmente no tenemos ningún contacto en la agenda
    }

    // si es posible, inserta un nuevo contacto en la agenda
    public void nuevoContacto(String nombre, String tlf) {
        if (numContactos < agenda.length) {
            agenda[numContactos] = new Contacto(nombre, tlf);
            numContactos++;
        }
    }

    // indica si la agenda está llena o por el contrario tienen cabida más contactos
    public boolean agendaLlena() {
        return (numContactos == agenda.length);
    }

    // busca y muestra todos los contactos cuyo nombre comienza o coincide con la
    // cadena pasado como parámetro
    public void buscaNombre(String nombre) {
        Contacto claveBusqueda = new Contacto(nombre, ""); //contacto para buscar
        for (int i = 0; i < numContactos; i++) {
            if (agenda[i].equals(claveBusqueda)) {
                System.out.println(agenda[i]);
            }
        }
    }

    // ordena la agenda por el nombre
    public void muestraTodos() {
        Arrays.sort(agenda, 0, numContactos); //ordena la agenda (0..numContactos-1)

        for (int i = 0; i < numContactos; i++) { //muestra los contactos hasta numContacto
            System.out.println(agenda[i]);
        }
    }

    // lee un fichero de texto con el formato "nombre:teléfono" en cada línea, y
    // carga la información en la agenda
    public void desdeFichero() throws Exception {
        try {
            // si existe el fichero cargaremos los datos almacenados, y si no
            // encontramos nada significa que la agenda está vacía

```

```
BufferedReader f = new BufferedReader(new FileReader("agenda.txt"));

String linea = f.readLine();
while (linea != null && !agendaLlena()) {
    // la línea tiene el formato: "nombre;teléfono"
    // separamos y guardamos
    nuevoContacto(linea.split(";")[0], linea.split(";")[1]);
    linea = f.readLine();
}
// salimos del while por dos motivos:
// - hemos leído todo el fichero agenda.txt
// - o la agenda está llena, y el fichero no cabe en las tablas
if (agendaLlena() && linea != null) {
    System.out.println("Se ignoran algunos contactos del fichero");
    System.out.println("agenda.txt demasiado grande");
}

f.close(); // ya hemos procesado el fichero
} catch (EOFException eof) {
    System.out.println("Error de fichero.");
} catch (FileNotFoundException fnf) {
    System.out.println("Agenda vacia.");
}
}

// Vuelca el contenido de la agenda en un fichero de texto. El formato para
// guardar la información en el fichero sera: "nombre;teléfono"
void aFichero() throws Exception {
    try {
        BufferedWriter f = new BufferedWriter(new FileWriter("agenda.txt"));

        for (int i = 0; i < numContactos; i++) {
            f.write(agenda[i].formatoFichero());
            f.newLine();
        }

        f.close(); // cerramos el fichero
    } // del try
    catch (EOFException eof) {
        System.out.println("Error de fichero.");
    } catch (FileNotFoundException fnf) {
        System.out.println("Fichero no encontrado.");
    }
}
}
```

Clase Contacto

```
/*
 * La clase Contacto almacenará información de una persona: nombre y teléfono.
 * Para impedir que una vez creado se manipulen los datos, pondremos todos los
 * atributos como final. */
public class Contacto implements Comparable<Contacto> {
    private final String nombre; // nombre del contacto
    private final String tlf;    // y teléfono

    Contacto(String nombre, String tlf) { // constructor
        this.nombre = nombre;
        this.tlf = tlf;
    }

    public String formatoFichero() {
        return (nombre + ";" + tlf); // devuelve una cadena con el formato:
        // "nombre;teléfono" para guardar el contacto en el fichero de texto
    }
}
```

```
// Dos contactos son iguales si sus nombres son iguales, o this.nombre comienza por
// el nombre del contacto pasado.
@Override
public boolean equals(Object otro) {
    Contacto otroContacto = (Contacto) otro; //hacemos un cast de Object a Contacto
    return this.nombre.equals(otroContacto.nombre) ||
        this.nombre.startsWith(otroContacto.nombre);
}

// devuelve una representación del contacto
@Override
public String toString() {
    String aux;
    aux = "Nombre: " + nombre + "\tTeléfono: " + tlf;
    return (aux);
}

//Implementamos el método compareTo de Comparable, para poder comparar dos contactos.
// Devolveremos un entero <0, =0 ó >0 según la comparación alfabética del nombre.
@Override
public int compareTo(Contacto otro) {
    return this.nombre.compareTo(otro.nombre);
}
}
```

Ejercicios propuestos

- 10.1. Repetir el Ejercicio resuelto 10.4, pero sabiendo que una palabra no está separada de otra por único espacio en blanco. Como separadores podemos utilizar: espacio en blanco, tabuladores, punto, coma, punto y coma y paréntesis.
- 10.2. Diseñar un programa al que se le proporcione el nombre de un fichero de texto y una cadena. Debemos buscar todas las ocurrencias de la cadena en el fichero, indicando la fila y la columna donde aparece.
- 10.3. Escribir un programa que pida el nombre de un fichero de texto que contenga código fuente en Java. El programa debe crear un nuevo fichero que tenga como nombre el nombre del fichero original con el prefijo «sin_comentarios_». El nuevo fichero tendrá como contenido el código fuente sin ningún tipo de comentarios.
- 10.4. En ocasiones, es necesario particionar un fichero de texto de gran tamaño en otros ficheros más pequeños. Se pide crear una aplicación a la que se le proporciona un fichero de texto, y un tamaño. Este puede estar en Bytes, kiloBytes o MegaBytes. La aplicación debe fragmentar el fichero original en tantos ficheros como necesite. Los nombres de estos ficheros serán idénticos al nombre original con el prefijo «parte999_», donde 999 es el número del volumen.

Para indicar el tamaño se escribirá una cantidad seguida de b (Bytes), k (kBytes) o m (MBytes).

- 10.5. Escribir un programa en el que, partiendo de los ficheros generados en el ejercicio anterior restablezca el fichero original.

Capítulo 11

Ficheros binarios

En el Capítulo 10 vimos que hay dos tipos de flujos de datos en Java, los de tipo binario (también llamados *bytes*) y los de texto. Allí nos dedicamos a estudiar con detalle los de texto. Ahora nos ocuparemos de los de tipo *byte*, que nos van a permitir guardar (o transferir) y recuperar (o recibir) cualquier tipo de datos usados en un programa. No olvidemos que para usar cualquier flujo en Java, debemos importar las clases del paquete `java.io`,

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

Cuando se trata de escribir (o leer) bytes en un fichero, existen dos clases básicas `FileOutputStream` y `FileInputStream`. Pero nosotros no solemos manejar bytes individuales en nuestros programas, sino datos (eso sí, formados por bytes) más complejos, ya sean de tipos primitivos, tablas u objetos. Por eso necesitamos un intermediario capaz de convertir los datos en series planas de bytes o reconstruir los datos a partir de series de bytes. Estos dos procesos se llaman *serialización* y *deserialización de datos*, respectivamente. Esos intermediarios son flujos llamados *envoltorio*, `ObjectOutputStream` y `ObjectInputStream`, que se crean a partir de flujos de bytes planos, como `FileOutputStream` y `FileInputStream`.

11.1. Flujos de salida

Por ejemplo, si queremos utilizar un flujo de salida de datos para grabar en disco una tabla de enteros, crearemos primero un flujo de salida de tipo binario, asociado a un fichero que llamaremos «enteros.dat»,

```
FileOutputStream archivo = new FileOutputStream("enteros.dat");
```

Como ocurría con los archivos de texto, el nombre del archivo puede incluir una ruta de acceso, con los requisitos que vimos en el Capítulo 10. Una vez creado este flujo, lo «envolvemos» en un objeto de la clase `ObjectOutputStream`,

```
ObjectOutputStream out = new ObjectOutputStream(archivo);
```

El constructor de `ObjectOutputStream` puede arrojar una excepción `IOException`¹. Por tanto, debe ir encerrado en una estructura `try-catch`.

La clase `ObjectOutputStream` tiene una serie de métodos que permiten la escritura de datos complejos de cualquier tipo o clase. Eso sí, para escribir un objeto, su clase debe tener implementada la interfaz `Serializable`, que no es más que una marca que declara al objeto en cuestión como susceptible de ser serializado, es decir, convertible en una serie plana de bytes. Las clases implementadas por Java, como `String`, las `Collections` y las tablas, traen implementadas la interfaz `Serializable` y no hay que preocuparse de ellas. Lo mismo puede decirse de los datos de tipo primitivo. En cambio, las clases definidas por el usuario deben declararse como serializables en su definición, sin que esto nos obligue a implementar ningún método especial,

```
class miClase implements Serializable {  
    ... //cuerpo de la clase  
}
```

Con esto, `miClase` ya es serializable, y sus objetos susceptibles de ser enviados por un flujo binario.

`ObjectOutputStream` dispone de los siguientes métodos para la escritura de datos en un flujo de salida:

`void writeBoolean(boolean b)`: escribe un valor `boolean` en el flujo.

`void writeChar(int c)`: escribe el valor `char` que ocupa los dos bytes menos significativos del valor entero que se le pasa.

`void writeInt(int n)`: escribe un entero.

`void writeLong(long n)`: escribe un entero largo.

`void writeDouble(double d)`: escribe un número de tipo `double`.

`void writeObject(Object o)`: escribe un objeto serializable.

Como ejemplo, vamos a guardar en un archivo una tabla de números enteros. Para ello, inicializamos la tabla con los enteros del 0 al 9, y luego creamos un archivo «datos.dat» y le asociamos un flujo de salida de la clase `ObjectOutputStream`. A continuación, recorreremos la tabla escribiendo los enteros en él,

```
int[] t = new int[10];  
  
try {  
    ObjectOutputStream flujoSalida;  
  
    flujoSalida = new ObjectOutputStream(new FileOutputStream("datos.dat"));  
    for (int i = 0; i < 10; i++) {  
        t[i] = i; //inicializamos la tabla  
    }  
}
```

¹Excepción de entrada/salida.


```

for (int i = 0; i < 10; i++) {
    flujoSalida.writeInt(t[i]); //escribe en el flujo t[i]
}
flujoSalida.close(); //cerramos el flujo para que se vacíe el
//búfer y se escriban los datos pendientes en el archivo
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}

```

En este ejemplo, hemos recorrido la tabla para obtener sus elementos y grabarlos por separado, pero una tabla es un objeto en Java, y podríamos haberla escrito de una vez como objeto, por medio del método `writeObject()`. Por tanto, el segundo bucle `for` puede ser sustituido por una sentencia única,

```
flujoSalida.writeObject(t);
```

donde le hemos pasado como parámetro la referencia al objeto que queremos grabar, la tabla `t`. En este caso hemos grabado la tabla como objeto, que no es lo mismo que grabar los enteros por separado. Esta distinción será importante a la hora de recuperarla.

Igualmente, para guardar una cadena de caracteres se usa la función `writeObject`, ya que una cadena es un objeto de la clase `String`,

```

String cadena = "Sancho Panza";
flujoSalida.writeObject(cadena);

```

11.2. Flujos de entrada

Para leer de fuentes de datos, tales como los archivos, usaremos flujos de la clase `ObjectInputStream`. Por ejemplo, si leemos los datos escritos en el archivo «datos.dat» de la sección anterior, crearemos un flujo de entrada asociado al archivo,

```

ObjectInputStream flujoEntrada = new ObjectInputStream(
    new FileInputStream("datos.dat"));

```

Esta sentencia puede producir una excepción `IOException`; por tanto, deberá ir encerrada en una estructura `try-catch`. Algo exactamente igual ocurre con la sentencia `flujoEntrada.close()` utilizada en el fragmento de código más abajo.

A continuación usaremos los métodos de la clase `ObjectInputStream`, que permiten leer los mismos datos que grabamos con `ObjectOutputStream`. Por cada método de esta última hay otro correspondiente de la primera. En el caso de que hayamos grabado los 10 enteros de una tabla por separado usando `writeInt()`, los podemos recuperar, también por separado, con el método `readInt()`, que puede arrojar `IOException` si hay un error de lectura o `EOFException` si se ha llegado al final del fichero,

```

int[] tabla = new int[10];
for(int i = 0; i < 10; i++) {
    tabla[i] = flujoEntrada.readInt(); //leemos los enteros y los
    //guardamos en la tabla
}

```

En cambio, las cadenas de texto son objetos, y se deben recuperar utilizando el método `readObject()`,

```
try {
    String cadena = (String) flujoEntrada.readObject(); //leemos la cadena
} catch (ClassNotFoundException ex) {
    System.out.println(ex.getMessage());
}
```

Los métodos más importantes de `ObjectInputStream`, son los siguientes:

boolean readBoolean(): lee un booleano del flujo de entrada.

char readChar(): lee un carácter.

int readInt(): lee un entero.

long readLong(): lee un entero largo.

double readDouble(): lee un número real `double`.

final Object readObject(): lee un objeto.

En el ejemplo anterior, llama la atención la estructura `try-catch` asociada a la excepción `ClassNotFoundException`, que rodea al método `readObject()`. Esto se debe a que, cuando leemos un objeto de un flujo de entrada, puede ocurrir que intentemos asignarlo a una variable de distinta clase, en cuyo caso se arrojaría la excepción mencionada. Por esta misma razón, el `cast` aplicado delante de la sentencia de lectura, en este caso `String`, es necesario, ya que el método `readObject()` devuelve un `Object` y nosotros lo asignamos a una referencia de tipo `String`, lo cual supone una conversión de estrechamiento. Esto ocurre cuando a una variable de una determinada clase —en este caso, `String`— se le intenta asignar un objeto de una superclase —en este caso, `Object`—.

Dado que las tablas son objetos, si se ha guardado la tabla `t` usando el método `writeObject()`, el bucle `for` usado en la lectura debe ser sustituido por una sentencia única de lectura, ya que lo que hay guardado es un objeto, no una serie de enteros,

```
try {
    tabla = (int[]) in.readObject();
} catch (ClassNotFoundException ex) {
    System.out.println(ex.getMessage());
}
```

El `cast`, en este caso, es `(int[])`, ya que el objeto que se va a leer (y asignar) es del tipo tabla de enteros.

A menudo desconocemos el número de datos guardados en un archivo. En este caso, para recuperarlos todos no podemos usar un bucle `for` controlado por contador, sino que tenemos que leer hasta que se llegue al final del fichero, es decir, hasta que salte la excepción `EOFException`. Por ejemplo, si un fichero contiene una lista de enteros y no sabemos cuántos hay, para recuperarlos y mostrarlos todos por pantalla usamos un bucle infinito del que solo nos puede sacar la excepción `EOFException` de fin de fichero,

```

try {
    while (true) {
        System.out.println(in.readInt());
    }
} catch (EOFException ex) {
    System.out.println("Fin de fichero");
}

```

Cuando se haya leído el último entero, se habrá llegado al final del fichero. Entonces, se arrojará la excepción y el programa saldrá del bucle `while` y del bloque `try` para continuar en el bloque `catch`.

11.3. Cierre de flujos

Cuando dejamos de necesitar un flujo, ya sea de entrada o de salida, siempre debemos *cerrarlo* para provocar el vaciado de los búferes asociados y liberar el recurso. Esto se consigue con el método `close()`, disponible en todas las clases de entrada y salida que hemos estudiado. Se recomienda incluir este método en un bloque `finally` para asegurarnos de que se ejecuta independientemente de si han saltado excepciones o si todo ha ido correctamente. Un programa tipo que trabaje con flujos, tendría una estructura de la forma,

```

ObjectOutputStream out = null; //null es necesario para la
                               //comparación de finally

try {
    out = new ObjectOutputStream(new FileOutputStream("archivo.dat"));
    ... //sentencias que manipulan el flujo
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(out != null) { //si el flujo está abierto
            out.close();
        }
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}

```

Por otra parte, se puede prescindir de todas estas precauciones, ya que es posible crear el flujo asociándolo, entre paréntesis, al bloque `try`, dejando al programa encargado del cierre incondicional al final de la ejecución de la estructura. Se dice que estamos usando una estructura `try` con recursos (dichos recursos están sujetos al cierre automático). Utilizando el ejemplo anterior quedaría,

```

try (ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("archivo.dat"))) {
    ... //sentencias que manipulan el flujo
} catch (IOException e) {
    System.out.println(e.getMessage());
}

```

Como se puede observar, la declaración y construcción del flujo se coloca entre paréntesis, tras la palabra clave `try` y se prescinde tanto del método `close()`, como del bloque `finally`.

Ejercicios de ficheros binarios

11.1. Pedir un `double` por consola y guardarlo en un archivo binario.

```
import java.io.*;
import java.util.Scanner;
/*
 * Crearemos un flujo FileOutputStream de entrada de bytes planos y, a partir
 * de él, un flujo de objetos envoltorio ObjectOutputStream. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ObjectOutputStream out = null;

        try { // la creación de ObjectOutputStream puede arrojar una excepción
            // IOException
            out = new ObjectOutputStream(new FileOutputStream("datos.dat"));

            System.out.println("Introducir un número real: ");
            double x = sc.nextDouble();

            out.writeDouble(x); // también puede arrojar IOException
        } catch (IOException e) {
            System.out.println(e.getMessage()); // el mensaje describe la excepción,
            // que puede proceder tanto de la apertura o cierre del fichero,
            // como de la escritura del número
        } finally { // debemos asegurarnos del cierre del flujo
            try {
                if (out != null) {
                    out.close(); // hay que cerrar el flujo para que se escriban
                    // los datos pendientes que queden en el búfer
                }
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

11.2. Abrir el archivo del ejercicio anterior, leer el `double` y mostrarlo por pantalla.

```
import java.io.*;
/*
 * Crearemos un flujo FileInputStream de salida de bytes planos y, a partir de
 * él, un flujo de objetos envoltorio ObjectInputStream. */
public class Main {

    public static void main(String[] args) {
        ObjectInputStream in = null;

        try { // la creación de ObjectInputStream puede arrojar una
            // excepción IOException
            in = new ObjectInputStream(new FileInputStream("datos.dat"));
            double x = in.readDouble(); // también puede arrojar IOException
            System.out.println("x: " + x);
        }
    }
}
```

```

    } catch (IOException e) {
        System.out.println(e.getMessage()); // el mensaje describe
        // la excepción, que puede proceder tanto de la apertura
        // o cierre del fichero, como de la lectura del número

    } finally { // debemos asegurarnos del cierre del flujo
        try {
            if (in != null) {
                in.close(); // hay que cerrar el flujo para que se lean
                // los datos pendientes que queden en el búfer
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
}
}

```

11.3. Pedir números enteros positivos por consola, y guardarlos en un fichero binario hasta que se introduzca un número negativo. Leer del fichero todos los enteros guardados y mostrarlos por pantalla.

```

import java.util.Scanner;
import java.io.*;
/* Como no sabemos de antemano cuántos números vamos a guardar, llevaremos un
* contador, que usaremos a la hora de leer del archivo. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ObjectOutputStream out = null;
        ObjectInputStream in = null;
        int contador = 0; // para llevar constancia de cuántos números
        // hemos guardado en el fichero

        try {
            out = new ObjectOutputStream(new FileOutputStream("datos.dat"));
            System.out.println("Introducir un número entero: ");
            int x = sc.nextInt();
            while (x >= 0) {
                contador++;
                out.writeInt(x);
                System.out.println("Introducir un número entero: ");
                x = sc.nextInt();
            }

        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (out != null) {
                    out.close();
                }
            } catch (IOException e) {
            }
        }

        // abrimos de nuevo un flujo para la lectura de los números
        try {
            in = new ObjectInputStream(new FileInputStream("datos.dat"));
            for (int i = 0; i < contador; i++) {
                int x = in.readInt();
                System.out.println(x);
            }
        }
    }
}

```

```
    } catch (IOException e) {
        System.out.println(e.getMessage());
    } finally {
        try {
            if (in != null) {
                in.close();
            }
        } catch (IOException e) {
            //no hacemos nada
        }
    }
}
}
```

11.4. Pedir un entero n por consola. A continuación, pedir n números `double`, que iremos guardando en una tabla. Guardar la tabla en un archivo binario.

```
import java.io.*;
import java.util.Locale;
import java.util.Scanner;
/*
 * Para guardar una serie de números, no es necesario hacerlo de uno en uno, ni llevar el
 * control de cuántos se han guardado. Basta usar una tabla y guardarla como un objeto.*/
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        sc.useLocale(Locale.US); //parautilizar punto (.) en los decimales
        ObjectOutputStream out = null;

        try {
            System.out.println("Número de elementos: ");
            int n = sc.nextInt(); //cantidad de números a leer

            double tabla[] = new double[n]; //tabla con el tamaño adecuado
            out = new ObjectOutputStream(new FileOutputStream("datos.dat"));

            for (int i = 0; i < tabla.length; i++) {
                System.out.println("Introduzca un número real: ");
                tabla[i] = sc.nextDouble();
            }
            out.writeObject(tabla); // las tablas son objetos y se pueden guardar con
            // una sola sentencia, sin necesidad de guardar sus elementos por separado

        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (out != null) { //si hemos abierto el fichero
                    out.close(); //lo cerramos
                }
            } catch (IOException e) {
            }
        }
    }
}
```

11.5. Leer de un fichero binario una tabla de números `double`. Mostrar el contenido de la tabla por consola.

```

import java.io.*;
import java.util.Arrays;
/*
 * La tabla, al ser un objeto, se lee de un fichero de una vez (con una sola instrucción)
 * no es necesario leer sus elementos uno a uno. */
public class Main {

    public static void main(String[] args) throws ClassNotFoundException {
        ObjectInputStream in = null;

        try {
            in = new ObjectInputStream(new FileInputStream("datos.dat"));
            double tabla[] = (double[]) in.readObject(); // El cast indica al compilador
            // la clase de objeto que va a ser leído del fichero. Si en el fichero no se
            // encuentra un objeto de esa clase, se arroja una excepción
            // ClassNotFoundException, que nosotros no vamos a manipular, sino que lo
            // pasamos al sistema de tiempo de ejecución por medio de throws en el
            // encabezamiento del método main
            System.out.println(Arrays.toString(tabla)); //mostramos la tabla
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (in != null) {
                    in.close();
                }

                } catch (IOException e) {
                    System.out.println(e);
                }
            }
        }
    }
}

```

11.6. En un fichero binario, sabemos que se ha guardado una serie de números `double`, pero no sabemos cuántos. Implementar un programa que los lea todos y los muestre por pantalla.

```

import java.io.*;
/*
 * Si no sabemos de antemano cuántos elementos hay guardados en un fichero, podemos
 * valernos de la excepción EOFException, que se arroja al llegar al final del fichero,
 * para detectarlo y parar el proceso de lectura. */
public class Main {

    public static void main(String[] args) {
        ObjectInputStream in = null;

        try {
            in = new ObjectInputStream(new FileInputStream("datos.dat"));

            try { // un bucle infinito lee números indefinidamente hasta que,
                // al llegar al fin del fichero, salta una excepción que nos
                // saca del bucle y del bloque try
                while (true) {
                    System.out.println(in.readDouble());
                }
            } catch (EOFException e) {
                // hemos salido del bucle infinito, al llegar al final del fichero.
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

```
    } finally {  
        try {  
            if (in != null) {  
                in.close();  
            }  
        } catch (IOException e) {  
        }  
    }  
}  
}
```

11.7. Escribir por teclado una frase y guardarla en un archivo binario. A continuación, recuperarla del archivo y mostrarla por pantalla.

```
import java.io.*;  
import java.util.Scanner;  
/*  
 * Una cadena de caracteres, es un objeto de tipo String. */  
public class Main {  
  
    public static void main(String[] args) throws ClassNotFoundException {  
        Scanner sc = new Scanner(System.in);  
        ObjectOutputStream out = null;  
        ObjectInputStream in = null;  
  
        try {  
            System.out.println("Escriba una frase: ");  
            String frase = sc.nextLine();  
            out = new ObjectOutputStream(new FileOutputStream("datos.dat"));  
            out.writeObject(frase);  
        } catch (IOException e) {  
            System.out.println(e.getMessage());  
        } finally {  
            try {  
                if (out != null) {  
                    out.close();  
                }  
            } catch (IOException e) {  
            }  
        }  
  
        // Abrimos un flujo de entrada asociado al mismo archivo:  
        try {  
            in = new ObjectInputStream(new FileInputStream("datos.dat"));  
            //mostramos la cadena leída del flujo:  
            System.out.println((String) in.readObject());  
  
        } catch (IOException e) {  
            System.out.println(e.getMessage());  
        } finally {  
            try {  
                if (in != null) {  
                    in.close();  
                }  
            } catch (IOException e) {  
            }  
        }  
    }  
}
```


- 11.8. Escribir un texto, línea a línea, de forma que cada vez que se pulse **Intro**, se guarde la línea en un archivo binario. El proceso se termina cuando introduzcamos una línea vacía. Leer el texto completo del archivo y mostrarlo por pantalla.

```
import java.io.*;
import java.util.Scanner;
/*
 * Las líneas, incluyendo espacios y signos de puntuación son cadenas de caracteres, es
 * decir, objetos (de String) que podemos ir guardando en un fichero. */
public class Main {

    public static void main(String[] args) throws ClassNotFoundException {
        Scanner sc = new Scanner(System.in);
        ObjectOutputStream out = null;
        ObjectInputStream in = null;

        try {
            System.out.print("Escriba una línea: ");
            String texto = sc.nextLine();
            out = new ObjectOutputStream(new FileOutputStream("datos.dat"));
            while (!texto.isEmpty()) { //leemos líneas hasta que se pulse solo Intro
                out.writeObject(texto);
                System.out.print("Escriba una línea: ");
                texto = sc.nextLine();
            }

        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (out != null) {
                    out.close();
                }
            } catch (IOException e) {
            }
        }

        // Abrimos un flujo de entrada asociado al mismo archivo:
        try {
            in = new ObjectInputStream(new FileInputStream("datos.dat"));
            //mostramos la cadena leída del flujo:
            try {
                while (true) {
                    String frase = (String) in.readObject();
                    System.out.println(frase);
                }

            } catch (EOFException e) {
                // no mostramos nada para que no se añada a nuestro texto
            }

        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (in != null) {
                    in.close();
                }
            } catch (IOException e) {
            }
        }
    }
}
```

- 11.9.** Crear una tabla de 10 números enteros aleatorios menores que 100, ordenados de menor a mayor y guardarlos en un fichero binario. A continuación, recuperarlos y mostrarlos por consola.

```
import java.io.*;
import java.util.Arrays;

public class Main {

    public static void main(String[] args) {
        ObjectInputStream in = null;
        ObjectOutputStream out = null;
        int tabla[] = new int[10]; //tabla de enteros

        try {
            for (int i = 0; i < tabla.length; i++) {
                tabla[i] = (int) (Math.random() * 100); //generamos números enteros
                // aleatorios comprendidos entre 0 y 99 y los guardamos en una tabla
            }
            out = new ObjectOutputStream(new FileOutputStream("numerosAleatorios.dat"));
            Arrays.sort(tabla); // ordenamos los números
            out.writeObject(tabla); //guardamos la tabla en el fichero
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            if (out != null) { //si hemos llegado a abrir el fichero
                try {
                    out.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }

        try { // abrimos el flujo de entrada, para leer los números
            in = new ObjectInputStream(new FileInputStream("numerosAleatorios.dat"));
            try {
                tabla = (int[]) in.readObject(); //leemos un Object que se convierte en int[]
            } catch (ClassNotFoundException ex) {
                System.out.println(ex.getMessage());
            }
            System.out.println(Arrays.toString(tabla)); //mostramos la tabla
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            if (in != null) {
                try {
                    in.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

- 11.10.** Por motivos puramente estadísticos se desea llevar constancia del número de llamadas recibidas en una oficina. Para ello, al terminar cada jornada laboral se guarda dicho número al final de un archivo binario. Implementar una aplicación con un menú, que nos permita añadir el número correspondiente cada día y ver la lista completa en cualquier momento.

```

import java.io.*;
import java.util.Arrays;
import java.util.Scanner;
/*
 * La estrategia será guardar los números en una tabla y grabarla en el fichero
 * como un objeto. Cada vez que se añada un número, se redimensiona la tabla.
 */
public class Main {

    public static void main(String[] args) throws ClassNotFoundException {
        Scanner sc = new Scanner(System.in);

        ObjectOutputStream out = null;
        ObjectInputStream in = null;
        int[] llamadas = new int[0]; // le damos dimensión inicial 0 para que
        // no dé error al intentar evaluar su longitud cuando está vacía

        try {
            in = new ObjectInputStream(new FileInputStream("llamadas.dat"));
            llamadas = (int[]) in.readObject(); // leemos como Object y hacemos el cast
        } catch (IOException e) {
            System.out.println("No hay datos previos");
        } finally {
            if (in != null) {
                try {
                    in.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }

        int opcion;
        do {
            System.out.println("1. Mostrar llamadas");
            System.out.println("2. Añadir llamadas");
            System.out.println("3. Salir");
            System.out.println("");
            System.out.print("Escriba opción: ");

            opcion = sc.nextInt();
            switch (opcion) {
                case 1: // recorremos la tabla para mostrar las llamadas. Si no hay
                    // datos previos, llamadas.length es 0 y no se muestra nada
                    System.out.println("LLlamadas:");
                    System.out.println(Arrays.toString(llamadas));
                    break;

                case 2: // aumentamos la capacidad de la tabla guardando
                    // provisionalmente los datos en una tabla auxiliar
                    int aux[] = Arrays.copyOf(llamadas, llamadas.length+1); // copia elementos
                    System.out.print("Introduzca llamadas del día: ");
                    aux[llamadas.length] = sc.nextInt();
                    llamadas = aux; // llamadas referencia la nueva tabla
            }
        } while (opcion != 3);

        try { // guardamos en el fichero la tabla de llamadas actualizadas
            out = new ObjectOutputStream(new FileOutputStream("llamadas.dat"));
            out.writeObject(llamadas); // escribimos en el fichero
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

```
    } finally {
        if (out != null) {
            try {
                out.close();
            } catch (IOException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }
}
```

11.11. Disponemos de dos ficheros binarios que guardan números enteros ordenados de forma creciente (*numeros1.dat* y *numeros2.dat*). Fusionar ambos ficheros en un tercero (*numeros.dat*), de forma que todos los datos sigan ordenados. Para probar el algoritmo se pueden utilizar los ficheros generados por el Ejercicio resuelto 11.3, introduciendo números ordenados.

```
import java.io.*;
/*
 * Supondremos que los datos de los dos ficheros se guardaron usando un flujo
 * ObjectOutputStream. De no ser así, no podríamos usar ObjectInputStream para leerlos.
 * La estrategia será:
 * 1. Leer el primer número de ambos ficheros
 * 2. Copiar el menor en el fichero de salida y leer del fichero de donde provenía
 * 3. El punto anterior se repite hasta que uno de los dos ficheros se termina
 * 4. El otro fichero, que aún tiene datos, se vuelca a la salida.
 */
public class Main {

    public static void main(String[] args) {
        ObjectInputStream in1 = null, in2 = null; //fichero de entrada (con enteros)
        ObjectOutputStream out = null; //fichero donde fusionamos los datos
        int ultElem1; //último elemento leído del fichero "numeros1.dat"
        int ultElem2; //último elemento leído del fichero "numeros2.dat"

        try {
            // Abrimos los ficheros y leemos un número de cada uno, pararemos cuando se
            //termine de leer alguno de los dos ficheros
            in1 = new ObjectInputStream(new FileInputStream("numeros1.dat"));
            in2 = new ObjectInputStream(new FileInputStream("numeros2.dat"));
            out = new ObjectOutputStream(new FileOutputStream("numeros.dat"));

            try {
                ultElem1 = in1.readInt(); //leemos el primer número de cada fichero
                ultElem2 = in2.readInt();

                while (true) {
                    if (ultElem1 < ultElem2) {
                        out.writeInt(ultElem1); //escribimos el menor: ultElem1
                        System.out.println(ultElem1); //muestra para ver el resultado final
                        ultElem1 = in1.readInt(); //y leemos de numeros1.dat
                    } else {
                        out.writeInt(ultElem2); //escribimos el menor: ultElem2
                        System.out.println(ultElem2);
                        ultElem2 = in2.readInt(); //y leemos del segundo fichero
                    }
                }
            } catch (EOFException e) {
                //salimos del bucle infinito, al llegar al final (leer completamente) alguno
                //de los dos ficheros. Queda copiar el resto del otro fichero a la salida.
                //Naturalmente, solo uno de los dos bucles se va a ejecutar:
            }
        }
    }
}
```

```

        while (in1.available() > 0) {//si quedan por leer datos de in1
            ultElem1 = in1.readInt(); //leemos
            out.writeInt(ultElem1); //y volcamos en la salida
            System.out.println(ultElem1); //mostramos por consola
        }
        while (in2.available() > 0) {//si quedan por leer datos de in2
            ultElem2 = in2.readInt(); //leemos
            out.writeInt(ultElem2); //y volcamos en la salida
            System.out.println(ultElem2); //mostramos por consola
        }
    }

} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    if (in1 != null) { //cerramos in1
        try {
            in1.close();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
    if (in2 != null) { //cerramos in2
        try {
            in2.close();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
    if (out != null) { //cerramos fichero de salida
        try {
            out.close();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
}
}
}

```

11.12. En un comercio desean mantener los datos de sus clientes. Implementar una aplicación que permita guardar y recuperar los datos de los clientes. Para ello, definir la clase **Ciente**, que tendrá los siguientes atributos:

id: identificador de cliente (entero).

nombre: nombre y apellidos del cliente.

telefono: número de teléfono del clientes.

Para realizar las distintas operaciones, la aplicación tendrá el siguiente menú:

1. Añadir nuevo cliente
2. Modificar datos
3. Dar de baja cliente
4. Listar los clientes

La información se guardará en un fichero binario, que se cargará en memoria al iniciar la aplicación y se grabará en disco, actualizada, al salir de la aplicación.

Veamos un ejemplo de posibles valores,

id cliente	Nombre	Teléfono
4	Pepe Pérez	12345
2	José Sánchez	54321
5	Ana Gómez	11223

Clase Cliente

```
import java.io.Serializable;

public class Cliente implements Serializable {
    public final int id; //una vez asignado no permitimos que se modifique
    public String nombre;
    public String tlf;

    Cliente(int id, String nombre, String tlf) {
        this.id = id;
        this.nombre = nombre;
        this.tlf = tlf;
    }

    @Override
    public String toString() {
        return "(" + id + ") Nombre: " + nombre + "\tTeléfono: " + tlf + "\n";
    }
}
```

Clase GestionClientes

```
import java.io.*;
import java.util.Arrays;
import java.util.Scanner;

/* Esta clase dispone de los métodos necesarios para gestionar a los clientes. */
public class GestionClientes {
    private Cliente clientes[];
    static final String nombreFichero = "clientes.dat"; //nombre del fichero

    //abre el fichero con la información de los clientes, en modo lectura, y carga los
    //datos en la tabla clientes
    public void cargaDatos() throws ClassNotFoundException {
        ObjectInputStream in = null;
        this.clientes = new Cliente[0]; //tabla de tamaño 0

        try { // leemos los datos previamente almacenados en el fichero (de entrada)
            in = new ObjectInputStream(new FileInputStream(nombreFichero));
            clientes = (Cliente[]) (in.readObject());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {// nos aseguramos de que el archivo se cierra
            if (in != null) {
                try {
                    in.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

```

//Comienza la gestión de los clientes (crear, modificar y eliminar) mientras no se
//introduzca la opción de salir
public void gestion() {
    Scanner sc = new Scanner(System.in);
    int opcion;
    do {
        opcion = menu();
        switch (opcion) {
            case 1: // recorremos la tabla para mostrar los clientes. Si no hay datos
                    //previos no se muestra nada
                    System.out.println("Clientes:\n" + Arrays.toString(clientes));
                    break;

            case 2:
                    insertaCliente(); //la función se encarga de leer los datos
                    break;

            case 3:
                    System.out.println("Identificador del cliente a eliminar: ");
                    int id = sc.nextInt();
                    eliminaCliente(id); //esta función no lee nada, se le pasa id
                    break;

            case 4:// nos limitaremos a pedir de nuevo los datos del cliente excepto el
                    //id (no modificable). Para cambiarlo hay que dar de baja al cliente
                    System.out.println("Identificador del cliente a modificar: ");
                    id = sc.nextInt();
                    modificaCliente(id);
                }
        } while (opcion != 5);
    }

//Vuelca el contenido de todos los clientes en fichero de clientes
public void salvaDatos() {
    ObjectOutputStream out = null;

    //antes de terminar el programa, volcaremos la tabla al fichero (ahora de salida)
    try {
        out = new ObjectOutputStream(new FileOutputStream(nombreFichero));
        out.writeObject(clientes);
        out.close();
    } catch (IOException ex) {
        System.out.println(ex.getMessage());
    } finally {
        if (out != null) {
            try {
                out.close();
            } catch (IOException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }
}

//Lee los datos e inserta un nuevo cliente, redimensionando la tabla con tamaño +1
private void insertaCliente() {
    Scanner sc = new Scanner(System.in);

    System.out.println("Datos del nuevo cliente");
    System.out.print("Identificador: ");
    int id = sc.nextInt(); //leemos los datos del nuevo cliente
    sc.nextLine(); //tras leer un número el \n se queda en el búfer, lo limpiamos
    //leyendo con nextLine(), antes de leer otra cadena
    System.out.print("Nombre: ");
    String nombre = sc.nextLine();
    System.out.print("Teléfono: ");
    String tlf = sc.nextLine();
}

```

```
//redimensionamos el tamaño de la tabla en +1, para el nuevo cliente
clientes = Arrays.copyOf(clientes, clientes.length + 1); //redimensionamos
//insertamos en la última posición de la tabla
clientes[clientes.length - 1] = new Cliente(id, nombre, tlf);
}

//Elimina el cliente, si existe, con identificador id. Redimensiona la tabla
private void eliminaCliente(int id) {
    // buscamos en la tabla el cliente con el id leído por teclado
    int pos = busca(id);

    if (pos == -1) {
        System.out.println("El cliente no existe");
    } else { // machamos el cliente a borrar (índice pos) con el último
        clientes[pos] = clientes[clientes.length - 1];
        // trabajamos con tablas completas: redimensionamos la tabla a un tamaño -1
        clientes = Arrays.copyOf(clientes, clientes.length - 1);
    }
}

//Busca el cliente con identificador id, y modifica sus datos, leyendo nuevos valores
private void modificaCliente(int id) {
    Scanner sc = new Scanner(System.in);

    int pos = busca(id); // buscamos el cliente en la tabla

    if (pos == -1) {
        System.out.println("El cliente no existe");
    } else {
        Cliente clienteAModificar = clientes[pos]; //referencia al cliente a modificar
        System.out.println("Introduzca datos del cliente");
        System.out.print("Nombre: ");
        clienteAModificar.nombre = sc.nextLine(); //leemos y asignamos el nuevo nombre
        System.out.print("Teléfono: : ");
        clienteAModificar.tlf = sc.nextLine(); //leemos y asignamos el nuevo teléfono
    }
}

//la función busca y devuelve la posición del cliente con identificador id, en la
//tabla this.clientes. En el caso de que no se encuentre, se devuelve -1
private int busca(int id) {
    //implementamos un algoritmo de búsqueda secuencial
    int pos = 0;

    while (pos < clientes.length && id != clientes[pos].id) {
        pos++;
    }
    // a la salida del bucle, si pos = t.length, no hemos encontrado nada
    if (pos == clientes.length) {
        pos = -1;
    }

    return pos;
}

//muestra las opciones del menú, lee una opción correcta y la devuelve
private int menu() {
    Scanner sc = new Scanner(System.in);

    int opc; //opción del menú

    //mostramos opciones
    System.out.println("1. Listar Clientes");
    System.out.println("2. Añadir Cliente");
    System.out.println("3. Eliminar Cliente");
    System.out.println("4. Modificar Cliente");
    System.out.println("5. Salir");
}
```



```

        do {
            System.out.print("Escriba opción: ");
            opc = sc.nextInt();
        } while (opc < 1 || opc > 5);

        return opc;
    }
}

```

Programa Principal

```

public class Main {
    static public void main(String args[]) throws ClassNotFoundException {
        GestionClientes gestion = new GestionClientes();

        gestion.cargaDatos(); //leemos los datos de los clientes
        gestion.gestion(); //tratamos (altas, bajas y modificaciones) de los clientes
        gestion.salvaDatos(); //volcamos los datos al fichero
    }
}

```

Ejercicios propuestos

- 11.1. Añadir al Ejercicio propuesto 7.1, la funcionalidad de leer y escribir toda la información gestionada por la aplicación en un fichero binario.
- 11.2. Hacer lo mismo que en el ejercicio anterior, para el Ejercicio propuesto 7.2.
- 11.3. Implementar un programa que permita dar de alta y de baja locomotoras y vagones (véase Ejercicio resuelto 7.6). Los objetos se guardarán en las tablas correspondientes de clase *Locomotoras* y *Vagón*. Al iniciarse el programa se cargarán desde el disco (fichero binario) dichas tablas y se mostrará el menú. Al salir del programa se volverán a guardar en el disco las tablas actualizadas. En cualquier momento se podrán ver por pantalla las locomotoras y los vagones. Después de cada operación volverá a mostrarse el menú, salvo cuando se solicite el fin del programa. El menú deberá tener la siguiente forma:

1. Alta locomotora.
2. Baja locomotora.
3. Mostrar locomotoras.
4. Alta vagón.
5. Baja vagón.
6. Mostrar vagones.
7. Salir.

Realizar dos versiones:

- a) Guardar y leer de disco cada objeto (*Locomotoras* o *Vagón*) de forma individual.
- b) Leer y escribir directamente las tablas como objetos.

Capítulo 12

Collections

A menudo necesitamos guardar información, pero que no sabemos de antemano el espacio que va a ocupar en la memoria. En estos casos, las tablas no son la solución adecuada, ya que su tamaño debe permanecer fijo una vez declaradas. En su lugar, necesitaremos estructuras dinámicas de datos, es decir, objetos que alberguen datos que se crean y se destruyen en tiempo de ejecución, según las necesidades de la aplicación. Para este fin, Java nos proporciona una serie de estructuras dinámicas que comparten un conjunto de métodos declarados en la interfaz `Collection` (véase Figura 12.1). Todas ellas implementan dicha interfaz, aunque de distinta forma.

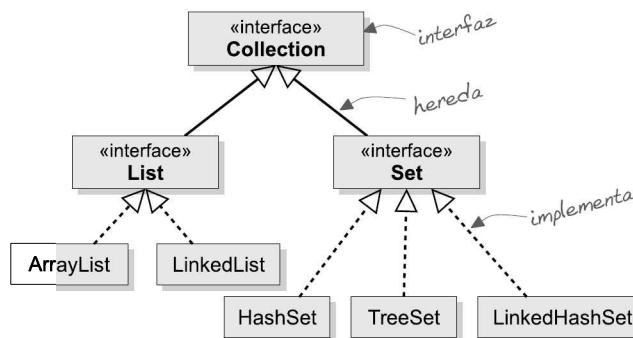


Figura 12.1. Clases que implementan las interfaces `List` y `Set`, que heredan de `Collection`

Hay tres tipos fundamentales de estructuras ligadas a la interfaz `Collection`,

- **Listas:** responden a la necesidad de manejar sucesiones de datos que pueden estar repetidos y cuyo orden puede ser importante. De alguna forma sustituyen a las tablas, con la diferencia de que podemos insertar o eliminar datos en ellas sin que sobre ni falte sitio.
- **Conjuntos:** el orden de los datos no es relevante y lo que realmente importa es la mera pertenencia, o no, de un dato a la estructura, con lo que las repeticiones tampoco tienen sentido.

- **Mapas o diccionarios:** relacionados con la interfaz *Collection*, aunque no la implementan. Sirven para guardar datos identificados por claves que no se repiten. Los estudiaremos al final del capítulo.

De estas estructuras, la más conocida y usada es la lista, aunque no es la única.

Todas estas estructuras son dinámicas, ya que permiten añadir y quitar unidades de información (objetos llamados nodos o elementos) en tiempo de ejecución, cambiando el tamaño total sin tener que volverlas a declarar. Nosotros, en general, llamaremos colecciones a todas las clases que implementen la interfaz *Collection*, aunque también usaremos el nombre particular de la clase implementada (*ArrayList*, por ejemplo).

Una particularidad de *Collection* es que trabaja con tipos genéricos de datos. Esto quiere decir que cuando declaremos una de estas estructuras, especificaremos la clase de objetos que se pueden insertar en ellas, lo que permite hacer un control de tipos más eficiente. Se puede pensar en una colección como una bolsa o contenedor donde guardar objetos. Para mantener la máxima generalidad, en un principio, se implementaron para trabajar con objetos *Object*, como ya hemos visto con otras interfaces, como *Comparable*. Pero entonces las comprobaciones de tipo se hacen en tiempo de ejecución, cuando ya es imposible evitar que se produzca una excepción por tipo incorrecto. En cambio, los tipos genéricos anticipan esos errores al tiempo de desarrollo, poniéndolos en manos del programador. Aquí no vamos a tratar en profundidad la programación con tipos genéricos y nos vamos a limitar a ilustrar su uso en las colecciones.

El conjunto de interfaces y clases del marco de trabajo *Collection* se halla en el paquete `java.util`.

12.1. Listas

Las *listas* nos servirán para almacenar datos que se pueden repetir y cuyo orden de inserción puede ser relevante. Hay dos implementaciones de lista, las clases *ArrayList* y *LinkedList*. Las dos proporcionan los mismos métodos y funcionalidades, ya que implementan la interfaz *List*, que hereda de *Collection*. La diferencia entre *ArrayList* y *LinkedList* radica en la implementación interna y solo afecta levemente al rendimiento. La primera es más rápida en las operaciones que impliquen recorrer la lista para la lectura o modificación de elementos, mientras que la segunda tiene mejor rendimiento en las operaciones de inserción y eliminación de nodos. Nosotros usaremos la primera en nuestros ejemplos, aunque la segunda nos valdría exactamente igual. En todo el código que vamos a escribir, podemos sustituir *ArrayList* por *LinkedList* sin alterar ningún resultado.

La sintaxis general para construir un *ArrayList* con un tipo genérico de datos E (sea *Cliente*, *Empleado*, *Integer*...) es,

```
ArrayList <E> lista = new ArrayList<>();
```

En esta lista solo se podrán insertar objetos (nodos) del tipo E. Si E es *Cliente*, solo se podrán insertar objetos *Cliente* o de una subclase de *Cliente* (no se debe olvidar que un objeto de una subclase de *Cliente* también es un *Cliente*). E debe ser siempre el nombre

de una clase, nunca un tipo primitivo. Cuando queramos insertar datos de tipos primitivos, como `int`, usaremos las clases envoltorio¹, como `Integer` o `Double`.

La declaración de un objeto de la clase `ArrayList` que nos sirva para guardar objetos de tipo `Cliente`, sería:

```
ArrayList<Cliente> cliel = new ArrayList<Cliente>();
```

En realidad, en el constructor no hace falta escribir `Cliente`, ya que el nombre de la clase se infiere de la parte izquierda de la declaración. Bastará poner,

```
ArrayList<Cliente> cliel = new ArrayList<>();
```

El operador `<>` de la derecha, llamado *diamante*, lo dejamos vacío.

Una vez creada la colección `cliel`, disponemos de una estructura dinámica donde insertar o eliminar objetos `Cliente`. Antes definamos una clase `Cliente` sencilla que nos permita probar los distintos métodos con ejemplos concretos,

```
class Cliente implements Comparable {
    String dni;
    String nombre;
    int edad;

    Cliente(String dni, String nombre, int edad) {
        this.dni = dni;
        this.nombre = nombre;
        this.edad = edad;
    }

    @Override
    public boolean equals(Object ob) {
        return dni.equals(((Cliente) ob).dni);
    }

    @Override
    public int compareTo(Object ob) {
        return dni.compareTo(((Cliente) ob).dni);
    }

    @Override
    public String toString(){
        return "Dni: " + dni + "Nombre: " + nombre + "Edad: " + edad + "\n";
    }
}
```

12.1.1. Métodos básicos de la interfaz `Collection`

Los métodos de la interfaz `Collection` son de dos tipos. Unos afectan a elementos individuales y otros a grupos de elementos. A los primeros los llamaremos métodos básicos y a los segundos métodos globales.

¹Las clases envoltorio se tratarán en el Anexo B.

Métodos de inserción

Son aquellos que sirven para añadir elementos nuevos en una colección.

boolean add(E elem): sirve para insertar un elemento en una lista. Se le pasa el objeto a insertar. Si la inserción tiene éxito, devuelve **true**. En caso contrario, **false**. Es común que un método devuelva **true** cuando, al ejecutarse, cambia la estructura de una colección y **false** si la colección queda inalterada. Ya veremos otros ejemplos. El nuevo nodo queda insertado al final de la lista. **E** es el tipo genérico con que se ha declarado la lista, en nuestro caso, **Cliente**. El compilador no nos va a permitir insertar un objeto de otro tipo. Probemos con nuestra lista de clientes,

```
Cliente cliente = new Cliente("111", "Marta", 20);
clie1.add(cliente);
```

Métodos de eliminación

boolean remove(Object ob): elimina un nodo **ob** de una lista. Devuelve **true** si la eliminación ha tenido éxito y **false** en caso contrario, por ejemplo, si el objeto no estaba en la lista. Por otra parte, debemos observar que no se exige a **ob** que sea del tipo genérico **E** con el que se ha declarado la lista, ya que el método no va a añadir ningún nodo, y no hay peligro de que se inserte un objeto de una clase no permitida. Para eliminar a Marta de nuestra lista escribiremos,

```
clie1.remove(cliente);
```

void clear(): nos permite eliminar todos los nodos de una lista y dejarla vacía. Esto no significa eliminar la propia lista, del mismo modo que vaciar una bolsa de caramelos no significa destruir la bolsa. La lista, simplemente, queda vacía. Después podremos volver a insertar nuevos elementos.

Métodos de comprobación

Insertemos algunos nodos para seguir experimentando,

```
clie1.add(new Cliente("111", "Marta", 20));
clie1.add(new Cliente("115", "Jorge", 21));
clie1.add(new Cliente("112", "Carlos", 18));
```

int size(): nos permite saber en cada momento el número de elementos (o nodos) insertados en una lista. Por ejemplo,

```
clie1.size() //devuelve 3
```

boolean isEmpty(): permite saber si una lista está vacía. Devuelve **true** si la lista está vacía y **false** en caso contrario. La expresión,

```
clie1.isEmpty() //devolverá false
```

boolean contains(Object ob): nos dice si un elemento **ob** determinado está en una lista. Devuelve **true** si **ob** pertenece a la lista y **false** en caso contrario. En nuestro ejemplo,

```
clie1.contains(new Cliente("115", "Jorge", 21)) //devuelve true
```

En la búsqueda del objeto **ob** (llamado clave de búsqueda), **contains()** usa el método **equals()** para compararlo con los nodos de la lista. En nuestro ejemplo con objetos **Cliente**, ese método está basado en el atributo **dni**. Por tanto, la expresión,

```
clie1.contains(new Cliente("115", "", 0))
```

también devolverá **true**, ya que, en la búsqueda del objeto, el programa solo va a comparar el **dni** de la clave de búsqueda con los de los distintos nodos de la lista. Esto nos permite hacer búsquedas sin conocer ni tener que escribir toda la información de un elemento. Basta conocer el o los atributos que usa el método **equals()**, en nuestro ejemplo, el **dni** del cliente.

Otros métodos

String toString(): devuelve una cadena que representa la colección. La clase **ArrayList**, igual que **LinkedList** y, como veremos, todas las colecciones, tienen implementado el método que muestra los elementos entre corchetes y separados por comas. Cada nodo se muestra, a su vez, según la implementación de **toString()** que tenga la clase genérica **E**, en nuestro caso la que hemos hecho para la clase **Cliente**. Por tanto, para mostrar los elementos de **clie1**, podemos escribir,

```
System.out.println(clie1);
```

Con nuestra implementación de **Cliente**, obtendríamos por pantalla,

```
[Dni: 111 Nombre: Marta Edad: 20
, Dni: 115 Nombre: Jorge Edad: 21
, Dni: 112 Nombre: Carlos Edad: 18
]
```

En nuestro ejemplo, los datos de los nodos aparecen en líneas distintas porque pusimos un carácter **'\\n'** al final de la cadena devuelta por **toString()** en la clase **Cliente**.

A menudo necesitamos recorrer una lista nodo a nodo. Una de las formas de hacerlo es por medio de iteradores, que son objetos que van apuntando sucesivamente a los nodos de la lista, empezando por el primero. Para usar un iterador con una lista concreta, primero hay que crearlo. El método **iterator()**, invocado por la lista, nos devuelve un iterador asociado a ella,

```
Iterator iterator()
```

Por ejemplo, si queremos recorrer nuestra lista de clientes, creamos el iterador con la sentencia,

```
Iterator it = clie1.iterator();
```

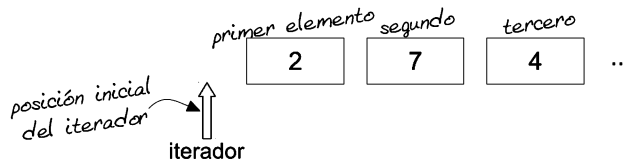
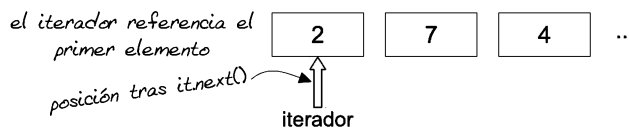


Figura 12.2. Posición inicial del iterador

`it` es el iterador que sirve para recorrer `clié1`. Para ello usaremos las funciones `hasNext()` y `next()`, que se complementan y emplean conjuntamente. Inicialmente `it` apunta al principio de la lista, justo antes del primer elemento (véase Figura 12.2).

boolean `hasNext()`: comprueba si quedan elementos por visitar y nos devuelve `true` o `false`, según el caso.

Object `next()`: nos devuelve una copia del siguiente elemento (en nuestro ejemplo, un `Cliente`), si existe, y avanza una posición en la lista, apuntando al nodo siguiente.

Figura 12.3. Posición del iterador tras el primer `next()`

En caso de que no haya siguiente, porque estemos al final de la lista o porque esta estuviera vacía, `next()` lanzará la excepción `NoSuchElementException`. La primera llamada a `next()` nos devuelve el primer elemento de la colección (Figura 12.3).

En la práctica, para evitar la excepción, los dos métodos se usan conjuntamente, de forma que solo se llama al método `next()` si antes se ha comprobado que hay elemento siguiente, con `hasNext()`. Veamos un trozo de código donde se crea un iterador `it` que empieza apuntando al principio de `clié1` y luego la recorre, con un bucle `for`, mostrando todos sus nodos,

```
Iterator it = clié1.iterator();
for ( ; it.hasNext(); ) {
    Cliente p=(Cliente) it.next();
    System.out.println(p);
}
```

En el bucle hay que observar varias cosas. En primer lugar, la declaración e inicialización del iterador `it` se podría haber incluido en la parte de inicialización del bucle `for`. En segundo lugar, la parte de los incrementos también está vacía. Esto se debe a que el método `next()`, además de devolver el nodo al que apunta, incrementa el iterador para que

apunte al siguiente nodo. Otra cosa que llama la atención es el cast (**Cliente**) delante de `it.next()`. Aquí es necesario porque este método devuelve un objeto **Object** y hay que avisar al compilador antes de asignarlo a una variable **Cliente**, de que en realidad es un objeto **Cliente**. Este cast se puede evitar definiendo el iterador con un tipo genérico, en este caso **Cliente**, en el instante de crearlo,

```
Iterator<Cliente> it = cliel.iterator();
```

Con ello avisamos al compilador de que `it` va a devolver objetos de la clase **Cliente** y podremos escribir,

```
Cliente p = it.next();
```

sin necesidad del cast.

Los iteradores tienen un tercer método que permite eliminar nodos de una lista.

void remove(): elimina de la lista, en cada momento, el último elemento devuelto por `next()`. Así podemos eliminar un nodo de una lista, dependiendo de alguna condición, mientras se está recorriendo con un iterador, de forma que se preserve la integridad de la colección. Aunque tenga el mismo nombre, no debemos confundirlo con el método `remove(Object ob)` de la interfaz **Collection**, que vimos antes. Aquél se llamaba desde un objeto lista (u otra colección) y no debe usarse dentro del bucle de un iterador. En su lugar, debemos invocar el del iterador, al que no se le pasa ningún parámetro. Por ejemplo, si queremos eliminar de nuestra lista aquellos clientes con edad mayor que 20,

```
Iterator<Cliente> it = cliel.iterator();
while(it.hasNext()) {
    Cliente p = it.next();
    if (p.edad > 20) {
        it.remove(); //elimina el último cliente devuelto por next()
        //¡ No usar cliel.remove(p)!
    }
}
```

Si mostramos los clientes, veremos que Jorge ha sido eliminado de la lista.

Una forma mucho más simple de recorrer una colección es por medio de la estructura **for-each**,

```
for (Cliente c : cliel) {
    System.out.println(c);
}
```

que se puede leer algo así como: “para cada `c` de tipo **Cliente** que pertenece a `cliel`, mostrar `c`”. En este bucle, la variable local `c`, de tipo **Cliente**, va tomando todos los valores de la lista `cliel`. Aquí se aplica a una lista, pero se puede usar con cualquier colección. Sin embargo, hay operaciones para las que no vale. Por ejemplo, no podemos eliminar nodos, ya que `c` es siempre la referencia a una copia de un elemento de la lista. Para eso tenemos que usar un iterador.

12.1.2. Métodos globales de la interfaz Collection

Hasta ahora hemos visto métodos de las listas que afectan a un solo elemento. Existen otros métodos, llamados métodos globales, en los que intervienen otras colecciones.

boolean containsAll(Collection <?> c): al que se le pasa como parámetro otra colección, por ejemplo una lista. La expresión <?> significa que la colección *c* que se pasa como parámetro es de un tipo genérico cualquiera (de ahí el signo de interrogación ?, que se conoce como *comodín*). Devuelve **true** si todos los elementos de *c* están en la colección que hace la llamada y **false** si hay al menos un elemento de *c* que no está en ella.

Para ilustrarlo vamos a crear una segunda lista de objetos *Cliente*,

```
ArrayList<Cliente> clie2 = new ArrayList<>();
clie2.add(new Cliente("111", "Marta", 20));
clie2.add(new Cliente("112", "Carlos", 18));
```

Hemos insertado dos clientes que ya estaban en la primera lista. Por tanto, esta contiene a todos los elementos de la segunda. La expresión,

```
clie1.containsAll(clie2)
```

devolverá **true**. Si ahora añadimos un elemento nuevo a *clie2*,

```
clie2.add(new Cliente("211", "Ana", 19))
```

la misma expresión,

```
clie1.containsAll(clie2)
```

devolverá **false**, ya que el nuevo elemento de *clie2* no está contenido en *clie1*.

boolean addAll(Collection <? extends E> c): inserta al final de la colección que hace la llamada todos los nodos de la colección *c*. La expresión <? extends E>, donde aparece de nuevo el comodín, significa un tipo genérico cualquiera con la condición de que herede del tipo *E* de la colección invocante.

En nuestro caso, como *clie1* se ha declarado como una lista del tipo genérico *Cliente* (esa sería *E*), podemos pasarle como parámetro cualquier colección que sea del tipo *Cliente* o de una subclase de *Cliente*. En particular, *clie2* cumple la condición, ya que es del tipo *Cliente*. Por tanto, podemos añadir sus elementos a *clie1*,

```
clie1.addAll(clie2);
```

Si mostramos *clie1* tal como ha quedado se obtiene,

```
[Dni: 111 Nombre: Marta Edad: 20
, Dni: 112 Nombre: Carlos Edad: 18
, Dni: 211 Nombre: Ana Edad: 19
, Dni: 111 Nombre: Marta Edad: 20
, Dni: 112 Nombre: Carlos Edad: 18
]
```

Lo primero que llama la atención es que todos los nodos de la segunda lista se han añadido a la primera, aunque estén repetidos. Esta es una característica de la listas, pero no de todas las colecciones. Cuando estudiemos los conjuntos, veremos que los elementos que ya estaban en la colección, no se añaden. Por tanto, nunca tendrán elementos repetidos como pasa en las listas.

Hay un método que hace lo contrario del anterior:

boolean removeAll(Collection <?> c): elimina de la colección invocante todos los nodos que estén contenidos en c. Después de ejecutar el método no habrá elementos comunes a las dos colecciones.

Para probarlo en nuestro ejemplo, vamos a eliminar de `clie1` todos los elementos incluidos en `clie2`. Pero antes vamos a reducir esta última eliminando a Marta,

```
clie2.remove(new Cliente("111","",0))
```

Como ya hicimos con el método `contains()`, solo necesitamos el DNI en el constructor del objeto clave que pasamos a `remove()`, ya que este es el atributo que usa `equals()` para buscar e identificar el nodo que tiene que eliminar (si en `clie2` estuviera repetido el nodo de Marta, solo se eliminaría una de las copias, la que aparece antes). Después de esta operación solo quedarán en `clie2` los nodos de Ana y Carlos. Pues bien, vamos a eliminar de `clie1` los elementos de `clie2`,

```
clie1.removeAll(clie2);
```

con lo cual, desaparecen todas las ocurrencias de Ana y Carlos, quedando solo los dos nodos de Marta.

Cuando queramos eliminar de una lista todas las ocurrencias de un nodo repetido, debemos tener en cuenta que `remove(Object ob)` solo elimina la primera que encuentra, empezando por el principio de la lista. Si usamos este método tendremos que implementar un bucle. Sin embargo, `removeAll(Collection<?> c)` elimina de la colección que hace la llamada a la función todas las ocurrencias de los nodos de c.

Otro método global es:

boolean retainAll(Collection <?> c): elimina todos los nodos de la colección invocante, salvo aquellos que también estén en c.

12.1.3. Métodos de tabla de la interfaz Collection

Hay una tercera categoría de métodos que comparten todas las colecciones: aquellos que sirven para volcar sus datos en tablas.

El método:

Object[] toArray(): devuelve una tabla de tipo `Object` con los mismos elementos de la colección. En el caso de listas, como en estas el orden importa, la tabla alberga los mismos elementos, incluyendo las repeticiones, en el mismo orden.

Para obtener en una tabla los elementos de `clie2`,

```
Object[] t1 = clie2.toArray();
```

La tabla `t1` tiene longitud 2 y contiene los objetos correspondientes a Ana y Carlos, pero para acceder a sus nombres tendremos que poner un cast,

```
((Cliente)t1[0]).nombre // devolverá "Ana"
```

Como vemos, el método anterior tiene el inconveniente de que devuelve una tabla de tipo `Object`, aunque sabemos que son clientes. Esto nos obliga a emplear un cast para acceder a los miembros de la clase a la que pertenece. Para soslayar este problema, podemos usar otra versión del método,

<T>T[] toArray(T[] t): que es igual que el anterior, pero devuelve una tabla de tipo genérico `T`. El método es invocado por la colección y, como parámetro, le pasamos una tabla del tipo genérico con que se ha definido. No hay que inicializar la tabla ni importa su tamaño. El método la devuelve redimensionada con el tamaño necesario para albergar todos los elementos de la colección. De hecho, es costumbre definirla con tamaño 0,

```
Cliente[] t2 = clie2.toArray(new Cliente[0]);  
System.out.println(t2[0].nombre); // "Ana"
```

Ahora `t2` es de tipo `Cliente` y recoge los elementos de `clie2`. Con este procedimiento no es necesario el cast delante de `t2[0]`.

12.1.4. Métodos específicos de la interfaz `List`

Todos los métodos vistos hasta ahora pertenecen a la interfaz `Collection` y son, por tanto, implementados por todas las clases que la implementan, aunque los hemos probado con listas. En realidad las listas implementan la interfaz `List`, que hereda de `Collection` y añade una serie de métodos y funcionalidades específicas de las listas, no compartidos por otras colecciones, como los conjuntos. La funcionalidad más importante exclusiva de las listas (ya sean `ArrayList` o `LinkedList`) es el acceso posicional a sus nodos, a través de índices. El primer nodo tiene índice 0, el segundo tiene índice 1 y así sucesivamente, como en las tablas. Vamos a crear una nueva lista, ahora con números enteros, que serán de la clase `Integer`².

`Integer` es un clase envoltorio. Los tipos genéricos de las colecciones no pueden ser primitivos. Necesariamente tienen que ser clases y sus valores, objetos. Para cada tipo primitivo de Java se ha definido una clase: `Integer`, `Double`, `Long`, `Boolean` y `Character`, cuyos objetos «envuelven» un valor primitivo (de ahí del nombre de clases envoltorio). La forma más general de hacerlo es con un constructor,

```
Integer x = new Integer(5);
```

Pero Java los crea automáticamente por medio de una operación llamada *autoboxing*, en muchos contextos; por ejemplo en una asignación.

²Las clases envoltorio se tratan con detalle en el Anexo B.

```
Integer x = 5;
```

Definamos la lista con el tipo genérico `Integer`,

```
ArrayList<Integer> num1 = new ArrayList<>();
```

Ahora vamos a insertar algunos elementos,

```
num1.add(3); //autoboxing. No hace falta poner, num1.add(new Integer(3))
num1.add(1);
num1.add(-2);
num1.add(0);
num1.add(3);
num1.add(7);
```

Si mostramos la lista con,

```
System.out.println(num1);
```

obtendremos,

```
[3, 1, -2, 0, 3, 7]
```

Los métodos más importantes aportados por la interfaz `List` son,

E `get(int indice)`: devuelve el elemento que ocupa el lugar `indice` en la lista, siendo 0 el índice del primer elemento, como en las tablas.

Por ejemplo,

```
num1.get(2)
```

devolverá `-2`, que es el valor del nodo que ocupa el tercer lugar de la lista, con índice 2.

E `set(int indice, E elem)`: guarda el elemento `elem` en la posición `indice`, machacando el valor que hubiera previamente en esa posición, que es devuelto por el método.

Con el siguiente código, vamos a poner el valor 10 en el nodo 3, sustituyendo su valor actual (0), que es devuelto por el método y asignado a la variable `y`,

```
Integer y = num1.set(3, 10);
System.out.println("y: " + y);
System.out.println(num1);
```

Se mostrará por pantalla,

```
y: 0
[3, 1, -2, 10, 3, 7]
```

void `add(int indice, E elem)`: inserta el valor `elem` en la posición `indice`, añadiéndose, sin machacar el valor previo. Todos los nodos que ocupaban una posición mayor o igual que `indice`, se desplazan una posición hacia el final de la lista, para dejar hueco al nuevo elemento.

Por ejemplo, si queremos insertar el valor 5 entre el `-2` y el 10, se insertará en la posición 3, que actualmente ocupa el 10. Este y los nodos que le siguen se desplazarán un lugar hacia el final de la lista,

```
num1.add(3, 5);  
System.out.println(num1);
```

Veremos por pantalla,

```
[3, 1, -2, 5, 10, 3, 7]
```

boolean addAll(int indice, Collection<? extends E> c): inserta todos los elementos de la colección *c* en la lista que invoca al método, empezando por el lugar *indice* y desplazando hacia el final todos los nodos de la lista original a partir de *indice*, incluido este. La colección *c* debe ser de un tipo compatible con nuestra lista, es decir, de la clase genérica *E*, declarada en la lista, o de una subclase de *E*.

Vamos a crear una segunda lista de *Integer*,

```
ArrayList<Integer> num2 = new ArrayList<>();  
num2.add(20);  
num2.add(30);  
num2.add(40);
```

Ahora insertamos esta lista en el lugar de índice 2 de *num1*, es decir, donde se encuentra el valor *-2*. Este nodo, junto con los que le siguen se desplazan hacia el final para hacer sitio a los tres valores insertados,

```
num1.addAll(2, num2);  
System.out.println(num1);
```

Aparecerá en pantalla,

```
[3, 1, 20, 30, 40, -2, 5, 10, 3, 7]
```

Para eliminar un elemento hay una versión sobrecargada del método **remove()** que ya conocemos,

E remove(int indice): elimina el nodo que ocupa el lugar *indice* y devuelve el elemento eliminado.

En este caso, hay que tener cuidado si lo usamos en una lista de objetos *Integer*, como la de nuestro ejemplo, ya que una sentencia como,

```
num1.remove(5);
```

no será interpretada como que queremos eliminar un elemento *Integer* con valor 5 de la lista, sino el elemento con índice 5, es decir, el nodo de valor 3, ya que, al haber una versión de **remove()** que admite un *int* como argumento, Java no hace autoboxing. En el caso de que quisiéramos eliminar un nodo *Integer* cuyo valor es 5, escribiríamos,

```
num1.remove(new Integer(5));
```

Así, estaríamos pasando como argumento un objeto y no un *int*, con lo cual Java ejecuta la versión primera del método y elimina el nodo de valor 5.

Además de los métodos de lectura, escritura, inserción y eliminación de nodos, la interfaz `List` añade un par de funciones de búsqueda.

int indexOf(Object ob): devuelve el índice de la primera ocurrencia de `ob` en la lista. Si no está, devuelve `-1`.

int lastIndexOf(Object ob): hace lo mismo que `indexOf()`, pero empezando la búsqueda por el final.

boolean equals(Object otraLista): las listas, tanto `ArrayList` como `LinkedList`, se pueden comparar por medio del método `equals()`, que devuelve `true` si ambas tienen exactamente los mismos elementos en el mismo orden.

12.2. Interfaz Set

La interfaz `Set` trata los datos como un conjunto matemático, sin un orden preestablecido y eliminando las repeticiones, aunque tiene una implementación que permite introducir un orden. Todos sus métodos los hereda de `Collection`. Lo único que añade es la restricción de no permitir duplicados. Esto significa que si intentamos insertar un nodo que ya existe, no lo hará.

El conjunto de métodos disponibles es el mismo que vimos en los apartados de métodos básicos y globales de las colecciones:

```
int size()
boolean isEmpty()
boolean contains(Object element)
boolean add(E element)
boolean remove(Object element)
Iterator<E> iterator()
boolean containsAll(Collection<?> c)
boolean addAll(Collection<? extends E> c)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
Object[] toArray()
<T> T[] toArray(T[])
```

Asimismo, podemos recorrer un conjunto con una estructura `for-each`, igual que las listas. Las diferencias más importantes son el orden en que se van insertando los nodos nuevos y que un nodo que ya está en el conjunto no se puede volver a insertar, ya que no son posibles los nodos repetidos. Cuando intentemos insertar un nodo repetido con el método `add()`, no se producirá ningún error ni se arrojará ninguna excepción; sencillamente, el nodo no se inserta y la función devuelve `false`.

Sin embargo no tendremos los métodos propios de listas, que vienen declarados en la interfaz `List`. En particular, no es posible el acceso posicional por medio de índices.

Las implementaciones de `Set` son las clases: `HashSet`, `TreeSet` y `LinkedHashSet`.

- **HashSet**: tiene un buen rendimiento, aunque no garantiza ningún orden en la inserción.
- **TreeSet**: a pesar de tener peor rendimiento, garantiza el orden basado en el valor de los elementos insertados. El criterio de ordenación se lo proporciona un comparador en el constructor, o bien es el natural (el proporcionado por el método `compareTo()`), si no se especifica nada en el constructor.
- **LinkedHashSet**: garantiza el orden basado en la inserción, ya que siempre inserta los nodos al final.

Al contrario de lo que pasa con las listas, las implementaciones de **Set** tienen diferencias en su comportamiento. Es muy común utilizar variables de tipo **Set** para referenciarlos. Esto permite aprovechar el polimorfismo de las distintas implementaciones y, como veremos, hacer transformaciones de unas en otras.

Por ejemplo, vamos a declarar un conjunto con un orden natural, basado en su implementación de la interfaz **Comparable** del tipo genérico,

```
TreeSet<Cliente> c1 = new TreeSet<>();
```

donde la clase **Cliente** implementa el método `compareTo()` basado en el DNI. Los nodos se insertarán ordenados por el atributo `dni`, ordenación natural de los objetos **Cliente**.

Insertamos los mismos nodos que en la lista `clie1` y los mostramos,

```
c1.add(new Cliente("111", "Marta", 20));
c1.add(new Cliente("115", "Jorge", 21));
c1.add(new Cliente("112", "Carlos", 18));
System.out.println(c1);
```

Veremos por pantalla,

```
[Dni: 111 Nombre: Marta Edad: 20
, Dni: 112 Nombre: Carlos Edad: 18
, Dni: 115 Nombre: Jorge Edad: 21
]
```

Lo primero que salta a la vista es el orden en que aparecen, que no coincide con el orden de inserción, sino que están por `dni` creciente, como corresponde a la implementación que tenemos hecha para la interfaz **Comparable** de la clase **Cliente**.

Si ahora intentamos volver a insertar uno de los nodos anteriores, por ejemplo, el de Marta,

```
boolean insertado = c1.add(new Cliente("111", "Marta", 20));
System.out.println(insertado); //false, ya que no se ha insertado
System.out.println(c1);
```

aparece por pantalla,

```
false
[Dni: 111 Nombre: Marta Edad: 20
, Dni: 112 Nombre: Carlos Edad: 18
, Dni: 115 Nombre: Jorge Edad: 21
]
```


donde vemos que la inserción ha devuelto **false** y que el conjunto no ha cambiado.

Pero si queremos otro conjunto de clientes ordenados por nombre, lo creamos pasando al constructor, como argumento, un comparador basado en el atributo nombre. Antes vamos a revisar la interfaz **Comparator**, que ya estudiamos en el capítulo de interfaces.

En su momento, al hablar de la interfaz **Comparator**, definimos el método,

```
int compare(Object ob1, Object ob2)
```

que, como vimos, exige poner sendos cast a los objetos **ob1** y **ob2** que se quieren comparar para poder acceder a los atributos. Allí pusimos como ejemplo la implementación de **Comparator** para comparar personas por el atributo **edad**,

```
public class ComparaEdades implements Comparator {
    @Override
    public int compare(Object ob1, Object ob2) {
        Persona p1=(Persona)ob1;
        Persona p2=(Persona)ob2;

        return p1.edad-p2.edad;
    }
}
```

Sin embargo, la interfaz **Comparator** puede recibir tipos genéricos. En el ejemplo anterior escribiríamos,

```
public class ComparaEdades implements Comparator <Persona> {
    @Override
    public int compare(Persona p1, Persona p2) {
        return p1.edad-p2.edad;
    }
}
```

Al haber especificado la clase genérica, ya no hacen falta los cast.

Volviendo a nuestra clase **Cliente**, un comparador para nombres sería,

```
public class ComparaNombres implements Comparator<Cliente> {
    @Override
    public int compare(Cliente cli1, Cliente cli2) {
        return cli1.nombre.compareTo(cli2.nombre);
    }
}
```

Ahora podemos crear un conjunto con un orden basado en los nombres de los clientes. Esto se lleva a cabo usando un **TreeSet**, a cuyo constructor le pasamos un comparador de la clase **ComparaNombres**,

```
TreeSet<Cliente> c2 = new TreeSet<>(new ComparaNombres());

c2.add(new Cliente("111", "Marta", 20));
c2.add(new Cliente("115", "Jorge", 21));
c2.add(new Cliente("112", "Carlos", 18));
System.out.println(c2);
```

Veremos por pantalla,

```
[Dni: 112 Nombre: Carlos Edad: 18
, Dni: 115 Nombre: Jorge Edad: 21
, Dni: 111 Nombre: Marta Edad: 20
]
```

donde vemos los tres nodos ordenados por nombres.

Si queremos que los nodos se vayan colocando por orden de inserción, como en las listas, en vez de un `TreeSet` crearemos un objeto `LinkedHashSet`. Si el orden nos es indiferente, podemos usar un `HashSet`, que tiene un mejor rendimiento.

12.2.1. Conversiones entre colecciones

Una característica interesante de los conjuntos y de todas las colecciones en general es la posibilidad de transformar unas en otras por medio de los constructores. Por ejemplo, para obtener un conjunto ordenado (un `TreeSet`) a partir de otro que no lo está (un `HashSet` o `LinkedHashSet`) o, dicho de otra forma, si queremos ordenar un conjunto, disponemos de dos caminos a seguir,

1. Construimos un `TreeSet` con el criterio de ordenación que deseamos y luego le añadimos con `addAll()` el conjunto a ordenar.
2. Si el criterio de ordenación va a ser el natural (el de la interfaz `Comparable`), podemos construir un `TreeSet` pasando como argumento al constructor el conjunto que queremos ordenar.

Para ilustrar los dos métodos, vamos a crear un `LinkedHashSet` de números enteros. Después le vamos a añadir 5 nodos, que se irán colocando por orden de inserción,

```
Set<Integer> s1 = new LinkedHashSet<>();
s1.add(4);
s1.add(1);
s1.add(5);
s1.add(10);
s1.add(3);
System.out.println(s1);
```

Obtenemos,

```
[4, 1, 5, 10, 3]
```

Hemos usado el tipo `Set`, es decir, el nombre de la interfaz, para la variable `s1`. Esto es una práctica común si queremos tener la posibilidad de referenciar, con la misma variable, conjuntos con distintas implementaciones. Si la variable `s1` fuera de tipo `LinkedHashSet`, solo serviría para referenciar objetos de esa clase, pero no un `TreeSet`. Es una forma más del polimorfismo de Java. Volviendo a nuestro ejemplo, si queremos obtener un conjunto ordenado a partir de los elementos de `s1`, podemos crear un `TreeSet` y añadirse los,

```
Set<Integer> s2 = new TreeSet<>();
s2.addAll(s1);
System.out.println(s2);
```

obteniendo,

```
[1, 3, 4, 5, 10]
```

Ahora podemos mantener las dos variables `s1` y `s2`, cada una referenciando un tipo distinto de conjunto, o referenciar con `s1` el nuevo conjunto ordenado,

```
s1 = s2;
```

con lo cual, a todos los efectos, habríamos ordenado `s1`. A partir de ese momento `s1` sería un `TreeSet` y mantendría el orden con la inserción o eliminación de nodos.

La segunda forma de ordenar un conjunto es pasarlo al constructor de un `TreeSet`,

```
Set<Integer> s2 = new TreeSet<>(s1);
```

con lo que obtendríamos el mismo resultado. No obstante, este segundo método solo sirve si queremos un conjunto con el orden natural de los nodos. Si, en vez de enteros, tuviéramos un conjunto de clientes sin ordenar y quisiéramos ordenarlos por nombre, tendríamos que usar el primer procedimiento, construyendo un `TreeSet` con un comparador `ComparaNombres`,

```
Set<Cliente> s1 = new LinkedHashSet<>(); //conjunto sin orden
s1.add(new Cliente("111", "Marta", 20));
s1.add(new Cliente("115", "Jorge", 21));
s1.add(new Cliente("112", "Carlos", 18));
Set<Cliente> s2 = new TreeSet<>(new ComparaNombres());
s2.addAll(s1); //el mismo conjunto ordenado por nombres
System.out.println(s2);
```

que nos muestra los clientes ordenados por nombre.

En realidad, utilizando los constructores, es posible hacer conversiones entre todo tipo de colecciones. Se pueden crear listas pasando conjuntos a su constructor y viceversa; también se pueden añadir a una lista todos los elementos de un conjunto. Un ejemplo útil es la creación de un conjunto a partir de una lista para eliminar elementos repetidos,

```
List<Integer> lista = new ArrayList<>();
lista.add(5);
lista.add(3);
lista.add(5); //elemento repetido
lista.add(2);
lista.add(5); //elemento repetido
Set<Integer> s = new TreeSet<>(lista); //conjunto sin elementos repetidos
System.out.println(s);
```

Obtenemos,

```
[2, 3, 5]
```

donde se han eliminado las repeticiones y, de paso, por ser un `TreeSet`, se han ordenado los elementos restantes.

Cuando se trabaja con colecciones de distinto tipo, siempre que se usen constructores o métodos comunes a listas y conjuntos, es costumbre definir las variables de tipo `Collection` para permitir la referencia a diferentes tipos de colección con una misma variable en caso de conversiones. El código anterior podría ser,

```
Collection<Integer> col = new ArrayList<>();
lista.add(5);
...
col = new TreeSet<>(col);
System.out.println(col);
```

dando los mismos resultados. En este caso, con `col` podemos referenciar cualquier lista o conjunto. El comportamiento dependerá del objeto referenciado: otro caso de polimorfismo.

12.2.2. Clase Collections

Además de los métodos aportados por la interfaces `Collection`, `List` y `Set` con sus implementaciones, la clase `Collections` (no confundirla con la interfaz `Collection`) reúne una serie de utilidades en forma de métodos estáticos que trabajan con tipos genéricos. En ellos, el primer parámetro de entrada es la colección sobre la que deseamos operar y comprenden métodos de búsqueda, ordenación y manipulación de datos entre otros. Casi todos ellos operan sobre listas, aunque algunos valen para cualquier colección. El símbolo `T` es el tipo genérico de las colecciones y los nodos involucrados.

Métodos de ordenación

La clase `Collections` posee métodos sobrecargados para ordenar. El primero es,

`static void sort(List<T> lista):` ordena la lista que se le pasa como argumento. Esta será de un tipo genérico `T`, es decir, podemos pasar una lista de cualquier tipo. El criterio de ordenación será el llamado criterio «natural», que es el que establecerá el método `compareTo()` de la interfaz `Comparable` para la clase `T` (el tipo de los nodos de la lista, sea el que sea). Las clases envoltorio (wrapper) proporcionadas por Java, como `Integer`, `Character` o `Double`, así como la clase `String` lo traen implementado, de forma que ordenan los números de menor a mayor, los caracteres según el orden Unicode y los `String`, por orden alfabético.

Veamos un ejemplo. Creamos una lista `ArrayList`, para objetos `Cliente`,

```
List<Cliente> lista = new ArrayList<>();
```

e insertamos varios elementos,

```
lista.add(new Cliente("111", "Marta", 20));
lista.add(new Cliente("115", "Jorge", 21));
lista.add(new Cliente("112", "Carlos", 18));
```

Si queremos ordenar la lista, solo tenemos que escribir,

```
Collections.sort(lista);
```

La lista se ordenará con el criterio natural del tipo con el que se declaró. En nuestra clase `Cliente`, por `dni`. Si queremos ordenar con otro criterio, tendremos que usar comparadores. Para ello disponemos de otra versión sobrecargada de `sort()`, en la que se añade como segundo parámetro el comparador con el criterio deseado. Si queremos ordenar por nombre, escribiremos,

```
Collections.sort(lista, new ComparaNombres());
```

Métodos de búsqueda

Uno de los métodos más importantes de la clase `Collections`, `binarySearch()`, hace una búsqueda binaria de un objeto, llamado *clave de búsqueda*, en una lista que debe estar ordenada previamente. Todo ello necesita un criterio de ordenación que, por defecto, es el natural del tipo genérico de la lista. No obstante, aquí la sintaxis es un poco más complicada. Se exige que la implementación de `Comparable` sea también genérica. Esto, en la clase `Cliente` sería de la forma,

```
class Cliente implements Comparable<Cliente> {
    ...
    public int compareTo(Cliente ob) {
        return dni.compareTo(ob.dni);
    }
    ...
}
```

que es similar a lo que hicimos con la interfaz `Comparator` cuando implementamos el comparador `ComparaEdades` con los `TreeSet`. Allí lo hicimos para ahorrar escritura, pero aquí es una exigencia de la sintaxis de `binarySearch()`.

Vamos a ver un ejemplo con lista. En primer lugar, la volvemos a ordenar por DNI, que es el orden natural,

```
Collections.sort(lista); //ordenada por dni (orden natural)
```

Al método `binarySearch()` se le pasan como parámetros la lista en la que queremos hacer la búsqueda y el objeto clave que queremos buscar. Devuelve el índice de este último si lo encuentra en la lista. Por ejemplo, si queremos buscar a Carlos, cuyo DNI es "112",

```
int indice = Collections.binarySearch(lista, new Cliente("112", null, 0));
```

En caso de que la clave no esté en la lista, devolverá un entero negativo del que se puede deducir el índice `indiceInsercion` que le correspondería al nodo si lo insertáramos manteniendo la lista ordenada. La fórmula es,

```
indiceInsercion = -indice - 1
```

No olvidemos que `indice` sería un número negativo que, con el menos que tiene delante, da un valor positivo. El valor de `indiceInsercion` será siempre mayor o igual que 0. Supongamos que queremos insertar a Eva en la lista en caso de que no esté. Para ello, primero la buscamos con `binarySearch()`; si la búsqueda nos da un valor negativo, calculamos su índice de inserción y la insertamos,

```
Cliente nuevo = new Cliente("555", "Eva", 17);
int indice = Collections.binarySearch(lista, nuevo);
if (indice < 0) { //no esta en la lista
    lista.add(-indice - 1, nuevo); //lista sigue ordenada
}
```

Si queremos hacer una búsqueda en una lista ordenada con un criterio distinto al natural, tendremos que pasar a `binarySearch()` como tercer parámetro, el mismo comparador que se usó para ordenarla. Por ejemplo si ordenamos lista por orden alfabético de nombres,

```
Collections.sort(lista, new ComparaNombres());
```

Ahora, para buscar a Carlos, debemos hacerlo por nombre,

```
indice = Collections.binarySearch(lista, new Cliente(null, "Carlos", 0),  
                                   new ComparaNombres());
```

que devolverá 0, ya que Carlos es el primero de la lista por orden alfabético de nombres.

El método `binarySearch()` es extremadamente eficiente y sus tiempos de búsqueda son muy cortos en comparación con otros métodos de búsqueda, como el secuencial. El inconveniente es que precisa que la lista esté ordenada previamente. Merece la pena ordenar la lista si vamos a tener que hacer muchas búsquedas con el mismo criterio. En caso contrario, es más eficiente hacer una búsqueda secuencial, por medio de un bucle.

Métodos para la manipulación de datos

Si queremos intercambiar dos nodos en una lista usaremos,

`static void swap(List<?> lista, int i, int j):` intercambia los nodos con índices *i* y *j* entre sí.

Pongamos un ejemplo con una lista de enteros,

```
List<Integer> num1 = new ArrayList<>();  
num1.add(1); //índice 0  
num1.add(2);  
num1.add(3);  
num1.add(4); //índice 3  
num1.add(5);  
Collections.swap(num1, 0, 3); //cambia los elementos con índices 0 y 3
```

La lista quedaría,

```
[4, 2, 3, 1, 5]
```

Para reemplazar todas las ocurrencias de un nodo determinado por otro,

`static boolean replaceAll(List<T> lista, T valorAntiguo, T valorNuevo):` reemplaza el nodo `valorAntiguo`, en todos los lugares en que aparezca en la lista, por `valorNuevo`.

Por ejemplo, si queremos reemplazar los nodos que valgan 4 por el valor 100,

```
Collections.replaceAll(num1, 4, 100);
```

dando,

```
[100, 2, 3, 1, 5]
```

Podemos rellenar todos los nodos que tiene una lista con un valor que pasamos como parámetro:

static void fill(List<? super T> lista, T valorDeRelleno): sustituye todos los valores de los nodos de la lista por el de **valorDeRelleno**. La lista debe ser de tipo **<? super T>**, es decir, de la clase **T** o cualquier superclase de **T**. Dicho de otra forma, el nodo de relleno debe ser de la clase de la lista o de una subclase. Esto garantiza que el elemento de relleno, de la clase **T**, se pueda insertar en ella.

Rellenemos **num1** con el valor 7,

```
Collections.fill(num1, 7);
```

quedando **num1** como,

```
[7, 7, 7, 7, 7]
```

Para copiar una lista en otra usamos:

static void copy(List<? super T> destino, List<? extends T> origen): copia los elementos de la lista origen en la lista destino, empezando por el principio y sobrescribiendo los valores previos. La lista destino deberá ser, como mínimo, del tamaño de la lista origen. Los nodos de la lista origen a insertar deben ser de clase compatible con la lista destino. Por eso, esta última debe ser de clase **T** o superclase de **T** (**<? super T>**) y la lista origen debe ser clase **T** o subclase de **T** (**<? extends T>**). En particular, si las dos listas son de la misma clase genérica, se podrán copiar sin problema.

Si construimos una segunda lista de enteros,

```
List<Integer> num2 = new ArrayList<>();
num2.add(1);
num2.add(2);
num2.add(3);
```

Ahora vamos a copiarla en **num1**,

```
Collections.copy(num1, num2);
```

con lo que **num1** queda,

```
[1, 2, 3, 7, 7]
```

12.2.3. Otras utilidades

A veces hace falta que los elementos estén desordenados. Por ejemplo, en aplicaciones de juegos, es útil la función,

static void shuffle(List<?> lista): significa *barajar* en inglés, desordena los elementos de lista.

Si escribimos,

```
Collections.shuffle(num1);
```

num1 quedará desordenada³.

static int frequency(Collection<?> col, Object ob): nos devuelve el número de veces que aparece un nodo en una colección. Tiene sentido con listas, ya que en los conjuntos no hay repeticiones.

Por ejemplo,

```
Collections.frequency(num1, 7)
```

devolverá 2, ya que hay dos 7 en num1.

static T max(Collection<? extends T> col): nos devuelve el valor máximo de una colección (no tiene por qué ser una lista). Busca el máximo basándose en el orden natural. Eso exige que la clase genérica de los nodos tenga implementada la interfaz Comparable.

Por ejemplo,

```
Integer maximo = Collections.max(num1);
```

nos dará 7. Si queremos el valor máximo atendiendo a un criterio de ordenación distinto del natural, le pasaremos a **max()** un segundo parámetro con un comparador adecuado.

static T max(Collection<? extends T> col, Comparator<? super T> comp):
devuelve el máximo utilizando **comp** como criterio de comparación.

Por ejemplo, volviendo al conjunto de **Cliente**, **s1**, con Marta, Carlos y Jorge, si queremos obtener el máximo, es decir, el nodo que ocuparía el último lugar si el conjunto estuviera ordenado por orden alfabético de nombres, pondremos,

```
Cliente ultimo = Collections.max(s1, new ComparaNombres());
```

con lo que obtendríamos a Marta.

Es importante resaltar que para llamar al método **max()** hace falta un criterio de ordenación, pero eso no implica que la colección tenga que estar ordenada. En ninguno de los dos ejemplos anteriores lo estaba.

Hay métodos análogos para calcular el mínimo de una colección, que funcionan exactamente igual,

```
Integer minimo = Collections.min(num1);  
Cliente primero = Collections.min(s1, new ComparaNombres());
```

También podemos invertir el orden de una lista con:

³En realidad, Java utiliza una fórmula para generar valores pseudoaleatorios, con lo cual el desorden es aparente. Pero el efecto es el mismo, ya que el usuario es incapaz de predecir los resultados.

void reverse(List<?> lista): invierte lista, colocando los nodos en orden inverso. Aquí es importante observar que la función no devuelve una nueva lista invertida; invierte la lista original.

Por último, podemos crear un conjunto a partir de un nodo con:

Set<T> singleton(T elem): devuelve un conjunto con el tipo genérico T del nodo. Es un conjunto inmutable, es decir, no podemos añadir más nodos ni eliminar el que ya está. Se suele emplear para eliminar un nodo repetido de una lista sin necesidad de usar un bucle.

Por ejemplo, para eliminar el 7, que aparece dos veces en `num1`, escribimos,

```
num1.removeAll(Collections.singleton(7));
```

Ahora `num1` será,

```
[2, 1, 3]
```

12.3. Interfaz Map

Los *mapas* o *diccionarios* son estructuras dinámicas cuyos nodos, que aquí se llaman *entradas* (objetos de la clase `Map.Entry`), son pares *Clave/Valor* en vez de valores individuales como en las colecciones. Todas ellas implementan la interfaz `Map`, que no hereda de `Collection`. Por tanto, los mapas no son colecciones, aunque están íntimamente relacionados con ellas y funcionan dentro del mismo entorno de trabajo. Vamos a usar tres implementaciones de `Map`: `HashMap`, `TreeMap` y `LinkedHashMap`, que se diferencian entre sí de forma similar a `HashSet`, `TreeSet` y `LinkedHashSet`.

En un mapa se insertan nodos, llamados entradas, que constan de una clave, que no se puede repetir, y un valor asociado con ella, que sí puede estar repetido. Un mapa es una estructura semejante a la aplicación matemática⁴.

Las operaciones fundamentales en un mapa son la inserción, la lectura y la eliminación de entradas, aunque veremos algunas más.

Para ilustrar el uso de mapas vamos a empezar utilizando la implementación `HashMap`, que no garantiza ningún orden de inserción en las entradas, aunque es muy eficiente en cuanto a la velocidad de acceso a los datos. El constructor más sencillo es de la forma,

```
Map<K, V> m = new HashMap<>();
```

donde K es el tipo de las claves y V el de los valores. Son tipos genéricos que, necesariamente, serán clases y no tipos primitivos. Como hicimos con los conjuntos, hemos elegido `Map` como tipo de la variable `m` (podríamos haber puesto `HashMap`), con objeto de garantizar la posibilidad de un mayor polimorfismo. El comportamiento de `m` estará determinado por la clase del objeto referenciado. Como ejemplo vamos a suponer que queremos mantener la información de las estaturas de un grupo de escolares, con entradas en las que figura el nombre del alumno (clase `String`) como clave y la estatura (clase envoltorio `Double`, ver Anexo B) como valor,

⁴De hecho, la traducción al inglés de aplicación matemática es *mapping*.

```
Map<String, Double> m = new HashMap<>();
```

Para insertar entradas usamos el método:

V put(K clave, V valor): se le pasan como parámetros la clave y el valor asociado con ella. Si no había ninguna entrada previa con la misma clave, se inserta en el mapa la nueva entrada con esa clave y ese valor, y el método devuelve `null`. Si ya había una entrada con la misma clave, se sustituye el valor antiguo por el nuevo y la función devuelve el valor antiguo. Insertemos unas cuantas entradas,

```
m.put("Ana", 1.65);  
m.put("Marta", 1.60);  
m.put("Luis", 1.73);  
m.put("Pedro", 1.69);
```

Con los mapas, igual que con los conjuntos, disponemos también de una implementación de `toString()`, de forma que podemos visualizarlos,

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

obteniéndose por pantalla,

```
{Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.69}
```

Si ahora queremos cambiar la estatura de Pedro, insertamos otra vez un nodo con la misma clave y el nuevo valor,

```
m.put("Pedro", 1.71);
```

obteniéndose,

```
{Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.71}
```

Si queremos eliminar un nodo:

V remove(Object k): elimina la entrada cuya clave es `k`, si existe. En este caso, devuelve el valor asociado con esa clave. En caso contrario, devuelve `null`.

Para eliminar todas las entradas de un mapa, llamamos a la función:

void clear(): que elimina todas las entradas, dejando el mapa vacío.

Si queremos conocer el valor de una entrada a partir de su clave:

V get(Object k): devuelve el valor asociado con la clave `k` o `null` si no hay ninguna entrada con esa clave.

Por ejemplo,

```
m.get("Ana");
```

devuelve 1.65.

Para saber si una determinada clave está presente en un mapa:

boolean containsKey(Object k): devuelve **true** si hay una entrada con la clave **k**

Por ejemplo,

```
m.containsKey("Ana");
```

devolverá **true**.

Análogamente, para saber si hay alguna entrada con un valor determinado:

boolean containsValue(Object v): **true** si hay alguna entrada con valor **v**.

Dos mapas se pueden comparar entre sí con el método **equals()**, que devuelve **true** si ambos tienen exactamente las mismas entradas.

12.3.1. Vistas *Collection* de los mapas

Aunque **Map** no hereda de **Collection**, los mapas están íntimamente ligados a las colecciones, de forma que se trabaja simultáneamente con ambas interfaces a través de distintas vistas con estructura de colección. Por *vista* entendemos una colección respaldada por el mapa original, de forma que cuando accedemos a un nodo de la vista estamos accediendo a la entrada original en el mapa, con lo que los cambios que se hagan en aquella se reflejarán en este. Hay tres tipos de vistas de un mapa. En primer lugar, podemos obtener una vista de las claves del mapa. Para ello disponemos del método,

Set<K> keySet(): nos devuelve una vista, con estructura **Set**, de las claves presentes en un mapa.

Para obtener las claves del mapa del ejemplo escribimos:

```
Set<String> claves = m.keySet();
System.out.println(claves);
```

mostrará,

```
[Marta, Ana, Luis, Pedro]
```

con corchetes, que es la representación de las colecciones.

También podemos obtener una vista de los valores del mapa por medio del método:

Collection<V> values(): devuelve una vista **Collection** de los valores. Si alguno se encuentra más de una vez en el mapa, también aparece repetido en la colección devuelta.

En nuestro ejemplo,

```
Collection<Double> valores1 = m.values();
System.out.println(valores1);
```

devolverá,

```
[1.6, 1.65, 1.73, 1.71]
```

Y por último, disponemos de un método para obtener una vista de las entradas:

Set<Map.Entry<K, V>> entrySet(): devuelve una vista conjunto de las entradas, objetos de la clase `Map.Entry`, de las que se puede obtener la clave con `getKey()` o el valor con `getValue()`.

En nuestro mapa,

```
Set<Map.Entry<String, Double>> e1 = m.entrySet();
System.out.println(e1);
```

obteniéndose,

```
[Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.71]
```

Se puede usar la vista de entradas para acceder a las entradas individuales y obtener la clave o el valor, o bien para cambiar su valor, con los métodos de la clase `Map.Entry`:

K getKey(): devuelve la clave de la entrada.

V getValue(): devuelve el valor de la entrada.

V setValue(V nuevoValor): asigna `nuevoValor` a la entrada y devuelve el valor el antiguo.

Uno de los inconvenientes de los mapas es que no son iterables. Esto supone que, además de no poder usar iteradores para recorrerlos ni eliminar nodos, tampoco es posible el uso de la estructura `for-each`, cosa que sí podemos hacer con las vistas obtenidas, ya que son colecciones. Como hemos visto, los cambios que hagamos en ellas se reflejarán en el mapa. En particular, podemos eliminar elementos del conjunto de claves devuelto por `keySet()` por medio de los métodos `remove()` de `Iterator`, `remove()` de `Set`, `removeAll()` o `retainAll()`, con los cuales se eliminarán las entradas correspondientes en el mapa. Por ejemplo, si eliminamos la clave “Marta” del conjunto claves,

```
claves.remove("Marta");
```

obtenemos,

```
{Ana=1.65, Luis=1.73, Pedro=1.71}
```

donde vemos que la entrada correspondiente a Marta ha desaparecido.

La única forma segura de eliminar entradas durante un proceso de iteración sobre cualquiera de las tres vistas es el método `remove()` de la interfaz `Iterator`. Veamos un ejemplo, pero antes vamos a añadir algunas entradas a nuestro mapa,

```
m.put("Lucas", 1.8);
m.put("Marta", 1.60);
m.put("Jorge", 1.75);
```

con lo que tenemos,

```
{Marta=1.6, Ana=1.65, Luis=1.73, Lucas=1.8, Pedro=1.71, Jorge=1.75}
```

Ahora vamos a filtrar el mapa eliminando todas aquellas entradas con estatura mayor que 1.71. Para ello iteramos sobre el conjunto de las entradas,

```
Set<Map.Entry<String, Double>> e2 = m.entrySet(); //conjunto de entradas
for (Iterator<Map.Entry<String,Double>> it = e.iterator(); it.hasNext();) {
    Map.Entry<String, Double> e1 = it.next();
    if (e1.getValue() > 1.71) {
        it.remove();
    }
}
```

Obtendremos,

```
{Marta=1.6, Ana=1.65, Pedro=1.71}
```

Esto mismo podríamos haberlo hecho iterando sobre la vista de los valores,

```
Collection<Double> valores2 = m.values();
for (Iterator<Double> it = valores2.iterator(); it.hasNext();) {
    Double v = it.next();
    if (v > 1.71) {
        it.remove();
    }
}
```

En cambio, no podemos añadir entradas a un mapa por medio de `add()` o `addAll()` a través de ninguna de sus vistas de tipo colección.

12.3.2. Implementaciones de Map

En todos los ejemplos de mapas hemos usado la implementación `HashMap`, que destaca por su eficiencia, pero que no garantiza ningún orden en la inserción de los nodos. La interfaz `Map` tiene otras dos implementaciones, `TreeMap` y `LinkedHashMap`.

`TreeMap`, a semejanza de `TreeSet`, tiene una estructura de árbol que permite una inserción ordenada y una búsqueda rápida y eficiente de los nodos. Las entradas se insertan por orden natural creciente de las claves. Por ejemplo,

```
TreeMap<String, Double> tm = new TreeMap<>();
tm.put("Ana", 1.65);
tm.put("Marta", 1.60);
tm.put("Luis", 1.73);
tm.put("Pedro", 1.71);
```

`tm` quedará,

```
{Ana=1.65, Luis=1.73, Marta=1.6, Pedro=1.71}
```

Podemos hacer que el orden de un `TreeMap` sea distinto. Para ello le pasamos un comparador al constructor como parámetro de entrada, igual que hacíamos con `TreeSet`. En cualquier caso, el orden siempre se refiere a las claves, nunca a los valores.

Por último, la implementación `LinkedHashMap` mantiene el orden en que se van insertando los nodos, de forma similar a lo que ocurre con `LinkedHashSet`. Es muy eficiente en las operaciones de inserción y eliminación de entradas y algo más lento en las búsquedas.

Ejercicios de Collections

- 12.1.** Crear una colección de 20 números enteros aleatorios menores que 100, y guardarlos en el orden en que se vayan generando; mostrar por pantalla dicha lista una vez creada. Ordenarla en sentido creciente y volverla a mostrar por pantalla.

```
import java.util.*;

/*
 * Los números aleatorios pueden repetirse, por este motivo no utilizaremos colecciones
 * del tipo Set (que elimina los repetidos). En su lugar utilizaremos List.
 * Recordar que Math.random() genera números aleatorios entre 0 y 1.
 */
public class Main {

    public static void main(String[] args) {
        List<Integer> lista = new ArrayList<>();
        Integer temp;
        for (int i = 0; i < 20; i++) {
            temp = (int) (Math.random() * 100); // generamos un entero entre 0 y 99
            lista.add(temp); // lo insertamos al final de la lista
        }

        // Todas las clases de Collection traen implementada toString():
        System.out.println("Lista con el orden de inserción: ");
        System.out.println(lista);
        // Ordenamos la lista según el orden natural de Integer:
        Collections.sort(lista);
        System.out.println("Lista en orden creciente: ");
        System.out.println(lista);
    }
}
```

- 12.2.** Repetir el ejercicio anterior, pero ordenar la lista en sentido decreciente.

```
import java.util.*;

/*
 * Igual que en el ejercicio anterior, pero ordenando la lista en sentido decreciente.
 */
public class Main {

    public static void main(String[] args) {
        List<Integer> lista = new ArrayList<>();
        Integer temp;
        for (int i = 0; i < 20; i++) {
            temp = (int) (Math.random() * 100); // generamos un entero entre 0 y 99
            lista.add(temp); // lo insertamos al final de la lista
        }

        System.out.println("Lista con el orden de inserción: ");
        System.out.println(lista);

        // Para ordenar la lista en orden decreciente, creamos el comparador:
        EnterosDecrecientes comp = new EnterosDecrecientes();
        Collections.sort(lista, comp);
        System.out.println("Lista en orden decreciente: ");
        System.out.println(lista);
    }
}
```

```

/*
 * Creamos una clase comparadora que ordene los números enteros de mayor a menor. */
class EnterosDecrecientes implements Comparator <Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
}

```

- 12.3.** Crear una colección de 20 números enteros aleatorios distintos menores que 100, guardarlos por orden decreciente a medida que se vayan generando y mostrar la colección por pantalla.

```

import java.util.*;

/* Creamos una colección que permita inserción ordenada. Como los números no se pueden
 * repetir, usamos un conjunto, en vez de una lista. Además, escogeremos la implementa-
 * ción TreeSet para mantener un orden, pasando el criterio de ordenación
 * al constructor. */
public class Main {

    public static void main(String[] args) {
        Set<Integer> conj = new TreeSet<>(new EnterosDecrecientes());

        while (conj.size() < 20) { //mientras no existan 20 distintos en la colección
            Integer temp = (int) (Math.random() * 100); // generamos un entero entre 0 y 99
            conj.add(temp); // lo insertamos en el conjunto
        }

        System.out.println("Números aleatorios insertados en orden decreciente: ");
        System.out.println(conj);
    }
}

/* Creamos una clase comparadora que ordene los números enteros de mayor a menor. */
class EnterosDecrecientes implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        return (Integer) o2 - (Integer) o1;
    }
}

```

- 12.4.** Repetir el ejercicio anterior, pero esta vez permitir números repetidos y utilizar números aleatorios menores que 10.

```

import java.util.*;

/* El hecho de que se puedan repetir, descarta el uso de conjuntos. Desgraciadamente,
 * Collection no dispone de una implementación de List dotada de un determinado criterio
 * de ordenación. Tendremos que insertar los elementos ordenados de forma manual.
 * Usamos LinkedList por razones de eficiencia a la hora de insertar nuevos nodos. */
public class Main {

    public static void main(String[] args) {
        List<Integer> lista = new LinkedList<>();

        lista.add((int) (Math.random() * 10)); //insertamos el primer número aleatorio
        //Los otros 19 se van insertando en su sitio:
        for (int i = 1; i < 20; i++) {
            Integer temp = (int) (Math.random() * 10); // generamos el número

```

```
        //Buscamos su lugar entre los que ya están colocados:
        int indiceInsercion = 0;
        while (indiceInsercion < i && temp < lista.get(indiceInsercion)) {
            indiceInsercion++;
        }
        lista.add(indiceInsercion, temp); //insertamos en la posición adecuada
    }
    System.out.println("Números aleatorios en orden decreciente: ");
    System.out.println(lista);
}
}
```

- 12.5.** Introducir por teclado, hasta que se introduzca «fin», una serie de nombres que se insertarán en una colección, de forma que se conserve el orden de inserción y que no puedan repetirse. Mostrar la lista por pantalla.

```
import java.util.*;

/*
 * Para evitar elementos repetidos utilizamos un conjunto, con una implementación
 * LinkedHashSet, que respeta el orden de inserción. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Set<String> nombres = new LinkedHashSet<>();

        System.out.print("Introducir nombre (\\"fin\\" para terminar): ");
        String temp = sc.nextLine();
        while (!temp.equalsIgnoreCase("fin")) {
            nombres.add(temp);
            System.out.print("Introducir nombre: ");
            temp = sc.nextLine();
        }

        System.out.println("Lista de nombres sin repetidos:\\n" + nombres);
    }
}
```

- 12.6.** Introducir por teclado, hasta que se introduzca «fin», una serie de nombres, que se insertarán por orden alfabético en una colección que no permita repeticiones. Mostrar luego la lista de nombres por pantalla.

```
import java.util.*;

/*
 * Para evitar repeticiones utilizamos un conjunto, concretamente un TreeSet, que inserta
 * los String por orden alfabético.
 */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Set<String> nombres = new TreeSet<String>(); //creamos el conjunto

        System.out.print("Introducir nombre (\\"fin\\" para terminar): ");
        String temp = sc.nextLine();
        while (!temp.equalsIgnoreCase("fin")) {
            nombres.add(temp);
        }
    }
}
```



```

        System.out.print("Introducir nombre: ");
        temp = sc.nextLine();
    }
    System.out.println("Datos ordenados alfabéticamente, sin repetidos:\n" + nombres);
}
}

```

- 12.7. Implementar una función a la que se pase una lista de nombres y devuelva una copia sin elementos repetidos (sin modificar la original), con el prototipo,

List eliminaRepetidos(List c)

```

import java.util.*;

public class Main {

    //Aprovechamos la implementación de LinkedHashSet, que elimina los repetidos
    static List eliminaRepetidos(List c) {
        Set temp = new LinkedHashSet(c); //elimina los repetidos, sin reordenar
        return new ArrayList(temp); //la reconvertimos en lista
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<String> lista = new ArrayList<>();

        System.out.print("Introducir nombre (\nfin\n para terminar): ");
        String nombre = sc.nextLine();
        while (!nombre.equalsIgnoreCase("fin")) {
            lista.add(nombre);
            System.out.print("Introducir nombre: ");
            nombre = sc.nextLine();
        }

        System.out.println("lista original: ");
        System.out.println(lista);

        System.out.println("Sin repetidos: ");
        lista = eliminaRepetidos(lista);
        System.out.println(lista);
    }
}

```

- 12.8. Introducir por consola una frase que conste exclusivamente de palabras separadas por espacios. Almacenar en una lista las palabras de la frase, una en cada nodo y mostrar por pantalla las palabras que estén repetidas. A continuación, mostrar las que no lo estén.

```

import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introducir frase: ");
        String frase = sc.nextLine(); //suponemos palabras separados por solo un espacio
        String palabras[] = frase.split(" "); //separamos la frase en palabras:
        List<String> listaPalabras = new ArrayList<>(); //recorremos todas la palabras
    }
}

```

```
for (String p : palabras) {
    listaPalabras.add(p); //e insertamos al final de la lista
}
//una alternativa sería: listaCompleta.addAll(Arrays.asList(palabras));

Set<String> palabrasRepe = new LinkedHashSet<>(); //conjunto de palabras repetidas

/* La estrategia para encontrar las palabras repetidas es eliminarlas de la lista,
si después de eliminada una palabra, continúa en la lista es que estaba repetida.
Trabajaremos con un duplicado de la lista original para no perderla. */
List<String> aux = new ArrayList<>(listaPalabras); //duplicado de la lista

while (aux.size() > 0) {//mientras queden palabras en la lista
    String temp = aux.remove(0);//remove devuelve el elemento eliminado
    if (aux.contains(temp)) {//si quedan más en aux
        palabrasRepe.add(temp); //es que estaba repetida: la añadimos a repetidas
    }
}
System.out.print("Palabras repetidas:\n" + palabrasRepe);

aux = new ArrayList<>(listaPalabras); //hacemos otra copia de la lista
aux.removeAll(palabrasRepe); //elimina todas las ocurrencias de palabras repetidas

System.out.print("\nPalabras no repetidas:\n" + aux);
}
}
```

12.9. Implementar el método unión de dos conjuntos, que devuelva un nuevo conjunto con todos los elementos que pertenezcan, al menos, a uno de los dos conjuntos,

Set union(Set conjunto1, Set conjunto2)

```
import java.util.*;

public class Main {
    /* Creamos un conjunto donde añadir los elementos de los dos conjuntos. Como los
conjuntos no permiten repetidos, el conjunto resultante es la unión de ambos. */
    static Set union(Set conj1, Set conj2) {
        Set resultado = new HashSet(conj1); //copiamos el primer conjunto en resultado
        resultado.addAll(conj2);//añadimos los elementos del segundo conjunto
        return resultado; //el resultado es la unión
    }

    /*
    * Si se quiere hacer un control de tipos en tiempo de compilación usamos un método
    * parametrizado con tipos genéricos:
    * static Set<E> union(Set<E> conj1, Set<E> conj2) {
    *     Set<E> resultado = new HashSet<E>(conj1);
    *     resultado.addAll(conj2);//añadimos los elementos del segundo conjunto
    *     return resultado;
    * }
    */
    //probamos el método
    public static void main(String[] args) {
        Set<Integer> s1 = new HashSet<>();
        Set<Integer> s2 = new HashSet<>();

        for (int i = 1; i <= 6; i++) { //añadimos a s1 los números del 1 al 6
            s1.add(i);
        }
        for (int i = 3; i <= 10; i++) { //añadimos a s2 los números del 3 al 10
            s2.add(i);
        }
    }
}
```

```

        System.out.println("Conjunto 1: " + s1);
        System.out.println("conjunto 2: " + s2);
        System.out.println("\nUnión: " + union(s1, s2));
    }
}

```

12.10. Hacer lo mismo que en el ejercicio anterior con la intersección, formada por los elementos comunes a los dos conjuntos,

Set interseccion(Set conjunto1, Set conjunto2)

Solución a)

```

import java.util.*;

public class Main {

    /*La idea es recorrer el primer conjunto con un iterador y comprobar si cada
    elemento se encuentra en el segundo conjunto. El elemento que pertenece
    a ambos conjuntos, pertenece a la intersección. */
    static Set interseccion(Set conj1, Set conj2) {
        Set interseccion = new HashSet(); //creamos el conjunto resultante
        for (Iterator it = conj1.iterator(); it.hasNext();) { //recorremos conj1
            Object temp = it.next(); //obtenemos el elemento referenciado por el iterador
            if (conj2.contains(temp)) { //si el elemento se encuentra en el conj2
                interseccion.add(temp); //lo añadimos al conjunto "interseccion"
            }
        }
        return interseccion;
    }

    /*
    * Si se quiere hacer un control de tipos en tiempo de compilación
    * usamos un método parametrizado con tipos genéricos:
    * static Set<E> interseccion(Set<E> conj1, Set<E> conj2) {
    *     Set<E> interseccion = new HashSet<E>();
    *     for (Iterator<E> it = conj1.iterator(); it.hasNext();) {
    *         E temp = it.next();
    *         if (conj2.contains(temp)) {
    *             interseccion.add(temp);
    *         }
    *     }
    *     return interseccion;
    * }
    */
    //probamos el método
    public static void main(String[] args) {
        Set<Integer> s1 = new HashSet<>();
        Set<Integer> s2 = new HashSet<>();

        for (int i = 1; i <= 6; i++) { //añadimos los números del 1 al 6
            s1.add(i);
        }
        for (int i = 3; i <= 10; i++) { //añadimos los números del 3 al 10
            s2.add(i);
        }

        System.out.println("Conjunto 1: " + s1);
        System.out.println("Conjunto 2: " + s2);
        System.out.println("Intersección: " + interseccion(s1, s2));
    }
}

```

Solución b)

```
import java.util.*;

public class Main {
    /* Aprovechamos el método retainAll() de Set, que elimina todos los elementos de un
    conjunto, salvo los pertenecientes al conjunto pasado como parámetro de entrada.*/
    static Set interseccion(Set conj1, Set conj2) {
        Set interseccion = new HashSet(); //creamos el conjunto resultante

        interseccion.addAll(conj1); //Añadimos los elementos de conj1 a interseccion
        interseccion.retainAll(conj2); //elimina todos lo elementos de interseccion,
        //salvo los que estén en conj2. Solo quedan los comunes a ambos conjuntos
        return interseccion;
    }

    public static void main(String[] args) {
        Set<Integer> s1 = new HashSet<>();
        Set<Integer> s2 = new HashSet<>();
        for (int i = 1; i <= 6; i++) {
            s1.add(i);
        }
        for (int i = 3; i <= 10; i++) {
            s2.add(i);
        }
        System.out.println("Conjunto 1: " + s1);
        System.out.println("Conjunto 2: " + s2);
        System.out.println("Intersección: " + interseccion(s1, s2));
    }
}
```

12.11. Diseñar un método que devuelva la diferencia de dos conjuntos (elementos que pertenecen al primero, pero no al segundo). Con la sintaxis,

Set diferencia(Set conjunto1, Set conjunto2)

Solución a)

```
import java.util.*;

public class Main {
    /* Seguiremos el siguiente algoritmo: recorreremos el primer conjunto, quedándonos con
    los elementos que no se encuentran en el segundo conjunto. */

    static Set diferencia(Set conj1, Set conj2) {
        Set resultado = new HashSet(); //conjunto resultante

        for (Iterator it = conj1.iterator(); it.hasNext(); ) {
            Object temp = it.next(); //obtenemos el elemento referenciado por it
            if (!conj2.contains(temp)) { //si el elemento no está en conj2
                resultado.add(temp); //lo añadimos al conjunto resultante
            }
        }
        return resultado;
    }

    /* Si se quiere hacer un control de tipos en tiempo de compilación usamos un método
    * parametrizado con tipos genéricos:
    * static Set<E> diferencia(Set<E> conj1, Set<E> conj2) {
    *     Set resultado = new HashSet();
    *     for (Iterator<E> it = conj1.iterator(); it.hasNext(); ) {
    *         E temp = it.next();
```

```

*         if (!conj2.contains(temp)) {
*             resultado.add(temp);
*         }
*     }
*     return resultado;
* }
*/
//probamos el método
public static void main(String[] args) {
    Set<Integer> s1 = new HashSet<>();
    Set<Integer> s2 = new HashSet<>();

    for (int i = 1; i <= 6; i++) { //números del 1 al 6, inclusive
        s1.add(i);
    }
    for (int i = 3; i <= 10; i++) { //números del 3 al 10, inclusive
        s2.add(i);
    }
    System.out.println("Conjunto 1: " + s1);
    System.out.println("Conjunto 2: " + s2);
    System.out.println("Diferencia: " + diferencia(s1, s2)); //s1 - s2
}
}

```

Solución b)

```

import java.util.*;

public class Main {

    /* Utilizamos el método removeAll() de Set, que elimina de un conjunto todos
    los elementos del conjunto que se pasa como parámetro */
    static Set diferencia(Set conj1, Set conj2) {
        Set resultado = new HashSet(conj1); //creamos el conjunto idéntico a conj1
        resultado.removeAll(conj2); //eliminamos los elementos que estén en conj2
        return resultado;
    }

    //probamos el método
    public static void main(String[] args) {
        Set<Integer> s1 = new HashSet<>();
        Set<Integer> s2 = new HashSet<>();
        for (int i = 1; i <= 6; i++) {
            s1.add(i);
        }
        for (int i = 3; i <= 10; i++) {
            s2.add(i);
        }
        System.out.println("Conjunto 1: " + s1);
        System.out.println("Conjunto 2: " + s2);
        System.out.println("Diferencia: " + diferencia(s1, s2));
    }
}

```

- 12.12.** Escribir el método `incluido()`, que devuelve *true* si todos los elementos del primer conjunto pertenecen al segundo y *false* si hay algún elemento del primero que no pertenezca al segundo. Su sintaxis es,

```
boolean incluido(Set conjunto1, Set conjunto2)
```

```
import java.util.*;

public class Main {

    //Hay un método de Collection que realiza esa tarea
    static boolean incluido(Set conj1, Set conj2) {
        return conj2.containsAll(conj1); //si conj2 contiene todos los elementos de conj1
    }

    //probamos el método
    public static void main(String[] args) {
        Set<Integer> s1 = new HashSet<>();
        Set<Integer> s2 = new HashSet<>();

        for (int i = 1; i <= 6; i++) { //s1 contiene los números del 1 al 6
            s1.add(i);
        }
        for (int i = 1; i <= 4; i++) { //s2 contiene los números del 1 al 4
            s2.add(i);
        }
        System.out.println("Conjunto 1: " + s1);
        System.out.println("Conjunto 2: " + s2);
        System.out.println("Incluido conj1 en conj2: " + incluido(s1, s2));
        System.out.println("Incluido conj2 en conj1: " + incluido(s2, s1));
    }
}
```

12.13. Implementar una función a la que se le pasen dos listas ordenadas y nos devuelva una única lista, fusión de las dos anteriores. Desarrollar el algoritmo de forma no destructiva, es decir, que las listas utilizadas como parámetros de entrada se mantengan intactas.

```
import java.util.*;

public class Main {

    /* La estrategia será recorrer las dos listas simultáneamente, comparando los elementos de una y otra, de forma que nos quedamos con el menor de los dos en cada momento, hasta que se acabe una de las dos listas. Siempre se acaba una lista antes que la otra. Entonces debemos recorrer el resto de la lista que no se ha terminado. */
    static List<Integer> fusion(List<Integer> lista1, List<Integer> lista2) {
        List<Integer> resultado = new ArrayList<>();
        //Recorremos las dos listas a la vez:
        int i = 0, j = 0;
        while (i < lista1.size() && j < lista2.size()) {
            if (lista1.get(i) < lista2.get(j)) {
                resultado.add(lista1.get(i));
                i++;
            } else {
                resultado.add(lista2.get(j));
                j++;
            }
        }
        /* Copiamos los posibles elementos restantes de una y otra. Una de las dos listas, ya estará agotada y ni siquiera se entrará en el bucle correspondiente. */
        while (i < lista1.size()) {
            resultado.add(lista1.get(i));
            i++;
        }
    }
}
```

```

        while (j < lista2.size()) {
            resultado.add(lista2.get(j));
            j++;
        }
        return resultado;
    }

    public static void main(String[] args) {
        List<Integer> l1 = new ArrayList<>();
        List<Integer> l2 = new ArrayList<>();

        l1.add(1); l1.add(2); l1.add(5); l1.add(7); l1.add(7); l1.add(10);

        l2.add(1); l2.add(3); l2.add(4); l2.add(8); l2.add(10); l2.add(12); l2.add(12);
        l2.add(15); l2.add(18);

        System.out.println("Lista 1: " + l1);
        System.out.println("Lista 2: " + l2);
        System.out.println("Fusión: " + fusion(l1, l2));
    }
}

```

12.14. Implementar la función `leeCadena`, con el siguiente prototipo:

```
List<Character> leeCadena()
```

Dicha función lee una cadena por teclado y nos la devuelve en una lista con un carácter en cada nodo.

```

import java.util.*;

public class Main {

    //Recorreremos la cadena carácter a carácter, insertando estos en una lista
    static List<Character> leeCadena() {
        Scanner sc = new Scanner(System.in);
        List<Character> resultado = new ArrayList<>();

        System.out.print("Introduzca una frase: ");
        String cadena = sc.nextLine();

        for (int i = 0; i < cadena.length(); i++) { //recorremos
            resultado.add(cadena.charAt(i)); //insertamos
        }
        return resultado;
    }

    //probamos el método
    public static void main(String[] args) {
        List<Character> lista = leeCadena();
        System.out.println("Lista: " + lista);
    }
}

```

12.15. Implementar la función `uneCadenas`, con el prototipo:

```
List<Character> uneCadenas(List<Character> cad1,
                           List<Character> cad2)
```

que devuelva una lista con la concatenación de `cad1` y `cad2`

```
import java.util.*;

public class Main {
    /* Inicializamos la lista resultante con la primera lista a través del constructor,
    para luego añadirle al final la segunda lista. */
    static List<Character> uneCadenas(List<Character> cad1, List<Character> cad2) {

        List<Character> resultado = new ArrayList<>(cad1); //inicializamos con cad1
        resultado.addAll(cad2); //añadimos cad2 al final de la lista resultante
        return resultado;
    }

    public static void main(String[] args) {
        List<Character> cadena1 = new ArrayList<>();
        List<Character> cadena2 = new ArrayList<>();

        cadena1.add('S');
        cadena1.add('a');
        cadena1.add('n');
        cadena1.add('c');
        cadena1.add('h');
        cadena1.add('o');
        cadena2.add(' ');
        cadena2.add('P');
        cadena2.add('a');
        cadena2.add('n');
        cadena2.add('z');
        cadena2.add('a');

        System.out.println("uneCadenas: " + uneCadenas(cadena1, cadena2));
    }
}
```

12.16. Implementar la función:

`List clonaLista(List)`

que realice una copia exacta de una lista.

```
import java.util.*;

public class Main {

    /* Definimos el método con un tipo genérico E. Esto significa que la lista que se le
    pasa como parámetro de entrada tendrá datos de una clase E y que la lista devuelta
    los tendrá de la misma clase. El prototipo es: */
    static <E> List<E> clonaLista(List<E> lista) {
        //Basta crear una lista nueva pasando la original al constructor:
        return new ArrayList<>(lista);
    }

    /* Si queremos copiar la implementación (ArrayList o LinkedList) de lista
    * static <E> List<E> clonaLista(List<E> lista) {
    *     List<E> res;
    *     if (lista.getClass().equals(ArrayList.class)) {
    *         res = new ArrayList<>(lista);
    *     } else {
    *         res = new LinkedList<>(lista);
    *     }
    *     return res;
    * }
    */
}
```



```

//probamos el método
public static void main(String[] args) {
    List<Integer> original = new ArrayList<>();
    for (int i = 1; i <= 10; i++) { //lista de 10 elementos aleatorios
        original.add((int) (Math.random() * 100)); //insertamos un número entre 0 y 99
    }
    List<Integer> copia = clonaLista(original); //clonamos
    System.out.println("Original: " + original);
    System.out.println("Copia: " + copia);
}
}

```

12.17. Definir una clase `ListaOrdenada`, que hereda de `LinkedList`, que permita la inserción ordenada. Codificar un método que inserte un nuevo elemento con el prototipo:

```
void insertarOrdenado(E elemento)
```

```

import java.util.*;

/*
 * Definimos nuestra clase con el mismo tipo genérico E que tenga el LinkedList del que
 * hereda. Por otra parte, se le dota del criterio de ordenación por medio de un
 * comparador, que se le pasa en el constructor, y que será un atributo de la lista. */
class ListaOrdenada<E> extends LinkedList<E> {
    protected Comparator <E> comparador;

    //Solo implementamos un constructor (que utiliza el comparador) para obligar a que
    //toda ListaOrdenada tenga un orden definido.
    ListaOrdenada(Comparator c) {
        super();
        comparador = c;
    }

    void insertarOrdenado(E elemento) {
        if (size() == 0) { //si la lista estaba vacía.
            add(elemento); //insertamos al final
        } else { //si la lista no estaba vacía, buscamos el lugar correspondiente
            int indiceInsercion = 0;
            while (indiceInsercion < size()
                && comparador.compare(elemento, get(indiceInsercion)) > 0) {
                indiceInsercion++;
            }
            add(indiceInsercion, elemento); //insertamos en una posición concreta
        }
    }
} //de la clase ListaOrdenada<E>

/* Para probar la clase, vamos a implementar dos criterios de comparación:
 * Creamos una clase comparadora que ordene los números enteros de mayor a menor. */
class EnterosDecrecientes implements Comparator <Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
} //de la clase EnterosDecrecientes

/* Implementamos un comparador para cadenas, basado en el criterio de ordenación natural
 * (orden alfabético) de la clase String. */
class ComparaCadenas implements Comparator <String> {
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);
    }
} //de la clase ComparaCadenas

```

```
// Programa principal para probar la clase ListaOrdenada
public class Main {

    public static void main(String[] args) {
        //insertamos una serie de enteros aleatorios en orden decreciente, utilizando el
        //comparador EnterosDecrecientes
        ListaOrdenada<Integer> lista1 = new ListaOrdenada<>(new EnterosDecrecientes());

        for (int i = 0; i < 20; i++) {
            Integer temp = (int) (Math.random() * 10); // generamos el número
            lista1.insertarOrdenado(temp); //insertamos
        }

        System.out.println("Números aleatotos insertados en orden decreciente:");
        System.out.println(lista1);
        //La misma clase, al ser genérica, se puede usar para otros tipos de datos y
        // otros criterios de ordenación. Por ejemplo, para construir una lista de
        // nombres ordenada alfabéticamente (creciente)
        ListaOrdenada<String> nombres = new ListaOrdenada<>(new ComparaCadenas());
        nombres.insertarOrdenado("Juan");
        nombres.insertarOrdenado("Ana");
        nombres.insertarOrdenado("Pedro");
        nombres.insertarOrdenado("Mónica");
        nombres.insertarOrdenado("Juan");
        System.out.println("\nLista ordenada de nombres: ");
        System.out.println(nombres);
    }
}
```

Ejercicios propuestos

- 12.1. Diseñar una *suite*⁵ que ayude a gestionar un procedimiento selectivo de una administración (oposiciones). La suite estará formada por tres aplicaciones:

Introducir Aspirantes v 1.0: que ayuda a introducir los datos de los aspirantes.

Calificación Pruebas v 1.0: que ayuda a introducir la calificación obtenida por cada opositor en cada prueba.

Aprobados v 1.0: que genera un listado con los aspirantes que han superado las oposiciones.

Las tres aplicaciones harán uso de los mismos ficheros de datos.

Para introducir los datos de los aspirantes, debemos crear la aplicación Introducir Aspirantes v 1.0, que recoja de cada aspirante sus nombre, DNI y teléfono. La aplicación asignará un número identificativo incremental a cada aspirante. Toda la información recolectada se volcará a un fichero para su posterior uso.

La aplicación Calificación Pruebas v 1.0, leerá el número identificativo de cada aspirante (para mantener el anonimato) y la calificación de la última prueba realizada. Cuando se cierre la aplicación, toda la información debe almacenarse en disco. Para introducir las calificaciones la aplicación se ejecutará tantas veces como pruebas compongan el proceso selectivo. El número de pruebas depende de cada oposición.

⁵Conjunto de aplicaciones.

La última aplicación, Aprobados v 1.0, generará la lista de los aspirantes que han superado las pruebas selectivas. Para ello, se realiza la media de todas las pruebas, previa lectura del número de plazas se genera un informe con el formato:

Nombre	DNI	Calificación media
-----	-----	-----
xxxxxxxxxxxxx	99999999	99.99
xxxxxxxxxxxxx	99999999	99.99
. . .	. . .	. . .

de los aspirantes que han superado las oposiciones, ordenado alfabéticamente por el nombre del aspirante.

Capítulo 13

Stream

Las colecciones aportan versatilidad y potencia al procesamiento y la manipulación de datos complejos. Sin embargo, para recorrerlas disponemos de los iteradores, cuyo manejo puede resultar incómodo. A partir de Java 8, se ha introducido una serie de herramientas que permiten efectuar operaciones globales con los elementos de una colección completa, sin necesidad de recorrerlas nodo a nodo, aprovechando el procesamiento paralelo¹ de una forma transparente al programador. También pueden encadenarse, una a continuación de otra, formando tuberías, para dar un resultado final, sin necesidad de acceder a resultados intermedios. Aquí vamos a introducir los conceptos más importantes, como los **Stream**, los agregados o las tuberías, con su uso más frecuente. Un estudio más completo de todo este entorno de trabajo sobrepasa el propósito de este libro.

13.1. Interfaces funcionales y expresiones lambda

En el Capítulo 9, donde estudiamos las interfaces, distinguíamos entre métodos por defecto, estáticos y abstractos. De todos ellos, en la definición de la clase, solamente hay que implementar los últimos. Se llaman *interfaces funcionales* a aquellas que tienen un solo método abstracto. Son especialmente importantes porque tienen una sintaxis alternativa que permite una implementación más sencilla. Esto ha hecho que, de un tiempo a esta parte, proliferen las interfaces funcionales para tareas específicas que surgen con frecuencia en el trabajo del programador. Quizá la más conocida es la interfaz **Comparator**, que ya hemos usado repetidas veces y que nos va a servir de ejemplo.

A la hora de implementar una clase comparadora, podemos seguir varios caminos. Lo vamos a ilustrar manejando la lista de clientes del capítulo anterior. Supongamos que, en determinados momentos, queremos hacer una ordenación o una búsqueda por nombres, para lo cual necesitamos un comparador basado en el atributo nombre.

¹En un programa, a menudo, hay trozos de código (tareas a realizar) que se pueden ir ejecutando sin esperar a que hayan terminado otras que vienen escritas antes. Ejecuciones simultáneas de dos partes del código se conocen como *ejecución paralela* y se lleva a cabo con objetos de la clase **Thread**, llamados hilos de ejecución. Resultan eficaces en procesadores con varios núcleos, cuando cada hilo se ejecuta en un núcleo distinto, bajo el control del sistema. El estudio detallado de los hilos se sale del ámbito de este libro.

Primera forma. Creamos explícitamente una clase `ComparaNombres`, que implemente la interfaz `Comparator`, para comparar objetos `Cliente` basándose en el atributo `nombre`,

```
class ComparaNombres implements Comparator<Cliente> {
    public int compare(Cliente c1, Cliente c2) {
        return c1.nombre.compareTo(c2.nombre);
    }
}
```

A continuación creamos un objeto `ComparaNombres` y lo pasamos a la función donde se va a usar,

```
Comparator<Cliente> comp = new ComparaNombres();
Collections.sort(lista, comp); //la lista queda ordenada por nombres
```

Incluso podríamos prescindir de la variable `comp`, escribiendo una sola sentencia,

```
Collections.sort(lista, new ComparaNombres());
```

Segunda forma. Si vamos a usar el comparador una sola vez, no merece la pena implementar la clase comparadora explícitamente. Basta crear un objeto con una clase anónima,

```
Comparator<Cliente> comp = new Comparator<Cliente>() {
    public int compare(Cliente c1, Cliente c2) {
        return c1.nombre.compareTo(c2.nombre);
    }
}
Collections.sort(lista, comp);
```

O incluso, prescindiendo de la variable `comp`,

```
Collections.sort(lista, new Comparator<Cliente>() {
    public int compare(Cliente c1, Cliente c2) {
        return c1.nombre.compareTo(c2.nombre);
    }
});
```

Tercera forma (expresiones lambda). La sentencia anterior es la forma más corta de escribir el código para hacer la ordenación de la lista de clientes, pero en ella hay información redundante. Podríamos preguntarnos por qué es necesario especificar el nombre del método `compare()` cuando sabemos que la interfaz `Comparator` solo tiene ese método abstracto, que es el único que hay que implementar. Esa es la idea que subyace en la sintaxis de las expresiones *lambda*. Para implementar una interfaz funcional con una expresión lambda, basta escribir la lista de parámetros y el cuerpo de la función abstracta separados por una flecha (`->`). En nuestro ejemplo, implementar el comparador de nombres de clientes consiste en implementar el método `compare()` que, en forma de expresión lambda, quedaría así,

```
Comparator<Cliente> comp =
    (Cliente a, Cliente b) -> {return a.nombre.compareTo(b.nombre);};
```

Todo lo que está a la derecha del operador de asignación es la expresión lambda del método `compare()` de la interfaz `Comparator`, implementado para comparar nombres. El nombre del método no aparece, ya que Java lo infiere del lado izquierdo, donde aparece el de la interfaz `Comparator`, cuyo único método abstracto es `compare()`. Por tanto, Java sabe que en el lado derecho estamos implementando `compare()`. En realidad, también infiere el tipo de los parámetros de entrada (`Cliente` en nuestro caso) que, por tanto, se puede omitir del lado derecho, ya que en el lado izquierdo aparecen como el tipo genérico de la interfaz `Comparator`,

```
Comparator<Cliente> comp;
comp = (a,b) -> {return a.nombre.compareTo(b.nombre);};
```

En los casos como este, en que el cuerpo de la función es una sola sentencia, también podríamos prescindir de la orden `return`. En general, entre las llaves podemos escribir tantas sentencias como sean necesarias. También podemos prescindir de la variable `comp` y colocar la expresión lambda directamente en la lista de parámetros de `sort()`,

```
Collections.sort(lista,
    (a,b) -> {return a.nombre.compareTo(b.nombre);}
);
```

Java sabe que el segundo parámetro de `sort()` es un objeto `Comparator` e interpreta que el código que le pasamos corresponde al método `compare()`.

La sintaxis general de una expresión lambda consiste en,

```
(tipo1 param1, tipo2 param2,...) -> {Cuerpo de la expresión lambda};
```

Es decir,

- Una lista, entre paréntesis, de parámetros formales separados por comas. Los tipos de los parámetros se pueden omitir si Java los puede inferir del lado izquierdo en una operación de asignación. Cuando hay un solo parámetro de entrada, también se pueden omitir los paréntesis.
- Una flecha `->` (guión alto - seguido de `>`).
- El cuerpo de la función entre llaves, que puede consistir en una sentencia o un bloque de sentencias. Si es una única sentencia y no devuelve ningún valor, las llaves se pueden omitir. Si es una única sentencia y devuelve un valor, la orden `return` se puede omitir, ya que Java devuelve automáticamente el resultado de la sentencia.

Con una expresión lambda, más que una clase anónima, estamos creando un método anónimo. En realidad, cuando pasamos una de ellas a la función `sort()`, le estamos pasando un método como argumento. Hasta ahora, para hacer esto, había que crear un objeto que llevara el método encapsulado, ya que los argumentos en Java solo podían ser valores primitivos y objetos.

13.2. Algunas interfaces funcionales de la API

En vista de la simplicidad y la versatilidad de las interfaces funcionales, se ha definido un cierto número de ellas que, como `Comparator`, facilitan algunas operaciones frecuentes en las tareas del programador. Los ejemplos que vamos a poner aquí para ilustrarlas pueden parecer banales, pero estas interfaces son necesarias con los objetos de la interfaz `Stream`, que estudiaremos más adelante. Veamos las más útiles.

Predicate<T>: se emplea para comprobar una condición en un valor del tipo genérico `T`. Su método abstracto es,

```
boolean test(T valor)
```

que devuelve `true` si la condición se verifica para `valor` y `false` en caso contrario.

Por ejemplo, para comprobar si un `Integer` es positivo, podemos definir el predicado,

```
Predicate<Integer> esPositivo = x -> x > 0;
```

Entonces,

```
esPositivo.test(5)
```

devolverá `true`.

El método `test()`, es el único abstracto de la interfaz `Predicate`, pero junto a él hay otros tres métodos por defecto.

```
Predicate<T> negate()
```

devuelve un nuevo predicado que es la negación del predicado invocante. En nuestro caso,

```
esPositivo.negate()
```

nos devuelve un predicado que comprueba si un `Integer` es no positivo (menor o igual que 0),

```
Predicate<Integer> esNoPositivo = esPositivo.negate();
```

La expresión,

```
esNoPositivo.test(5)
```

devolverá `false`. Para hacer una sola comprobación, podríamos prescindir de la variable `esNoPositivo` y escribir directamente,

```
esPositivo.negate().test(5)
```

que dará el mismo resultado, `false`.

El segundo método por defecto es,

```
Predicate<T> and(Predicate<? super T> otro)
```


que devuelve un predicado que es la conjunción del predicado invocante y del que se pasa como parámetro, de modo que `test()` devolverá `true` cuando los dos predicados sean ciertos para el valor que se le pase como parámetro. El tipo genérico de `otro` debe ser igual o una superclase de `T` para garantizar que no va contener ni evaluar más atributos que los de la clase `T`. Veámoslo con un ejemplo. Para ello vamos a definir un segundo predicado,

```
Predicate<Integer> esPar = n -> n % 2 == 0;
```

que comprueba si un entero es par.

Si queremos saber si el entero 6 es par y positivo, escribimos,

```
Predicate<Integer> esPositivoYPar = esPar.and(esPositivo);
```

Entonces, la expresión,

```
esPositivoYPar.test(6)
```

devolverá `true`. También podemos poner,

```
esPar.and(esPositivo).test(6)
```

ya que 6 es par y positivo a la vez. En cambio,

```
esPar.and(esPositivo).test(-6)
```

y

```
esPar.and(esPositivo).test(7)
```

devuelven `false`, ya que `-6` es par, pero no positivo y `7` es positivo, pero impar.

El tercer método es la disyunción,

```
Predicate<T> or(Predicate<? Super T> otro)
```

que devuelve un predicado cuyo método `test()` devolverá `true` cuando, al menos, uno de los dos predicados (invocante y otro) sean ciertos para el valor que se le pase como parámetro,

```
Predicate<Integer> esPosotivoOPar = esPar.or(esPositivo);
esPositivoOPar.test(6) //true, par y positivo
esPositivoOPar.test(5) //true, es positivo
esPositivoOPar.test(-2) //true, es par
esPositivoOPar.test(-3) //false, ni par ni positivo
```

Function<T, V>: a partir de un argumento de tipo `T`, devuelve un resultado de tipo `V`. Su única función abstracta es,

```
V apply(T x)
```

Por ejemplo, si queremos definir una función que calcula el cuadrado de un valor real (de tipo `Double`),

```
Function<Double, Double> cuadrado = x -> x*x;
System.out.println(cuadrado.apply(2.0)); //mostrará 4.0 por consola
```

Además, **Function** añade tres funciones por defecto, que sirven para componer funciones. No las vamos a estudiar aquí por salirse del propósito de este libro.

Consumer<T>: sirve para realizar una tarea a partir de un argumento de entrada. Su método abstracto,

```
void accept(T t)
```

recibe un valor de una clase T, con el que hace operaciones sin devolver nada. Por ejemplo, si queremos mostrar por pantalla un saludo a distintos clientes,

```
Consumer<Cliente> saludoClie = c -> System.out.println("Hola, " +
                                                    c.nombre);
```

El método **accept()**, recibirá como argumento un objeto **Cliente** y, a partir de él creará un mensaje de saludo con su nombre,

```
Cliente clie=new Cliente("123", "Jorge", 20);
saludoClie.accept(clie); //se mostrará "Hola, Jorge"
```

La API proporciona otras interfaces funcionales importantes que iremos viendo.

13.2.1. Referencias a métodos

A partir de la versión 8 de Java, es posible trabajar con referencias a métodos ya definidos en alguna clase. Cuando hemos implementado la interfaz **Function**, la función a aplicar la hemos pasado como expresión lambda, es decir, como método anónimo. Pero cuando la función ya está implementada en un método de alguna clase, como ocurre con **Math.sqrt()**, tenemos una forma aún más corta de escribirla: como una referencia al método. Una referencia a **Math.sqrt()** se escribe,

```
Math::sqrt
```

y se puede colocar en lugar de la expresión lambda,

```
x -> Math.sqrt(x)
```

Entonces, para calcular raíces cuadradas de valores **Double**, podemos implementar,

```
Function<Double, Double> raiz = Math::sqrt;
```

Para calcular una raíz cuadrada, pondríamos,

```
Double x = raiz.apply(9.); // devolvería 3.0
```

Las referencias a métodos se escriben poniendo el nombre de la clase, seguido de **::** y del nombre del método (sin paréntesis ni lista de argumentos) cuando este es estático. Si es no estático, en vez del nombre de la clase pondremos una referencia a un objeto de la

clase donde está definido el método. En nuestro caso, hemos escrito una referencia al método estático `sqr()`, definido en la clase `Math` de la API².

También se pueden usar referencias a métodos constructores. En este caso, la sintaxis es un poco especial. Como el constructor tiene el mismo nombre que la clase, cabría esperar algo así como, `Cliente::Cliente`, pero en realidad es `Cliente::new`. Como ejemplo, podríamos implementar la interfaz `Function` para construir objetos de la clase `Saludo`,

```
class Saludo {
    String nombre;
    Saludo(String nombre) {
        this.nombre = nombre;
    }
    public String toString() {
        return "Hola, " + nombre;
    }
}
```

El método `apply()` de la interfaz `Function`, recibirá una cadena con el nombre, y deberá construir y devolver un objeto `Saludo` con ese nombre,

```
Function<String, Saludo> construyeSaludo = Saludo::new;
Saludo s = construyeSaludo.apply("Julia");
System.out.println(s); //¡Hola Julia!
```

A la hora de ejecutar `apply()`, Java busca el constructor en la clase `Saludo` y lo ejecuta pasando el valor "Julia" como parámetro.

13.3. Interfaz Stream

Los objetos de las clases que implementan la interfaz `Stream` son sucesiones de objetos sobre los que se puede realizar una serie de operaciones que pueden ir encadenadas hasta dar un resultado final. Dichas operaciones pueden ser de dos tipos,

Intermedias: dan como resultado un nuevo `Stream` al que se le pueden seguir aplicando nuevas operaciones.

Terminales: dan un resultado final, numérico o de otro tipo, pero no un `Stream`.

La idea es crear, a partir de una colección o una tabla, o bien explícitamente, un `Stream` al que se aplican operaciones intermedias encadenadas (es lo que se conoce como una *tubería* o *pipeline*), obteniendo un resultado final por medio de una operación terminal. La ventaja de crear el `Stream` es que dispone de muchas más operaciones para procesar sus datos que las colecciones o las tablas.

²A primera vista puede parecer extraña la idea de una referencia a una función. Pero, cuando el sistema va a ejecutar un programa, antes carga su código en la memoria, de donde luego va leyendo y ejecutando sentencia a sentencia. Por tanto, un método que forma parte de una aplicación que se va a ejecutar, ocupa un cierto bloque de memoria. Cuando pasamos como parámetro la referencia de un método, lo que estamos pasando es la referencia del bloque donde está su código.

Los **Stream** son objetos que implementan la interfaz **Stream**. Por tanto, la clase **Stream** no existe y los objetos **Stream** no se pueden crear con un constructor, sino llamando a alguna de las funciones implementadas para ello.

Se dice que las operaciones sobre **Stream** son agregadas y se inspiran en las operaciones globales de las colecciones, ya que operan sobre la totalidad del **Stream**. Muchas de ellas hacen uso de interfaces funcionales de la API, de las que hemos visto algunas ya. De hecho, los **Stream** se han diseñado para trabajar con expresiones lambda.

13.3.1. Formas de crear un Stream

Hay diversas formas de obtener un **Stream** inicial, es decir, que no proceda de otro **Stream**. Nosotros vamos a ver cuatro.

- A partir de una colección: llamando al método `stream()`, definido en las clases de tipo `Collection`,

```
Stream<T> nombreStream = nombreColeccion.stream();
```

- A partir de una tabla: llamando al método `of()`, de la clase **Stream**, con la tabla como parámetro,

```
Stream<T> nombreStream = Stream.of(T[] tabla);
```

- A partir de una tabla: usando el método `stream()`, de la clase **Arrays**, con la tabla como parámetro

```
Stream<T> nombreStream = Arrays.stream(T[] tabla);
```

- Inicializándolo directamente: también con el método `of()` de **Stream**, pero pasándole como lista de parámetros los valores que lo inicializan,

```
Stream<T> nombreStream = Stream.of(T val1, T val2,...)
```

Todos ellos los iremos usando a lo largo del capítulo.

Supongamos que queremos trabajar con los datos de una lista. Para verlo con un caso práctico, vamos empezar creando una lista de cadenas,

```
List<String> lista = new ArrayList<>();
lista.add("dado");
lista.add("arte");
lista.add("bola");
lista.add("asa");
lista.add("buzo");
lista.add("coche");
lista.add("barco");
lista.add("duna");
```

A partir de ella, vamos a crear un **Stream** de cadenas por el primer método,

```
Stream<String> streamCad = lista.stream();
```

El **Stream** `streamCad` contiene una copia de todos los datos de la lista, no una referencia a los originales. Por tanto, los cambios que se hagan en el **Stream** no se van a reflejar en la lista original, que permanecerá intacta.

Una de las cosas que podemos hacer con los elementos de un **Stream** es filtrarlos. Para ello se usa el método,

```
Stream<T> filter(Predicate<? Super T> pred)
```

Invocado desde el **Stream** original, se le pasa un predicado que se aplicará a todos los elementos del **Stream**. Solo aquellos que devuelvan `true` formarán parte del nuevo **Stream** devuelto por el método. Naturalmente, `filter()` es un método intermedio, ya que devuelve un nuevo **Stream**, susceptible de llamar a nuevos métodos para producir nuevas transformaciones. Por ejemplo, si queremos obtener, a partir de `streamCad`, un nuevo **Stream** con los elementos que empiezan por “a”, crearemos el predicado,

```
Predicate<String> empiezaPorA = s -> s.startsWith("a");
```

donde se ha invocado al método `startsWith()` de la clase **String**. Este predicado se le pasa como argumento al método `filter()`, invocada por `streamCad`, y devuelve un nuevo **Stream** con los elementos filtrados,

```
Stream<String> streamA = streamCad.filter(empiezaPorA);
```

Ahora `streamA` contiene aquellos elementos del **Stream** original que empiezan por «a». En realidad, lo más común es que el filtro solo se tenga que aplicar una vez. Por tanto, generalmente no merece la pena crear una variable para el predicado. Lo normal es pasarlo como argumento directamente, en forma de expresión lambda, al método `filter()`, prescindiendo de la variable `empiezaPor`,

```
Stream<String> streamA = streamCad.filter(s -> s.startsWith("a"));
```

Si queremos ver los resultados obtenidos hasta ahora, tendremos que aplicar una nueva operación, ya que no disponemos de una función `toString()` para **Stream**. Es decir, no podemos escribir,

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

que mostraría la referencia del objeto, algo así como,

```
java.util.stream.ReferencePipeline$2@12a8025c
```

Si queremos que todos los elementos de un **Stream** se muestren por pantalla, deberemos hacer que para cada uno de ellos se ejecute el método,

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

Siempre que queramos que se ejecute una determinada acción «para cada» elemento de un **Stream**, usaremos el método,

```
void forEach(Consumer<? Super T> tarea)
```

donde `T` es el tipo genérico del **Stream** que invoca el método. El parámetro `tarea` es un **Consumer** que lleva encapsulado el método `accept()` que se tiene que ejecutar para todos y cada uno de los elementos del **Stream**. Como puede verse, `forEach()` no devuelve otro

Stream (de hecho, no devuelve nada), por lo cual es un método terminal. Si queremos mostrar por pantalla todos los elementos de **streamA**, llamamos a **forEach()** pasándole como argumento un **Consumer** que muestre cadenas por pantalla,

```
Consumer<String> mostrar = s -> System.out.println(s);
streamA.forEach(mostrar);
```

o más brevemente,

```
streamA.forEach(s -> System.out.println(s)); //se mostrará "arte" y "asa"
```

o incluso, usando referencias a métodos,

```
streamA.forEach(System.out::println);
```

Una cosa *muy importante* a tener en cuenta con los **Stream** es que no son reusables, es decir, cada operación intermedia sobre un **Stream**, nos devuelve un **Stream** transformado, pero el **Stream** original se pierde. Por ejemplo, si después de obtener **streamA**, con los elementos filtrados a partir de **streamCad**, intentamos volver a utilizar este último para filtrar los elementos que empiezan por “b”,

```
streamCad.filter(s -> s.startsWith("b")).forEach(System.out::println);
```

saltará la excepción `java.lang.IllegalStateException`, con la descripción,

```
stream has already been operated upon or closed
```

es decir, ya se ha operado antes sobre **streamCad** y no se puede volver a operar. Podemos aplicar un nuevo método al **Stream** devuelto, formando una tubería (ver el epígrafe siguiente), pero no podemos volver a usar el **Stream** original. Todo esto deberá tenerlo en cuenta el lector a la hora de probar las distintas funciones que estamos viendo, ya que un **Stream** usado con una función no puede ser reutilizado para probar otra. Si queremos hacerlo, deberemos volver a crearla desde el principio a partir de la colección o la tabla original.

13.3.2. Tuberías o *pipelines*

Si de lo que se trataba era de mostrar por pantalla los elementos que empiezan por «a», podríamos haber prescindido de la variable intermedia **streamA** y haber encadenado las dos operaciones para formar lo que se llama una tubería, que no es más que un **Stream** fuente (creado a partir de una colección, de una tabla o por otro medio) al que se aplica una serie de operaciones intermedias encadenadas acabando, generalmente, con una operación terminal. En el ejemplo anterior, podíamos haber escrito,

```
lista.stream().filter(s -> s.startsWith("a")).forEach(System.out::println);
```

Las tuberías, a menudo, son largas y no caben en una sola línea del editor. Además, la lectura puede ser incómoda. Por eso es costumbre poner cada operación en una línea,

```
lista.stream()
    .filter(s -> s.startsWith("a"))
    .forEach(System.out::println);
```

El **Stream** del ejemplo lo obtuvimos a partir de una lista. También podemos obtener un **Stream**, a partir de una tabla, con el método estático `of()` de la interfaz **Stream**. Para ver un ejemplo, vamos a crear una tabla de clientes,

```
Cliente[] tClie={new Cliente("111", "Marta", 20),
                 new Cliente("115", "Jorge", 21),
                 new Cliente("112", "Carlos", 18),
                 new Cliente("211", "Ana", 19)};
```

y, a partir de ella, obtendremos un **Stream** por cualquiera de los métodos,

```
Stream<Cliente> streamClie = Stream.of(tClie);
```

o bien,

```
Stream<Cliente> streamClie = Arrays.stream(tClie);
```

Como los **Stream** no son reutilizables, tiene poco sentido crear la variable `streamClie`. Lo habitual es escribir las tuberías completas, incluyendo la lista o la tabla iniciales cada vez. Así lo vamos a hacer con las nuevas operaciones de agregación que vamos a estudiar.

Una muy importante es ordenar un **Stream**, que se lleva a cabo por medio del método,

```
Stream<T> sorted()
```

que devuelve un nuevo **Stream** con los elementos ordenados según su orden natural,

```
Arrays.stream(tClie)
    .sorted()
    .forEach(System.out::println);
```

mostrará los clientes ordenados por DNI.

El método `sorted()` está sobrecargado y puede admitir como parámetro un comparador para cambiar el criterio de ordenación de los elementos. Por ejemplo, si queremos que los clientes se ordenen por nombre, definimos el comparador,

```
Comparator<Cliente> comp = (x, y) -> x.nombre.compareTo(y.nombre);
```

con lo cual,

```
Arrays.stream(tClie)
    .sorted(comp)
    .forEach(System.out::println);
```

o bien, prescindiendo de la variable `comp`,

```
Arrays.stream(tClie)
    .sorted((x,y) -> x.nombre.compareTo(y.nombre))
    .forEach(System.out::println);
```

muestra los clientes ordenados por nombres.

A partir de un **Stream** podemos obtener otro cuyos elementos se corresponden uno a uno con los del **Stream** original, pero con una determinada transformación. Por ejemplo, puede interesarnos un **Stream** con los DNI de los clientes, en el mismo orden en que aparecen en el **Stream** original. Esa tarea la lleva a cabo el método,

```
Stream<V> map(Function<? super T, ? extends V> mapper)
```

A pesar de lo aparatoso de la expresión, es fácil de usar. El método recibe como parámetro una función, con la cual transforma los elementos del **Stream** original del tipo **T** y devuelve un **Stream** con los elementos transformados, de tipo **V**. En el ejemplo propuesto, necesitamos una función (en realidad, el método abstracto `apply()`, que ya vimos) que reciba un objeto **Cliente** y devuelva su `dni`. La expresión lambda correspondiente será,

```
Function<Cliente, String> aDni = c -> c.dni;
```

que transforma un objeto `c` del tipo **Cliente** en su `dni`, de tipo **String**.

Por tanto, prescindiendo de la variable `aDni`, podemos escribir,

```
Arrays.stream(tClie)
    .map(c -> c.dni)
    .forEach(System.out::println);
```

que mostrará, para todos los elementos de la tabla de clientes, sus DNI.

El método terminal,

```
long count()
```

nos devuelve el número de elementos de un **Stream**. Por ejemplo,

```
long n = Arrays.stream(tClie)
    .sorted((c1,c2) -> c1.nombre.compareTo(c2.nombre))
    .filter(c -> c.edad < 20)
    .count();
```

devuelve 2, el número de clientes de menos de 20 años de edad.

Vamos a crear ahora un **Stream** de enteros inicializándolo de forma explícita,

```
Stream<Integer> streamEnteros = Stream.of(4, 3, 7, 1, 0, 8, 9, 3, 5
                                         4, 2, 1, 4, 6, 8, 1, 0, 2, 3);
```

Una de las cosas que podemos hacer es eliminar los elementos repetidos. Para ello existe el método,

```
Stream<T> distinct()
```

que devuelve un nuevo **Stream** sin repeticiones. Con nuestro **Stream**,

```
Stream.of(4, 3, 7, 1, 0, 8, 9, 3, 5, 4, 2, 1, 4, 6, 8, 1, 0, 2, 3)
    .distinct()
    .forEach(x -> System.out.print(x + " "));
```

mostrará por pantalla,

```
4 3 7 1 0 8 9 5 2 6
```

También podemos concatenar dos **Stream**,

```
static Stream<T> concat(Stream<? extends T> prim, Stream<? extends T> seg)
```

devuelve un nuevo **Stream** con los elementos del segundo a continuación de los del primero. El método es estático y se debe invocar desde la interfaz. Por ejemplo, si creamos un nuevo **Stream** de enteros, `streamNuevo`, y lo concatenamos con `streamEnteros` sin repeticiones,


```
Stream<Integer> streamNuevo = Stream.of(-1, -6, -3, -3);
Stream.concat(streamEnteros, streamNuevo)
    .distinct()
    .forEach(x -> System.out.print( x + " "));
```

obtendremos por pantalla,

```
4 3 7 1 0 8 9 5 2 6 -1 -6 -3
```

A menudo nos interesará crear una tabla con los elementos de un `Stream`. Para ello disponemos del método,

```
Object[] toArray()
```

Por ejemplo, si queremos una tabla con los números pares sin repetir de `streamNuevo`,

```
Object[] tObject = Stream.of(-1, -6, -3, -3)
    .distinct()
    .filter(x -> x % 2 == 0)
    .toArray();
```

Para transformar la tabla de tipo `Object[]` en una de tipo `Integer[]`, podemos usar el método `copyOf()` de la clase `Arrays`, sobrecargada con una versión que admite como último parámetro la clase de la tabla destino,

```
Integer[] tInt = Arrays.copyOf(tObject, tObject.length, Integer[].class);
```

También podemos agrupar los elementos de un `Stream` en una colección, un mapa o una cadena. O hacer estadísticas de sus datos. Todo esto se consigue con el método `collect()`. Es tan rico como complejo y no vamos a estudiarlo aquí a fondo. Solo vamos a ver algunas aplicaciones sencillas que resultan muy útiles. En todos los casos se le pasa como parámetro un objeto `Collector`, que se obtiene a partir de distintos métodos de la clase `Collectors`. Por ejemplo, si queremos una lista con los valores impares de un `Stream` de números enteros, pasamos como argumento el colector devuelto por `Collectors.toList()`,

```
List<Integer> listaImpares = Stream.of(2, 5, 1, 4, -6, -3, -3)
    .filter(x -> x % 2 != 0)
    .collect(Collectors.toList());
```

También podemos extraer un conjunto, en vez de una lista,

```
Set<Integer> conjuntoImpares = Stream.of(5, 1, 2, 6, 3, 9, 4, 1, 7, 3, 5)
    .filter(x -> x % 2 != 0)
    .collect(Collectors.toSet());
```

con lo cual se eliminan automáticamente las repeticiones, resultando,

```
[1, 3, 5, 7, 9]
```

Se puede escoger una implementación concreta de lista o de conjunto. Por ejemplo, si queremos mantener un orden, usaremos `Collectors.toCollection(TreeSet::new)`,

```
Set<Integer> conjuntoImpares = Stream.of(5, 1, 2, 6, 3, 9, 4, 1, 7, 3, 5)
    .filter(x -> x % 2 != 0)
    .collect(Collectors.toCollection(TreeSet::new));
```

Cualquier elemento que se inserte en `conjuntoImpares`, lo hará manteniendo en orden natural,

```
conjuntoImpares.add(-5);
conjuntoImpares.add(13);
System.out.println(conjuntoImpares);
```

Se mostrará,

```
[-5, 1, 3, 5, 7, 9, 13]
```

Volvamos al `Stream` de clientes. Si queremos crear un mapa de los DNI (claves) sobre los nombres (valores) de los clientes, usaremos `Collectors.toMap()` y deberemos especificar qué atributo es clave y cuál es valor, por ese orden,

```
Map<String, String> mapaClientes = Stream.of(tClie)
    .collect(Collectors.toMap(c -> c.dni, c -> c.nombre));
```

obteniéndose el mapa,

```
{111=Marta, 211=Ana, 112=Carlos, 115=Jorge}
```

Con `Collectors.averagingInt()` podemos calcular el promedio de las edades,

```
double edadMedia = Stream.of(tClie)
    .collect(Collectors.averagingInt(c -> c.edad));
```

o una estadística general de las edades,

```
IntSummaryStatistics sumarioEdad =
    streamClie.collect(Collectors.summarizingInt(c -> c.edad));
```

donde `IntSummaryStatistics` es una clase capaz de calcular diversos parámetros estadísticos. Podemos ejecutar,

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

y obtenemos por pantalla un sumario de dichos parámetros,

```
IntSummaryStatistics{count=4, sum=78, min=18, average=19.500000, max=21}
```

Ejercicios de Stream

- 13.1.** Crear una lista con 40 números enteros aleatorios entre -10 y 10. A partir de ella crear dos `Stream`, uno con los números positivos, sin repetir y otro con los negativos. Mostrar por pantalla el número de elementos de cada `Stream`. Crear otro `Stream` para contar los ceros.

```
import java.util.ArrayList;
import java.util.List;

/*
Generamos aleatoriamente los números entre -10 y 10. Como el rango
incluye 21 números, multiplicamos por 21. Como empieza en -10, restamos 10. */
public class Main {
```

```

public static void main(String[] args) {
    List<Integer> l = new ArrayList<>();
    for (int i = 0; i < 40; i++) {
        Integer n = (int) (Math.random() * 21) - 10;
        l.add(n);
    }

    //números positivos
    long numPositivos = l.stream()
        .filter(n -> n > 0)
        .distinct()//eliminamos repetidos
        .count();
    System.out.println("Positivos:" + numPositivos);

    //números negativos
    long numNegativos = l.stream()
        .filter(n -> n < 0)
        .distinct()
        .count();
    System.out.println("Negativos:" + numNegativos);

    //contamos los ceros
    long numCeros = l.stream()
        .filter(n -> n == 0)
        .count();
    System.out.println("Ceros: " + numCeros);
}
}

```

13.2. Repetir el ejercicio anterior, pero en vez de mostrar por pantalla el número de elementos, mostrar el contenido de los dos **Stream**:

- Ordenados de mayor a menor.
- Ordenados de menor a mayor.

```

import java.util.ArrayList;
import java.util.List;

/*
Para ordenar en sentido decreciente, al método sorted() le hemos pasado
como comparador una expresión lambda que invierte el orden natural
de los enteros. */
public class Main {

    public static void main(String[] args) {
        List<Integer> l = new ArrayList<>();

        for (int i = 0; i < 40; i++) {
            Integer n = (int) (Math.random() * 21) - 10;
            l.add(n);
        }

        //números positivos orden creciente
        System.out.println("Positivos crecientes:");
        l.stream()
            .filter(n -> n > 0)
            .distinct()//eliminamos repetidos
            .sorted()
            .forEach(System.out::println);

        //números negativos orden creciente
        System.out.println("Negativos crecientes:");
    }
}

```

```
l.stream()
    .filter(n -> n < 0)
    .distinct()
    .sorted()
    .forEach(System.out::println);

//números positivos orden decreciente
System.out.println("Positivos decrecientes:");
l.stream()
    .filter(n -> n > 0)
    .distinct()//eliminamos repetidos
    .sorted((x, y) -> y - x)
    .forEach(System.out::println);

//números negativos orden decreciente
System.out.println("Negativos decrecientes:");
l.stream()
    .filter(n -> n < 0)
    .distinct()
    .sorted((x, y) -> y - x)
    .forEach(System.out::println);
}
}
```

- 13.3.** A partir de una lista con los enteros del 1 al 100, crear un **Stream** con los múltiplos de 7. Mostrarlos por pantalla.

```
import java.util.ArrayList;
import java.util.List;

/* Filtramos los múltiplos de 7. */
public class Main {

    public static void main(String[] args) {
        List<Integer> l = new ArrayList<>();

        for (int i = 1; i <= 100; i++) {
            l.add(i);
        }

        l.stream()
            .filter(n -> n % 7 == 0)
            .forEach(System.out::println);
    }
}
```

- 13.4.** Fusionar dos listas, cada una con 20 enteros aleatorios entre 1 y 100, en un **Stream** ordenado sin repeticiones. Mostrar los elementos del **Stream**.

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

/*
 * Para unir los dos Stream, usamos el método estático concat() de la interfaz Stream. */
public class Main {

    public static void main(String[] args) {
        List<Integer> l1 = new ArrayList<>();
        List<Integer> l2 = new ArrayList<>();
```

```

    for (int i = 0; i < 20; i++) {
        l1.add(((int) (Math.random() * 100) + 1));
        l2.add(((int) (Math.random() * 100) + 1));
    }
    System.out.print("[");
    //pasamos como argumentos los dos Stream creados a partir de l1 y l2
    Stream.concat(l1.stream(), l2.stream()).distinct()
        .distinct()//eliminamos repetidos
        .sorted()//ordenamos en sentido creciente
        .forEach(x -> System.out.print(x + " "));
    System.out.println("]");
}
}

```

13.5. A partir de una cadena con palabras separadas por espacios introducida por teclado, construir una tabla con las palabras. A partir de ella, crear un **Stream** con las palabras ordenadas por orden alfabético. Mostrarlas por pantalla.

```

import java.util.Scanner;
import java.util.stream.Stream;

/* Creamos un Stream a partir de una tabla con las palabras de la frase que
 * previamente hemos separado con la función split() de la clase String. */
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introducir frase: ");
        String frase = sc.nextLine();

        Stream.of(frase.split(" "))
            .sorted()
            .forEach(System.out::println);
    }
}

```

13.6. Implementar la clase **Socio** con los atributos, DNI, nombre, fecha de nacimiento, fecha de alta (ambos de tipo **LocalDate**), cuota y el número de familiares del socio.

Además de un constructor, implementar los métodos **equals()**, **compareTo()** (basados en el DNI) y **toString()**. Crear una tabla con 5 socios. A partir de ella, crear un **Stream** con los socios,

- a) Ordenados por DNI.
- b) Con una cuota mayor de 100 €.
- c) Cuyo nombre empieza por 'A'.

En todos los casos mostrar por pantalla los elementos del **Stream**.

Clase **Socio**

```

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

/* Para las fechas usaremos la clase LocalDate (ver anexo A) con sus formateadores,
 * que usará tanto en el constructor como en la salida por pantalla. */

```

```
public class Socio implements Comparable {
    String dni;
    String nombre;
    LocalDate fechaNacimiento;
    LocalDate fechaAlta;
    double cuota;
    int numeroFamiliars;

    public Socio(String dni, String nombre, String fechaNacimiento,
                  String fechaAlta, double cuota, int numeroFamiliars) {
        //formateador para interpretar las cadenas de los parámetros de entrada
        //para las dos fechas
        DateTimeFormatter f = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        this.dni = dni;
        this.nombre = nombre;
        this.fechaNacimiento = LocalDate.parse(fechaNacimiento, f);
        this.fechaAlta = LocalDate.parse(fechaAlta, f);
        this.cuota = cuota;
        this.numeroFamiliars = numeroFamiliars;
    }

    //equals() de Socio se basa en el equals() de los dni como cadenas
    @Override
    public boolean equals(Object otro) {
        return dni.equals(((Socio) otro).dni);
    }

    //compareTo() de Socio se basa en la comparación entre los dni como cadenas
    @Override
    public int compareTo(Object o) {
        return dni.compareTo(((Socio) o).dni);
    }

    @Override
    public String toString() {
        //definimos el formateador de fechas para la salida en forma de cadena
        DateTimeFormatter f = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        return "Dni: " + dni + " Nombre: " + nombre + " Fecha nac.: "
            + f.format(fechaNacimiento) + " Alta: "
            + f.format(fechaAlta) + " Cuota: " + cuota
            + " Familiares: " + numeroFamiliars;
    }
}
```

Programa principal

```
import java.util.Arrays;
/*
    Crearemos una tabla de socios y, a partir de ella, generaremos Streams con los
    filtros correspondientes. */
public class Main {

    public static void main(String[] args) {
        Socio[] socios = {
            new Socio("323", "Aurora", "03/12/1980", "10/10/2010", 60, 2),
            new Socio("123", "Justino", "12/02/1990", "11/11/2014", 40, 1),
            new Socio("452", "Antonio", "10/11/1967", "03/03/2000", 150, 5),
            new Socio("222", "Rosa", "21/06/1975", "23/05/1998", 120, 3),
            new Socio("132", "Ricardo", "18/09/1970", "15/08/2009", 130, 4)
        };

        System.out.println("Socios por orden de DNI:");
        Arrays.stream(socios)
            .sorted()//ordenamos por orden natural de DNI
            .forEach(System.out::println);
    }
}
```

```

        System.out.println("\n\nSocios con cuota mayor que 100 euros:");
        Arrays.stream(socios)
            .filter(s -> s.cuota > 100) //filtramos los que pagan más de 100
            .forEach(System.out::println);

        System.out.println("\n\nSocios con nombre que empieza por A: ");
        Arrays.stream(socios)
            .filter(s -> s.nombre.startsWith("A")) //nombres que empiezan por A
            .forEach(System.out::println);
    }
}

```

13.7. Añadir a la clase Socio los métodos,

int edad(): que calcula la edad del socio en años, a partir de la fecha de nacimiento y de la fecha actual.

int antigüedad(): que calcula la antigüedad del socio en meses completos.

Crear (y mostrar) un **Stream** con los socios ordenados por antigüedad, y otro con los socios ordenados por edad.

Clase Socio

```

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;

/* Para las fechas usaremos la clase LocalDate (ver anexo A) con sus formateadores,
 * que usará tanto en el constructor como en la salida por pantalla. Asimismo
 * usamos el método until() para el cálculo de periodos. */
public class Socio implements Comparable {
    ... //añadimos los métodos edad() y antigüedad()

    //utilizamos la función until() para calcular los periodos entre las fechas
    //de nacimiento y alta y el momento presente
    int edad() {
        return (int)fechaNacimiento.until(LocalDate.now(), ChronoUnit.YEARS); //en años
    }

    int antigüedad() {
        return (int)fechaAlta.until(LocalDate.now(), ChronoUnit.MONTHS); //en meses
    }
}

```

Programa principal

```

import java.util.Arrays;
/*
 * Crearemos una tabla de socios y, a partir de ella generaremos Streams ordenadas
 * por los criterios antigüedad y edad. El cast (int) es necesario porque el
 * método compare() devuelve un int, mientras que el método until() devuelve long */
public class Main {

    public static void main(String[] args) {
        Socio[] socios = {
            new Socio("323", "Aurora", "03/12/1980", "10/10/2010", 60, 2),
            ... //evitamos repetir líneas de código
            new Socio("132", "Ricardo", "18/09/1970", "15/08/2009", 130, 4)
        };
    }
}

```

```
System.out.println("Ordenados por antigüedad creciente:");
Arrays.stream(socios)
    .sorted((s1, s2) -> (int) (s1.antigüedad() - s2.antigüedad()))
    .forEach(System.out::println);

System.out.println("\nOrdenados por edad creciente: ");
Arrays.stream(socios)
    .sorted((s1, s2) -> (int) (s1.edad() - s2.edad()))
    .forEach(System.out::println);
}
}
```

13.8. A partir de una tabla de socios, crear un **Stream** con los dni ordenados.

```
import java.util.Arrays;

/*
 * A partir del Stream de los socios, hacemos un mapping o aplicación haciendo
 * corresponder a cada socio su DNI, generando así un Stream de cadenas. */
public class Main {

    public static void main(String[] args) {
        Socio[] socios = {
            new Socio("323", "Aurora", "03/12/1980", "10/10/2010", 60, 2),
            new Socio("123", "Justino", "12/02/1990", "11/11/2014", 40, 1),
            new Socio("452", "Antonio", "10/11/1967", "03/03/2000", 150, 5),
            new Socio("222", "Rosa", "21/06/1975", "23/05/1998", 120, 3),
            new Socio("132", "Ricardo", "18/09/1970", "15/08/2009", 130, 4)
        };

        Arrays.stream(socios)
            .map(s -> s.dni)//aplicación de socios en dni
            .sorted()//ordenamos los dni
            .forEach(System.out::println);
    }
}
```

13.9. Repetir el ejercicio anterior pero, en vez de mostrar los elementos del **Stream** obtener, a partir de él, una tabla de **String** con los DNI.

```
import java.util.Arrays;

/*
 * El método toArray() convierte los elementos de un Stream en una tabla de Object
 * Por tanto hay que hacer una conversión de Object[] a String[]. Para esto no vale
 * un cast (String[]) como hacíamos con los flujos para archivos, sino que tenemos
 * que usar la versión apropiada de Arrays.copyOf(), donde se pasa, como último
 * parámetro, la clase de la tabla destino, String[].class. */
public class Main {

    public static void main(String[] args) {
        Socio[] socios = {
            new Socio("323", "Aurora", "03/12/1980", "10/10/2010", 60, 2),
            new Socio("123", "Justino", "12/02/1990", "11/11/2014", 40, 1),
            new Socio("452", "Antonio", "10/11/1967", "03/03/2000", 150, 5),
            new Socio("222", "Rosa", "21/06/1975", "23/05/1998", 120, 3),
            new Socio("132", "Ricardo", "18/09/1970", "15/08/2009", 130, 4)
        };

        Object[] tablaObjects = Arrays.stream(socios)
            .map(s -> s.dni)//aplicación de socios en dni
```



```

        .sorted()//ordenamos los dni
        .toArray();//tabla con los dni como Object

    //Ahora tenemos que convertir Object[] a String[]
    String[] tablaDni;
    tablaDni = Arrays.copyOf(tablaObjects, tablaObjects.length, String[].class);

    System.out.println(Arrays.toString(tablaDni));
}
}

```

13.10. Realizar el ejercicio anterior, pero obteniendo una lista, en vez de una tabla.

```

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

/*
 * Entre las muchas utilidades del método collect() está la de convertir un Stream
 * en una lista usando el colector Collectors.toList() como argumento. */
public class Main {

    public static void main(String[] args) {
        Socio[] socios = {
            new Socio("323", "Aurora", "03/12/1980", "10/10/2010", 60, 2),
            new Socio("123", "Justino", "12/02/1990", "11/11/2014", 40, 1),
            new Socio("452", "Antonio", "10/11/1967", "03/03/2000", 150, 5),
            new Socio("222", "Rosa", "21/06/1975", "23/05/1998", 120, 3),
            new Socio("132", "Ricardo", "18/09/1970", "15/08/2009", 130, 4)
        };

        List<String> l
            = Arrays.stream(socios)
                .map(s -> s.dni)//aplicación de socios en dni
                .sorted()//ordenamos los dni
                .collect(Collectors.toList());//lista de cadenas con los dni

        System.out.println(l);
    }
}

```

13.11. Volver a obtener un **Stream** con los DNI de los socios ordenados, utilizando un objeto **Map**, donde las claves sean los DNI y los valores los nombres de los socios.

```

import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

/*
 * El método collect(), con Collectors.toMap(), genera un mapa con las claves como
 * primer argumento del método y los valores como segundo argumento. */
public class Main {

    public static void main(String[] args) {
        Socio[] socios = {
            new Socio("323", "Aurora", "03/12/1980", "10/10/2010", 60, 2),
            new Socio("123", "Justino", "12/02/1990", "11/11/2014", 40, 1),
            new Socio("452", "Antonio", "10/11/1967", "03/03/2000", 150, 5),
            new Socio("222", "Rosa", "21/06/1975", "23/05/1998", 120, 3),
            new Socio("132", "Ricardo", "18/09/1970", "15/08/2009", 130, 4)
        };
    }
}

```

```
Map<String, String> m
    = Arrays.stream(socios)
        .collect(Collectors.toMap(s -> s.dni, s -> s.nombre)); //mapa (dni, nombre)

    System.out.println(m);
}
}
```

13.12. Calcular, usando un **Stream**, el número medio de familiares por socio.

```
import java.util.Arrays;
import java.util.stream.Collectors;

/*
Una de las utilidades de collect() es el cálculo de promedios de atributos
numéricos con Collectors.averagingInt(), averagingLong() o averagingDouble()
*/
public class Main {

    public static void main(String[] args) {
        Socio[] socios = {
            new Socio("323", "Aurora", "03/12/1980", "10/10/2010", 60, 2),
            new Socio("123", "Justino", "12/02/1990", "11/11/2014", 40, 1),
            new Socio("452", "Antonio", "10/11/1967", "03/03/2000", 150, 5),
            new Socio("222", "Rosa", "21/06/1975", "23/05/1998", 120, 3),
            new Socio("132", "Ricardo", "18/09/1970", "15/08/2009", 130, 4)
        };

        double mediaFamiliares
            = Arrays.stream(socios)
                .collect(Collectors.averagingInt(s -> s.numeroFamiliares));

        System.out.println(mediaFamiliares);
    }
}
```

Ejercicios propuestos

- 13.1.** Escribir los números primos menores que 100 por medio de la criba de Eratóstenes. A partir de una lista con los números del 2 al 100, eliminamos los múltiplos de los números primos menores que 10: 2, 3, 5 y 7. Utilizar **Stream** para cribar los números.
- 13.2.** Con la tabla de socios del Ejercicio resuleto 13.6, crear y mostrar un **Stream** con los socios ordenados por nombre.

Anexo A

Utilidades

A.1. Números aleatorios

Hay más de una forma de generar números aleatorios en Java. En este libro usamos el método `random()` de la clase `Math`. Cuando se llama a este método por primera vez en un programa, devuelve un `double` aleatorio. En realidad, son pseudoaleatorios, porque se generan con una fórmula matemática que, a partir de un número inicial llamado *semilla*, va generando una serie de números. A igual semilla, igual serie de números. Los números generados por `random()` son números decimales mayores o iguales que 0 y menores que 1, y toman como semilla un número obtenido del reloj del sistema. En llamadas posteriores, va devolviendo valores de la misma serie de números aleatorios, sin cambiar la semilla. Pero si se vuelve a ejecutar el programa, se volverá a leer la semilla del reloj. Esto garantiza que en cada ejecución del programa, sea casi imposible que se repita la misma serie de números. Por ejemplo, para mostrar 10 números aleatorios de 0.0 a 1.0 pondremos,

```
for (int i = 0; i < 10; i++) {  
    double num = Math.random();  
    System.out.println(num);  
}
```

Si ejecutamos este código varias veces, obtendremos series distintas de números, ya que en la primera iteración del bucle se establece la semilla al leer del reloj del sistema y ya no se cambia hasta el final de la ejecución del programa. En ejecuciones posteriores, la semilla leída será distinta, ya que la lectura del reloj será otra.

Lo más común es que necesitemos números aleatorios enteros. Para conseguirlo, basta con multiplicar los valores devueltos por `random()`, por 10, 100, etc., dependiendo del número de cifras deseado, y nos quedemos con la parte entera por medio de un molde o cast (`int`). Por ejemplo, para obtener 20 números aleatorios entre 0 y 9 (inclusive),

```
for (int i = 0; i < 20; i++) {  
    int num = (int) (Math.random() * 10);  
    System.out.println(num);  
}
```

Si queremos que los valores generados por `Math.random()` estén entre 1 y 10, sumaremos 1 a los valores obtenidos,

```
for (int i = 0; i < 20; i++) {  
    System.out.println((int) (Math.random() * 10) + 1);  
}
```

Así, jugando con el factor multiplicador y el valor sumado, podemos obtener números aleatorios enteros en cualquier intervalo deseado.

A.2. Intervalos de tiempo

Cuando queremos medir un periodo de tiempo corto durante la ejecución de un programa, podemos usar el método `System.currentTimeMillis()` que, al leer del reloj del procesador, devuelve la hora actual como valor `long`, con el número de milisegundos transcurridos entre las 0 horas del 1 de enero de 1970 y el instante actual. Naturalmente, el interés radica en llamar a esta función en dos instantes diferentes y calcular el tiempo transcurrido restando los dos valores devueltos.

Si el intervalo de tiempo que queremos medir es demasiado pequeño para que nos sirvan los milisegundos (por ejemplo, si queremos medir la eficiencia de un algoritmo calculando el tiempo que tarda en ejecutarse), podemos usar el método `System.nanoTime()`, que devuelve el tiempo transcurrido desde un instante de referencia arbitrario hasta el momento presente en nanosegundos¹. No obstante, la exactitud del valor devuelto va a depender de la arquitectura de nuestro ordenador.

Por ejemplo, para medir el tiempo empleado para ejecutar un bucle donde se generan 100 números aleatorios, ejecutaremos el trozo de código:

```
long principio = System.nanoTime();  
for (int i = 0; i < 100; i++) {  
    System.out.println(Math.random());  
}  
double fin = System.nanoTime();  
System.out.println("Tiempo: " + (fin - principio) + " nanosegundos");
```

A veces, nos interesa detener la ejecución del programa durante un intervalo de tiempo preciso. Para eso se usa el método `sleep()` de la clase `Thread`. Este método tiene dos implementaciones sobrecargadas,

```
static void sleep(long milisegundos)  
static void sleep(long milisegundos, int nanosegundos)
```

La primera detiene la ejecución durante los milisegundos que se le pasan como argumento, mientras que la segunda, añade a la pausa los nanosegundos que se le pasan como segundo argumento. Por ejemplo, si queremos que la ejecución se detenga 4000 milisegundos (4 segundos) y 100 nanosegundos pondremos,

```
Thread.sleep(4000, 100);
```

Se puede comprobar con la función `.nanoTime()`, como hicimos más arriba.

¹Mil millonésimas de segundos.

A.3. Fechas y horas

La implementación de las horas y las fechas en Java se basa en el calendario Gregoriano. Para disponer de las clases que vamos a ver deberemos importarlas con el paquete `java.time` y diversos subpaquetes, así como `java.util.Locale` (mirar la ayuda de Java en línea para cada clase), aunque se puede dejar a Netbeans esa tarea.

A.3.1. `LocalDate`

Para fechas donde no tenemos en cuenta zonas horarias, usaremos `LocalDate`. Los objetos `LocalDate` se crean con la función estática `of()` de la clase `LocalDate`. Si queremos crear un objeto con la fecha 12 de febrero de 2016, escribiremos,

```
LocalDate fecha1 = LocalDate.of(2016, 2, 12);
```

o bien, para el mes, podemos poner `Month.FEBRUARY` en vez de 2.

`Month` es un tipo enumerado con los valores `JANUARY`, `FEBRUARY`, etc. de los meses del año en inglés. Un objeto `LocalDate` se puede mostrar por consola directamente, ya que tiene implementado `toString()`,

```
System.out.println(fecha1); //mostrará: 2020-02-12
```

Si necesitamos un `LocalDate` con la fecha actual, leída del sistema, llamaremos al método `now()`,

```
LocalDate fechaActual = LocalDate.now();
```

A partir de ella, podemos obtener el día de la semana, que es un objeto de la clase `DayOfWeek`,

```
DayOfWeek diaSemana = fecha1.getDayOfWeek();
System.out.println(diaSemana);
```

Aparecerá por pantalla `WEDNESDAY`, ya que el 2 de febrero de 2016 es miércoles. `DayOfWeek` es otro tipo enumerado con los valores `MONDAY`, `TUESDAY`, etc. de los días de la semana en inglés.

Para obtener, a partir de un `LocalDate`, el día del mes, el mes del año o el año, se usan los métodos,

```
int getDayOfMonth() //devuelve el día del mes
Month getMonth() //devuelve un valor del tipo enumerado Month
int getMonthValue() //devuelve el mes como número del 1 al 12
int getYear() //devuelve el año
```

Por ejemplo,

```
fecha1.getMonthValue()
```

devolverá un 2.

Las fechas se pueden incrementar con el método `plus()`. Por ejemplo,

```
LocalDate fecha2 = fecha1.plus(3, ChronoUnit.DAYS); //2020-02-15
```

incrementará en 3 días `fecha1`. El segundo argumento es del tipo enumerado `ChronoUnit`, que tiene como valores distintas unidades de medida del tiempo, como `DAYS`, `MONTHS`, `YEARS` y `WEEKS` entre otros. El primer parámetro de `plus()` es el número de unidades que queremos añadir del tipo de unidad del segundo parámetro. También podemos decrementar con el método `minus()`. Por ejemplo, para disminuir `fecha1` en dos años,

```
LocalDate fecha3 = fecha1.minus(2, ChronoUnit.YEARS); //2018-02-12
```

Asimismo podemos comparar dos fechas con,

```
boolean equals() //true si son iguales
int compareTo() //compara con orden cronológico
```

El método,

```
boolean isLeapYear()
```

nos devuelve `true` si el año de la fecha que lo invoca es bisiesto.

Para calcular el período transcurrido entre dos fechas, usamos el método,

```
Period until(LocalDate otraFecha)
```

donde `Period` es una clase cuyos objetos representan un periodo de tiempo, que consta de años, meses y días. Para obtener los días, meses o años de un periodo usaremos el método,

```
long get(TemporalUnit unidad)
```

donde `unidad` puede ser `ChronoUnit.DAYS`, `ChronoUnit.MONTHS` o `ChronoUnit.YEARS`.

Por ejemplo, si creamos `fecha4`, 14 meses posterior a `fecha1` y calculamos el periodo entre una y otra,

```
LocalDate fecha4 = fecha1.plus(14, ChronoUnit.MONTHS);
Period periodo = fecha1.until(fecha4);
```

El periodo devuelto consiste en 1 año y 2 meses. En efecto,

```
periodo.get(ChronoUnit.MONTHS)
```

devuelve un 2. También podemos obtener el periodo completo expresado en meses con,

```
periodo.toTotalMonths()
```

que nos devuelve 14.

A la hora de mostrar una fecha, o de obtenerla a partir de una cadena, se usa la clase `DateTimeFormatter`, cuyos objetos definen un formato de fecha y/o hora. El método `ofPattern()` genera el formato a partir de un patrón que se le pasa como parámetro. Por ejemplo, si queremos mostrar por pantalla una fecha con un formato del tipo 12-02-2020, crearemos el formato,

```
DateTimeFormatter formato1 = DateTimeFormatter.ofPattern("dd-MM-yyyy");
```

es decir, dos cifras para el día (dd), guión (-), dos cifras para el mes (MM), guión y cuatro cifras para el año (yyyy). El formateador se aplica a una fecha para generar la cadena que la representa,

```

LocalDate fecha1 = LocalDate.of(2016, 2, 12);
String cadenaFecha = fecha1.format(formato1);
System.out.println(cadenaFecha); //mostrará 12-02-2020

```

Podemos añadir cualquier texto que queramos en el formato, encerrándolo entre comillas simples,

```

DateTimeFormatter formato2 =
DateTimeFormatter.ofPattern("dd 'del mes' MM 'del año' yyyy");

```

Con `formato2`, `cadenaFecha` sería,

12 del mes 02 del año 2020

Si queremos un formato estándar adaptado a un país concreto,

```

DateTimeFormatter formato3 = DateTimeFormatter
    .ofLocalizedDate(FormatStyle.FULL)
    .withLocale(Locale.getDefault());

```

donde el método `ofLocalizedDate()` nos permite escoger un formato más o menos detallado (`FormatStyle.FULL`, `FormatStyle.SHORT`, `FormatStyle.LONG` o `FormatStyle.MEDIUM`), mientras que `withLocale()` nos adapta el formato al país que queramos de entre una lista (`Locale.US`, `Locale.ITALY`,...) o bien adopta el formato del país del sistema con `Locale.getDefault()`.

Con `formato3`, `fecha1` daría lugar a la cadena,

miércoles 12 de febrero de 2020

en castellano, ya que `getDefault()` detectará que nos hallamos en España.

Un formateador también sirve para analizar una cadena y convertirla en un objeto `LocalDate`, haciendo el trabajo inverso al que acabamos de ver. Para ello se usa el método estático de `LocalDate`,

```

static LocalDate parse(String cadEntrada, DateTimeFormatter formateador)

```

Por ejemplo, si queremos generar la fecha 15 de junio de 2010, a partir de la cadena "15-06-2010" con el formato definido en `formato1` (dd-MM-yyyy),

```

String cadenaFecha = "15-06-2010";
LocalDate fecha5 = LocalDate.parse(cadenaFecha, formato1);

```

Así podremos leer fechas a partir de cadenas introducidas por el teclado.

Si la cadena no se ajusta al formato que le hemos pasado como segundo parámetro, saltará la excepción: `DateTimeParseException`.

A.3.2. `LocalTime`

Para el tratamiento de las horas en Java se usa la clase `LocalTime`, que tiene un conjunto de métodos similares a `LocalDate`, pero con campos distintos. En vez de días, meses y años, trabaja con horas, minutos, segundos y nanosegundos (mil millonésimas de segundo).

Como ocurre con `LocalDate`, los objetos `LocalTime` se pueden crear con el método estático sobrecargado,

```
static LocalTime of(int hora, int minuto, int segundo, int nanosegundo)
static LocalTime of(int hora, int minuto, int segundo)
```

de la clase `LocalTime`,

```
LocalTime time1=LocalTime.of(12, 30, 45);
System.out.println(time1); //muestra 12:30:45
```

También podemos leer la hora del sistema con,

```
static LocalTime now()
```

Por ejemplo,

```
LocalTime time2 = LocalTime.now();
```

A partir de un objeto `LocalTime`, se pueden obtener sus distintos campos con,

```
int getHour()
int getMinute()
int getSecond()
```

Asimismo, `LocalTime` dispone de los métodos de comparación ,

```
boolean equals(Object otraHora)
int compareTo(Object otraHora)
```

así como los de incremento y decremento,

```
LocalTime plus(long cantidad, TemporalUnit unidad)
LocalTime minus(long cantidad, TemporalUnit unidad)
```

Por ejemplo, para incrementar `time1` (recordemos que es 12:30:45) en 45 minutos,

```
LocalTime time3 = time1.plus(45, ChronoUnit.MINUTES);
System.out.println(time3); //aparece en pantalla 13:15:45
```

Los periodos comprendidos entre dos objetos `LocalTime`, se calculan con,

```
long until(LocalTime otraHora, TemporalUnit unidad)
```

que, a diferencia de `LocalDate`, devuelve un entero largo con el periodo expresado en la unidad que se le pasa como segundo parámetro. Por ejemplo, calculemos en minutos la diferencia entre `time1` y `time3`, que sabemos son 45 minutos,

```
long diferencia = time1.until(time3, ChronoUnit.MINUTES); //45
```

Por último, para imprimir una hora o generar un objeto `LocalTime` a partir de una cadena, se usa la misma clase formateadora `DateTimeFormatter`, así como los métodos `ofPattern()` y `parse()`,

```
DateTimeFormatter formato4 = DateTimeFormatter.ofPattern("HH.mm.ss");
String cadenaTime = formato4.format(time3); //"13.15.45"
```

o, la inversa,

```
LocalTime time4 = LocalTime.parse(cadenaTime, formato4);
```

Hay una tercera clase que combina las fechas y las horas, `LocalDateTime`. Sus métodos son los mismos que los de `LocalDate` y `LocalTime`, y emplea también `DateTimeFormatter`.

Clases envoltorio (*wrappers*)

Hay estructuras de datos, como las colecciones, que no trabajan con tipos primitivos, sino con objetos. Esto nos impide insertar valores como los números enteros o los reales. Para esto, Java ha implementado unas clases que «envuelven» datos primitivos dentro de un objeto llamado envoltorio o *wrapper*, que se puede insertar en una lista o un conjunto, o bien pasarlo a una función donde se espera un objeto como argumento.

Para cada tipo primitivo hay definida una clase envoltorio (véase Tabla B.1).

Tabla B.1. Tipos primitivos y clases envoltorio

Tipo primitivo	Clase envoltorio
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

Todas las clases envoltorio, salvo **Character**, heredan de la clase abstracta **Number**. La forma de construir un *wrapper* a partir de un valor primitivo, es decir, de «envolverlo», es pasándole el valor como argumento al constructor,

```
Integer x = new Integer(3);
```

x sería un **Integer** que «guarda» un 3 en su interior. Los *wrappers* se pueden manipular como si fueran primitivos,

```
Integer x = new Integer(3);
Integer y = new Integer(5);
Integer z = x + y;
System.out.println(z);
```

que mostrará en pantalla un 8, ya que la suma entre **Integer** se realiza como si fueran **int**.

En realidad, Java se suele encargar de envolver automáticamente los valores primitivos sin tener que recurrir al constructor. Podríamos haber inicializado,

```
Integer x = 3, y = 5;
```

como si fueran del tipo `int`. El compilador se encarga de envolver el 3 y el 5 antes de asignarlo a `x` e `y` respectivamente. Este mecanismo se llama *autoboxing*. Del mismo modo, lo desenvuelve cuando hace falta,

```
int w = x;
```

que desenvuelve el 3 de su envoltorio antes de asignarlo a la variable entera `w`.

Algo parecido ocurre cuando queremos realizar operaciones que involucran primitivos y *wrappers* a la vez, como `int` con `Integer`,

```
Integer v = x + 9;
```

El compilador desenvuelve `x`, lo suma con el 9 y envuelve el resultado para asignarlo a `v`. Todo lo dicho de `int` e `Integer`, puede decirse de las otras parejas primitivo-*wrapper*.

Con todas las clases envoltorio que heredan de `Number` se pueden realizar las mismas operaciones que con sus tipos primitivos. Además, son aplicables los operadores relacionales,

```
Double x1 = 1.23, x2 = 4.21;  
System.out.println(x1 > x2);
```

que devuelve un `false`, como haría con tipos `double`. Sin embargo, hay que usar `equals()` en lugar de `==`, ya que se comparan objetos.

En general, Java envuelve y desenvuelve, según las necesidades, para colocar un tipo primitivo donde se espere primitivo y un *wrapper* donde se espere un *wrapper*, por ejemplo, en un parámetro de una función, como,

```
void funcion(Integer x) {  
    ...  
}
```

que tiene un parámetro envoltorio `Integer`. Sin embargo, puede ser llamada con un parámetro `int`,

```
funcion(5);
```

Java se encarga de envolver el 5 antes de pasarlo a la función.

Los envoltorios, como objetos que son, disponen de una serie de métodos. Los más útiles son los que interpretan cadenas de caracteres (a menudo leídas del teclado) que representan valores y los convierten. Por ejemplo, la clase `Double` dispone del método:

```
static double parseDouble(String cadena)
```

al que se le pasa una cadena que representa un número real y lo convierte en un `double`. Por ejemplo,

```
String cad = "23.546";  
double t = Double.parseDouble(cad);  
System.out.println(t);
```

La variable real `t` contiene el valor decimal 23.546.

El resto de los *wrappers* disponen de métodos análogos.

Bibliografía

- [1] Barnes, D. J. y Kölling, M. *Programación Orientada a Objetos con Java*. Prentice-Hall, Madrid, 2013.
- [2] Brassard, G. y Bratley, P. *Fundamentos de Algoritmos*. Prentice Hall, Madrid, 2000.
- [3] Cadenhead, R. y Lemay, L. *Teach Yourself Java 6 in 21 days*. Howard W. Sams & Co, Indianápolis, 2007.
- [4] Eckel, B. *Thinking in Java*, 4th edition. Prentice Hall, Engelwood Clifts, 2008.
- [5] Schildt, H. *The Complete Reference*, 9th edition. McGraw-Hill Osborne, Nueva York, 2014.

Aprende a programar con Java

Un enfoque práctico partiendo de cero

Este libro va dirigido a aquellas personas que necesitan iniciarse en el mundo de la programación partiendo de cero, es decir, sin conocimientos previos. Está pensado para el aprendizaje autodidacta, aunque también sirve como apoyo práctico para el estudio de la programación en facultades, escuelas universitarias y ciclos superiores de formación profesional.

En esta segunda edición se ha revisado y actualizado el contenido para introducir las importantes novedades incorporadas en la versión 8 de Java. Se han reelaborado algunos capítulos y se han añadido otros nuevos. Asimismo, se han incluido nuevos ejercicios que permiten poner en práctica los nuevos conceptos incluidos.

Para aprender a programar se puede usar cualquier lenguaje. Para esta obra se ha elegido el lenguaje Java por ser el más extendido en el mundo de las empresas y en internet dadas su seguridad y su portabilidad. La metodología seguida es eminentemente práctica, pues está basada en breves introducciones teóricas con abundantes ejemplos a las que siguen numerosos ejercicios prácticos, resueltos con detalle y cuya dificultad crece gradualmente. Por ello, si el lector sigue el libro hasta el final con dedicación y paciencia, estará en posesión de los conceptos y las herramientas más importantes de la programación avanzada y del lenguaje Java sin excesiva dificultad.

Los autores son profesores de Informática, especialistas en lenguajes de programación y con una larga experiencia investigadora y docente, tanto en la enseñanza secundaria como en la universitaria. A lo largo de su trayectoria profesional se han mantenido en contacto con el mundo laboral, por lo que están al día de sus necesidades y de la evolución de las nuevas tecnologías. Actualmente imparten la asignatura de Programación del Ciclo Formativo de grado superior de Desarrollo de Aplicaciones Web, de la familia profesional de Informática y Comunicaciones.

