SILVIA GUARDATI BUEMO

Estructuras de Datos Básicas

PROGRAMACIÓN ORIENTADA A OBJETOS CON Java

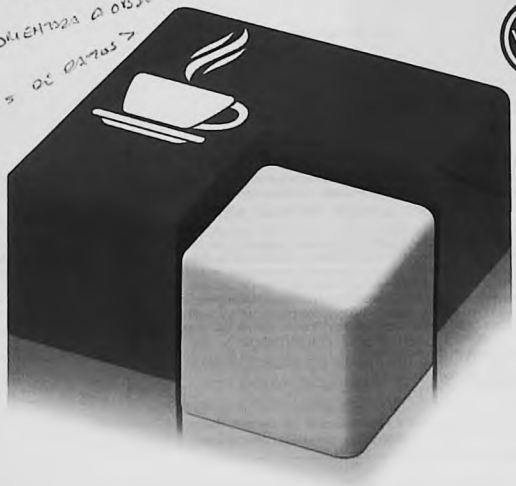
Apoyo en la web

Alfaomega

Estructuras de Datos Básicas

PROGRAMACIÓN ORIENTADA A OBJETOS CON Java

Apoyo en la web



Alfaomega

Director Editorial
Marcelo Grillo Giannetto
mgrillo@alfaomega.com.mx

Jefe de Ediciones
Francisco Javier Rodríguez Cruz
jrodriguez@alfaomega.com.mx

Datos catalográficos
Guardati Buemo Silvia
Estructuras de datos básicas. Programación orientada a objetos con Java
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V. México
ISBN: 978-607-622-451-9
Formato: 21 x 24 cm Páginas: 416

Estructuras de datos básicas. Programación orientada a objetos con Java
Silvia Guardati Buemo

Primera edición: Alfaomega Grupo Editor, México, junio de 2015

Pitágoras 1139, Col. Del Valle, C.P. 03100, México, D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN 978-958-778-073-4

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No 20-46, Barrio San Luis– Bogotá, Colombia,
Tel.: (57-1) 746 0102. E-mail: cliente@alfaomegacolombiana.com

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile.
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. "11", Buenos Aires, Argentina,
C.P. 1057 – Tel.: (54-11) 4811-7183 / 8352, E-mail: ventas@alfaomegaeditor.com.ar

ACERCA DEL AUTOR

Silvia Guardati Buemo. Profesora del Departamento Académico de Computación, del Instituto Tecnológico Autónomo de México (ITAM), desde 1988 a la fecha. Es egresada de la Universidad Tecnológica Nacional, Argentina (1984) y del CINVESTAV-IPN, México (1987). Sus áreas de interés son las Estructuras de Datos, la Programación Orientada a Objetos y la Ingeniería de Software. Es autora de otros libros de texto sobre Estructuras de Datos y de capítulos de libros en otras áreas de la computación.

Se debe pedir a cada cual, lo que está a su alcance realizar

Antoine Saint Exupery

MENSAJE DEL EDITOR

Una de las convicciones fundamentales de Alfaomega es que los conocimientos son esenciales en el desempeño profesional, ya que sin ellos es imposible adquirir las habilidades para competir laboralmente. El avance de la ciencia y de la técnica hace necesario actualizar continuamente esos conocimientos, y de acuerdo con esto Alfaomega publica obras actualizadas, con alto rigor científico y técnico, y escritas por los especialistas del área respectiva más destacados.

Consciente del alto nivel competitivo que debe de adquirir el estudiante durante su formación profesional, Alfaomega aporta un fondo editorial que se destaca por sus lineamientos pedagógicos que coadyuvan a desarrollar las competencias requeridas en cada profesión específica.

De acuerdo con esta misión, con el fin de facilitar la comprensión y apropiación del contenido de esta obra, cada capítulo inicia con el planteamiento de los objetivos del mismo y con una introducción en la que se plantean los antecedentes y una descripción de la estructura lógica de los temas expuestos, asimismo a lo largo de la exposición se presentan ejemplos desarrollados con todo detalle y cada capítulo concluye con un resumen y una serie de ejercicios propuestos.

Además de la estructura pedagógica con que están diseñados nuestros libros, Alfaomega hace uso de los medios impresos tradicionales en combinación con las Tecnologías de la Información y las Comunicaciones (TIC) para facilitar el aprendizaje. Correspondiente a este concepto de edición, todas nuestras obras tienen su complemento en una página Web en donde el alumno y el profesor encontrarán lecturas complementarias así como programas desarrollados en relación con temas específicos de la obra.

Los libros de Alfaomega están diseñados para ser utilizados en los procesos de enseñanza aprendizaje, y pueden ser usados como textos en diversos cursos o como apoyo para reforzar el desarrollo profesional, de esta forma Alfaomega espera contribuir a la formación y al desarrollo de profesionales exitosos para beneficio de la sociedad, y espera ser su compañera profesional en este viaje de por vida por el mundo del conocimiento.

TABLA DE CONTENIDO

Capítulo 8

INTRODUCCIÓN

Este libro está dedicado al estudio y a la aplicación de estructuras de datos básicas en el entorno de la programación orientada a objetos (POO). Por lo tanto, el contenido se centra en los principales conceptos de la POO y las estructuras de datos lineales implementadas usando este paradigma de programación.

Todos los ejemplos del libro están desarrollados en Java y para esto se usó el IDE Netbeans, el cual está disponible gratuitamente en: https://netbeans.org

Para la representación gráfica de las clases y de las relaciones entre ellas se usó el lenguaje UML (Unified Modeling Language) que es un estándar reconocido y aceptado por la industria del software.

Si bien se utilizó Java, muchos de los conceptos son presentados de manera independiente del lenguaje de tal manera que el lector puede posteriormente implementar esos conceptos por medio de otros lenguajes de POO.

A lo largo de toda la obra se busca desarrollar en el lector responsabilidad y compromiso con respecto al software que genera. En los casos que así lo justifican se comentan distintas alternativas de solución, de tal manera que el estudiante tome conciencia de las características deseables en un buen producto de software y procure tenerlas presentes al momento de codificar. Es importante que aspectos como eficiencia, mantenibilidad, pruebas, eficacia, completitud, etc., sean parte de cualquier código generado, independientemente del tamaño o destino del mismo. Se dice que *la calidad de un producto de software está determinada por la calidad de su peor componente*.

Esta obra está orientada a:

- Los que quieran aprender a programar en un entorno de POO.
- Los que quieran aprender a programar en Java.
- Los que quieran entender y aplicar el paradigma orientado a objetos en el diseño e implementación de soluciones basadas en la programación.
- Los que quieran entender y aplicar estructuras de datos lineales en la solución de problemas.

ORGANIZACIÓN DEL LIBRO

El primer capítulo está dedicado a estudiar los conceptos básicos usados en la programación, sin importar el tipo de paradigma que se utilice. Por lo tanto, se presentan algunos conceptos, las estructuras algorítmicas de control, la programación modular y una introducción a las pruebas de software. Este capítulo es importante para aquellos lectores que no sepan programar o que no estén familiarizados con el lenguaje Java.

Los capítulos segundo y tercero introducen al lector al mundo de la programación orientada a objetos. Principios, características, elementos, etc. son explicados y ejemplificados para ayudar a su comprensión. En el segundo capítulo se presentan los elementos pilares de la POO, mientras que el tercero se dedica a conceptos más avanzados.

Los siguientes capítulos, cuarto y quinto, presentan los arreglos, primera estructura de datos estudiada en esta obra. El material cubre desde los conceptos básicos de los arreglos hasta elementos más complejos aprovechando los beneficios que ofrece la POO. En el quinto capítulo también se presentan algunas clases del lenguaje Java definidas para el manejo de arreglos unidimensionales.

El sexto capítulo se dedica al estudio de estructuras enlazadas, siendo éstas una introducción a las estructuras dinámicas de datos. Se presentan ventajas y desventajas del manejo dinámico sobre el estático que caracteriza a los arreglos.

En el capítulo séptimo se estudian las pilas y colas, siendo ambas tipos abstractos de datos. Por lo tanto, se analiza la implementación de las dos por medio de arreglos y de estructuras enlazadas. Para que el lector adquiera una idea más completa de estas estructuras se presentan aplicaciones de las mismas.

Por último, en el capítulo octavo se presenta la recursión como una herramienta para resolver ciertos tipos de problemas. La recursión está basada en la división del problema, de tal manera que resulte más fácil de resolver.

MATERIAL WEB

Todos los capítulos cuentan con ejemplos y ejercicios que ayudan al lector a reafirmar los conceptos estudiados.

Como parte del material adicional del libro, en el sitio Web de esta obra se incluye:

- Proyecto Netbeans (Java) con todos los programas presentados en cada uno de los capítulos del libro. Además, en algunos capítulos se muestra parte del código de ciertos algoritmos, pero en el proyecto se encuentra el código completo.
- Proyecto Netbeans (Java) con la solución de todos los ejercicios de todos los capítulos que requieren programación.
- Archivo con la solución de aquellos ejercicios que implican análisis, pero no programación.
- Presentación "power point" del contenido del libro.
- Presentación descargable en formato pdf.
- Mapas conceptuales.

PLATAFORMA DE CONTENIDOS INTERACTIVOS

Para tener acceso al material de la plataforma de contenidos interactivos de esta obra, siga los siguientes pasos:

1. Ir a la pagina: http://libroweb.alfaomega.com.mx
2. Ir a la sección de catálogo y seleccionar la imagen de la portada del libro, al dar doble clic sobre ella, tendrá acceso al material descargable.

NOTA: se recomienda respaldar los archivos descargados de la página Web en un soporte físico.

ELEMENTOS BÁSICOS PARA EMPEZAR A PROGRAMAR EN JAVA

1

Contenido

Competencias

- Explicar los conceptos básicos necesarios para el desarrollo de soluciones algorítmicas de problemas.
- Explicar las estructuras algorítmicas selectivas y repetitivas.
- Explicar la programación modular como una estrategia de solución de problemas.
- Señalar la importancia de las pruebas en el desarrollo de software.
- Señalar buenas prácticas de programación.

1.1 INTRODUCCIÓN

En este capítulo se presentan los elementos necesarios para empezar a programar. Si el lector ya cuenta con este conocimiento, puede ir directamente al capítulo 2, que trata de la programación orientada a objetos.

En aquellos temas que así lo requieren, se utiliza Java como lenguaje de programación. Sin embargo, es importante resaltar que la mayoría de los conceptos son generales y podrían implementarse usando diversos lenguajes de programación.

1.2 CONCEPTOS BÁSICOS

A continuación estudiaremos algunos conceptos básicos usados en el planteamiento algorítmico de la solución de un problema y en la programación de la misma.

Variable

Es un espacio en la memoria de la computadora donde se almacena información. Dicho espacio está referenciado por un nombre único. La principal característica de una variable es que puede cambiar su valor durante la ejecución del programa. Es decir, puede guardar cierto valor y posteriormente se puede modificar. Una variable se representa gráficamente como lo muestra la figura 1.1.

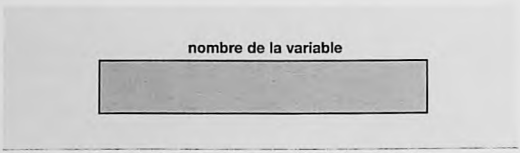


Figura 1.1 Representación gráfica de una variable

En el lenguaje Java las variables se identifican por medio de letras que pueden estar seguidas de dígitos y/o algunos caracteres especiales. Se tiene la convención de que los nombres se escriban con minúsculas y en caso de que estén formados por más de una palabra la inicial de la segunda palabra y consecutivas se escriben con mayúscula. A esta notación se le conoce con el nombre de "camello" o "camel". Es importante tener en cuenta que las letras mayúsculas son distintas de las letras minúsculas. Ejemplos de nombres de variables:

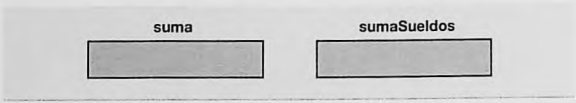


Figura 1.2 Ejemplo de representación gráfica de variables

Constante

Es un espacio en la memoria de la computadora donde se almacena información. Dicho espacio está referenciado por un nombre único. La principal característica de una constante es que NO puede cambiar el valor que almacena durante el tiempo de ejecución. Es decir, una vez que se le asigna un valor, no es posible modificarlo durante la corrida del programa.

Una constante se representa gráficamente como una variable. En el lenguaje Java las constantes se identifican por medio de caracteres que pueden estar seguidos de dígitos y/o algunos caracteres especiales (por ejemplo: _). Se tiene la convención de que los nombres se escriban todos con mayúsculas, y si están formados por dos o más palabras, éstas se separan con guión bajo. Por ejemplo: PI, TOTAL_ALUMNOS, etcétera.

Tipos de datos

Hace referencia a la naturaleza de la información que se almacena y se procesa en un programa. Los tipos de datos más usados son los números y las cadenas de caracteres. También, en la mayoría de los lenguajes de programación, se tienen datos lógicos o booleanos, los cuales hacen referencia a los valores posibles de la lógica binaria (verdadero/true o falso/false).

Dependiendo del tipo de datos es la cantidad de espacio de memoria que se reserva. Por ejemplo, a un número con parte decimal se le reserva más espacio que a un número entero. En Java los tipos primitivos de datos son: int, short, long, double, float, boolean, char y byte.

Para almacenar y manipular cadenas de caracteres se usa la clase String. Por ahora diremos que esta clase permite representar el concepto de cadena de caracteres y ofrece algunos métodos para poder trabajar con esa información. Más adelante, en este capítulo, se volverá sobre este tema.

Buenas prácticas

- ☑ Los nombres de las variables se escriben con minúsculas.
- ☑ Si está formado por dos o más palabras, la segunda y subsecuente se escriben con la inicial mayúscula.
- ☑ Los nombres de las constantes se escriben con mayúsculas.
- ☑ Si está formado por dos o más palabras, éstas se separan con guión bajo.
- ☑ Se deben elegir nombres para las variables y para las constantes relacionados con la información que almacenan.

Expresiones

Hacen referencia a operaciones que se realizan sobre variables y/o constantes. Según la operación aplicada, las expresiones reciben el nombre de aritméticas, relacionales y lógicas.

- *Aritméticas*: aquellas que resultan de aplicar operadores aritméticos a números, cuyo resultado es un número. Los operadores aritméticos más usados son:
  - + (suma), – (resta), / (división), * (multiplicación), % (módulo o residuo de la división entera), () (los paréntesis no son operadores, se usan para alterar la prioridad de los operadores).

Además, Java tiene una gran cantidad de métodos para calcular: potencia, valor absoluto de un número, raíz cuadrada, seno, coseno, etc. Estos métodos pertenecen a la clase Math. A continuación se presentan algunos de ellos:[1]

[1] La lista completa puede obtenerse en: http://docs.oracle.com/Javase/1.4.2/docs/api/Java/lang/Math.html

- **pow (base, exponente)**: calcula la potencia de *base* elevada al *exponente*. Se invoca:

  variable = Math.pow (base, exponente);

Ejemplo 1.1

resultado = Math.pow (13.8, 3);

Luego de ejecutarse la instrucción, el *resultado* será igual a: 2628.072

- **sqrt (número)**: calcula la raíz cuadrada de *número*. Se invoca:

  variable = Math.sqrt (número);

Ejemplo 1.2

resultado = Math.sqrt (36);

Luego de ejecutarse la instrucción, el *resultado* será igual a: 6.0

En el caso de los métodos pow y sqrt, la *variable* debe ser de tipo double. Los métodos también se pueden usar en una expresión aritmética, como se verá más adelante.

- **abs (número)**: calcula el valor absoluto de *número*. Se invoca:

  variable = Math.abs (número);

  Donde *variable* debe ser del mismo tipo que *número* o compatible. Es decir, si número es de tipo int, variable puede ser int (o double). Este método también se puede usar en una expresión aritmética, como se verá más adelante.

Ejemplo 1.3

resultado = Math.abs (-22.8);

Luego de ejecutarse la instrucción, el *resultado* será igual a: 22.8

entero = Math.abs (16);

Luego de ejecutarse la instrucción, el *entero* será igual a: 16

Cada operador tiene asociada una prioridad, misma que determina el orden en que se van a ejecutar las operaciones dentro de una expresión. En la tabla 1.1 se observan los operadores y su prioridad.

Tabla 1.1 Operadores y sus prioridades

Operador	Prioridad
()	Si se encuentran () en una expresión, lo primero que se evalúa es la subexpresión encerrada entre los paréntesis.
Métodos de Java para potencia, valor absoluto, raíz cuadrada, etcétera.	Son los siguientes en ser evaluados en una expresión, si hubiera paréntesis. En caso contrario, son los primeros en aplicarse.
*, /, %	Todos tienen la misma prioridad, la cual es menor a la categoría anterior. En Java, si la / afecta a 2 números enteros, entonces el resultado también es un entero (perdiendo la parte decimal si la tuviera). En Java, si la / afecta a 2 números con parte decimal, entonces el resultado también es un número con parte decimal (aunque la misma sea 0).
+, −	Todos tienen la misma prioridad, la cual es menor a la categoría anterior.

Si en una expresión se presentan dos o más operadores de un mismo nivel de prioridad, entonces se ejecutan de izquierda a derecha. Obsérvense los siguientes ejemplos (la primera expresión es la original, la que se quiere evaluar; la de la derecha se calcula paso a paso, según las prioridades de sus operadores):

Ejemplo 1.4

4 + 2 * 3 → 4 + 6 → **10**

23 / 5 → **4**

23 % 5 → **3**

23.0 / 5.0 → **4.6**

20.0 / 5.0 → **4.0**

20 / 5 → **4**

3 * 2 * 8 / 2 → 6 * 8 / 2 → 48 / 2 → **24**

2 * (3 + 5) % 10 → 2 * 8 % 10 → 16 % 10 → **6**

2 * (3 + 5) / 10 → 2 * 8 / 10 → 16 / 10 → **1**

23 – 8 * Math.abs (-4) / 10 → 23 – 8 * 4 / 10 → 23 – 32 / 10 → 23 – 3 → **20**

Math.pow(3, 2) / 2.0 → 9.0 / 2.0 → **4.5**

31 % 4 + 2.5 – 9 /2 * (3 + 2) → 31 % 4 + 2.5 – 9 / 2 * 5 → 3 + 2.5 – 9 / 2 * 5 → 3 + 2.5 – 4.* 5 → 3 + 2.5 – 20 → 5.5 – 20 → **– 14.5**

Otros ejemplos pueden consultarse en el archivo[2] PruebasCapítulo1.java, subprograma *ejemplosExpresionesAritméticas( )*.

[2] Todos los programas de este capítulo se encuentran en el proyecto EstructurasDatosBásicas, paquete cap1.

- *Relacionales:* aquellas que resultan de aplicar operadores relacionales a números o caracteres, cuyo resultado es un valor booleano o lógico (true o false). Los operadores relacionales son:

  < (menor que), > (mayor que), == (igual que), <= (menor o igual que), >= (mayor o igual que), != (distinto que).

Los operadores relacionales se usan en expresiones cuyo resultado se guarda en una variable o en condiciones como parte de una estructura selectiva o cíclica. Esto último se verá con mayor detalle en las siguientes secciones. A continuación se presentan algunos ejemplos en que se resuelven las operaciones paso por paso:

Ejemplo 1.5

23.5 > 18 → al evaluarse da **true**

18 < 13 → al evaluarse da **false**

24 == 12 * 2 → 24 == 24 → al evaluarse da **true**

(3 + 5) * 2 <= 50 → 8 * 2 <= 50 → 16 <= 50 → al evaluarse da **true**

Math.sqrt (25) – 10 + 3.5 != 18 * Math.pow (3.25, 2) / 5 → 5.0 – 10 + 3.5 != 18 * Math.pow (3.25, 2) / 5 → 5.0 – 10 + 3.5 != 18 * 10.5625 / 5 → 5.0 – 10 + 3.5 != 190.125 / 5 → 5.0 - 10 + 3.5 != 38.025 → -5.0 + 3.5 != 38.025 → -1.5 ! = 38.025 → al evaluarse da **true**

En algunos de los ejemplos anteriores se combinan operadores aritméticos y relacionales. En estos casos, primero se operan los aritméticos —con los que se obtiene números— y luego los relacionales, y cuyo resultado final es un valor booleano.

- *Lógicas (o booleanas):* son aquellas que resultan de aplicar operadores lógicos a valores lógicos, lo que da como resultado también un valor lógico (*true o false*). Los operadores lógicos son:

  && (conjunción), || (disyunción), ! (negación)

Los operadores lógicos se usan para generar condiciones compuestas a partir de condiciones simples. Si una expresión tiene && combinando dos condiciones se exige que ambas sean verdaderas para que la expresión sea verdadera. Si el operador fuese ||, entonces es suficiente que alguna de las condiciones sea verdadera para que toda la expresión resulte verdadera. Por su parte, el ! obtiene el otro valor de verdad. Es decir, si se niega un valor true, se tiene el false. Si el valor original es false y se niega, entonces se obtiene el true. A continuación se presentan las tablas de verdad de estos operadores.

p	q	p && q
true	true	**true**
true	false	false
false	false	false
false	true	false

p	q	p || q
true	true	**true**
true	false	**true**
false	false	false
false	true	**true**

p	! p
true	false
false	**true**

Enseguida se muestran algunos ejemplos del uso de operadores lógicos.

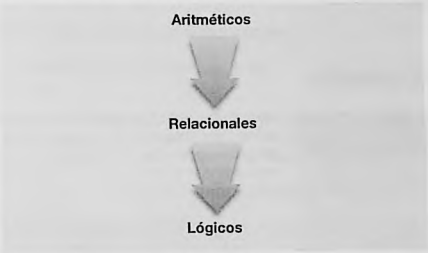
Ejemplo 1.6

!(25.3 > 8.3 * 2) → !(25.3 > 16.6) !(true) → al evaluarse da **false**

12.5 – 3.2 == 9.3 && 23.0 < Math.pow (8, 3) → 9.3 == 9.3 && 23.0 < Math.pow (8, 3) → true && 23.0 < Math.pow (8, 3) → true && 23.0 < 512 → true && true → al evaluarse da **true**

16.5 > 8.3 || Math.sqrt (30.25) > 7.5 → true || Math.sqrt (30.25) > 7.5 → true || 5.5 > 7.5 → true || false → al evaluarse da **true**

En aquellas expresiones que combinen operadores de los tres tipos vistos, se ejecutan primero los aritméticos, posteriormente los relacionales y por último los lógicos, tal como se vio en los ejemplos previos.



En las siguientes secciones veremos posibles usos de los operadores lógicos, resaltando la importancia que tienen para la creación de expresiones complejas.

Programas en Java

Java es un lenguaje orientado a objetos. Este paradigma de programación es tema de estudio del siguiente capítulo. Por ahora, diremos que un programa es una secuencia de instrucciones escritas usando el lenguaje Java.

Declaración de variables

En Java se deben declarar —indicar qué tipo de dato pueden almacenar— todas las variables antes de usarse. Se sigue la siguiente sintaxis:

```
tipo nombreVariable;
```

donde tipo es cualquiera de los tipos de datos válidos en Java y nombreVariable es el nombre que se le da a la variable. La declaración termina con punto y coma. Observe los siguientes ejemplos:

Ejemplo 1.7

```
int num, edad;  // Se declaran dos variables capaces de almacenar números enteros.
double promedio; // Promedio podrá almacenar un número de doble precisión.
String nombre; // La variable nombre podrá almacenar una cadena de caracteres.
char carácter; // La variable carácter podrá almacenar un solo carácter.
```

A partir de que la variable se declara, se reserva el espacio de memoria necesario para almacenar datos compatibles con el tipo dado.

Declaración de constantes

Para declarar una constante se usa la palabra final. Así se indica que la misma no podrá ser reasignada. Como en el caso de las variables, se debe fijar el tipo de dato y además se le debe asignar un valor, el cual ya no podrá cambiar durante la ejecución del programa. Se sigue la siguiente sintaxis:

```
final tipoDato NOMBRE_CONSTANTE = valor;
```

Ejemplo 1.8

```
final double PI = 3.14159265;
final double DISTANCIA_MEDIA_LUNA = 384400;
final String NAVIDAD = "25 de diciembre";
```

Comentarios

Un comentario es información que se escribe en un programa con el objetivo de documentar el código. Es decir, los comentarios no se ejecutan, sino que están dirigidos al lector. Son muy útiles para verificar código y para darle mantenimiento. En Java existen dos maneras de escribir comentarios:

```
// Se usa para comentarios que ocupan un solo renglón.
/* Para comentarios que ocupan más de un renglón. Por ejemplo, la descripción
 * general del programa que se está documentando, nombre del autor, la fecha, etc.
 */
```

1.3 ALGORITMOS Y PROGRAMAS

Un **algoritmo** se define como una secuencia de pasos ordenada que permite alcanzar un resultado. En general, en todo algoritmo se distinguen los datos (entrada), el proceso y los resultados (salida).

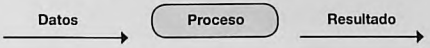


Figura 1.3 Algoritmo

- Los datos hacen referencia a toda la información necesaria para resolver el problema.
- El proceso hace referencia a las operaciones que se aplican a los datos.
- El resultado es el objetivo a alcanzar, es decir, qué se quiere lograr al aplicar el proceso a los datos.

Por ejemplo, en un algoritmo para preparar gelatina de almendras:

- los datos son todos los ingredientes (almendras, leche, grenetina, azúcar, etc.)
- el proceso son todos los pasos y el orden en el cual se deben ejecutar: 1) disolver la grenetina en 60 ml de leche fría; 2) en un procesador de alimentos, moler las almendras con 100 gr de azúcar y medio litro de leche; 3) etcétera.
- el resultado es la gelatina lista para disfrutar.

Cuando el algoritmo se desarrolla para que lo ejecute una computadora (no un cocinero!), los pasos del proceso deben expresarse en un lenguaje entendible por la misma. Estos lenguajes reciben el nombre de **lenguajes de programación** y el algoritmo ya traducido recibe el nombre de **programa**.

Una vez planteado el problema, luego de su análisis, resulta conveniente expresar la solución en algún lenguaje intermedio antes de escribir el programa. Generalmente los algoritmos se escriben en **pseudocódigo** o se hace un **diagrama de flujo** que es una representación gráfica del algoritmo. El objetivo de este libro no es enseñar a programar, por lo que si el lector no está familiarizado con los diagramas de flujo, se recomienda consultar el libro de Osvaldo Cairó.[3]

Buenas prácticas

- ☑ Leer con cuidado el enunciado del problema.
- ☑ Analizar cuidadosamente: ¿de qué datos disponemos?, ¿qué se espera como resultado?, ¿cuáles son los pasos y en qué orden deben llevarse a cabo para alcanzar el resultado?
- ☑ Escribir el algoritmo en pseudocódigo o diagrama de flujo.
- ☑ Verificar que la solución planteada sea la correcta.
- ☑ Buscar alternativas para una solución más eficiente.
- ☑ Programar el algoritmo, documentando el código generado.
- ☑ Probar el programa, buscando cubrir todos los casos que puedan presentarse.

[3] Cairó, Osvaldo. *Metodología de la programación. Algoritmos, diagramas de flujo y programas.* 3a. ed., Alfaomega, México, 2005.

Un archivo que contenga un programa escrito en Java, al menos en este capítulo, tendrá la siguiente forma:

```
[<declaración de paquete>]
[<declaración de "import">]
<declaración de clases>+  // El + indica 1 o más.
```

En el caso de NetBeans, que es el ambiente de desarrollo usado en este libro, el nombre del paquete se incluye automáticamente en el archivo que contendrá al programa (clase). Aparece con la palabra reservada package seguida del nombre del paquete y termina con punto y coma. En el capítulo 2 se retoma este tema.

Cuando se requiere importar una o varias clases de alguna de las librerías de Java o de algún paquete distinto del que se está usando, se utiliza la palabra reservada import seguida del nombre de la librería o paquete, luego punto y el nombre de la clase. Si se desea importar todas las clases de dicha librería, entonces luego del punto se pone asterisco. Considerando que puede haber paquetes definidos adentro de otros paquetes, se sigue la siguiente sintaxis:

```
import <nombre de paquete> [.<nombre de subpaquete>].<nombre de la clase>;
```

o bien:

```
import <nombre de paquete> [.<nombre de subpaquete>].*;
```

Ejemplo 1.9

```
import java.util.Scanner; // Se importa solo la clase Scanner de la librería útil de Java.
import java.io.*; // Se importan todas las clases de la libreria de Java.
```

El orden es muy importante: primero los paquetes, luego los archivos importados (import) y por último nuestras clases. En general, un programa en Java se verá así:

```
package nombrePaquete;  // Nombre del paquete en el cual se está trabajando

// Si se requiere importar una clase de Java:

import java.nombreLibreríaJava.NombreClase;

// Si se requiere importar todas las clases de la librería:

import java.nombreLibreríaJava.*;

public class NombrePrograma { // Nombre del programa (clase)

        public static void main (String arg[]){
```

```
            // Declaración de variables
            /* Cuerpo del programa: conjunto de instrucciones escritas en el lenguaje
            * Java, correspondientes al algoritmo diseñado.
            */
        }
}
```

Observe que en la estructura anterior se dejaron sangrías y renglones en blanco para tener un código más legible y, por lo tanto, más mantenible. En las siguientes secciones veremos con mayor detalle las instrucciones que pueden formar parte de un programa en Java.

1.3.1 Impresión, lectura y asignación

» **Impresión de resultados**

Es la operación que permite mostrar en pantalla el resultado obtenido y cualquier información dirigida al usuario. En la figura 1.4 se muestra la representación gráfica de la impresión por medio del símbolo usado en los diagramas de flujo. Se puede imprimir un mensaje, el cual se encierra entre comillas, y/o el contenido de una variable, de la cual se pone su nombre. Si fueran más de una, se separan por comas.

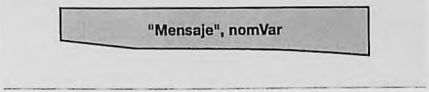


Figura 1.4 Impresión

En Java la instrucción usada es:

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

En el primer caso se muestra en pantalla lo indicado entre los paréntesis y el cursor permanece en la misma línea; mientras que en el segundo, luego de desplegar la información, el cursor se baja a la siguiente línea.

Ejemplo 1.10

```
System.out.print("Ingrese total de empleados: ");
/* Imprime el mensaje y el cursor se queda en la misma línea. Es conveniente usar esta
 * forma precediendo a una lectura de datos, como se verá un poco más adelante.
 */

System.out.println("El promedio de calificaciones es: " + promedio);
/* Despliega en la pantalla el mensaje encerrado entre comillas, seguido del valor que
 * almacene la variable promedio. Posteriormente, el cursor baja a la siguiente línea.
 */
```

Se puede usar el carácter de control "\n" para producir un salto de línea en la posición en la cual se agregue. Retomando el ejemplo anterior:

```
System.out.println("\nEl promedio de calificaciones es: " + promedio);
/* Baja un renglón antes de desplegar en la pantalla el mensaje, el cual irá seguido del
 * valor que almacene la variable promedio. Posteriormente, el cursor baja a la siguiente
 * línea.
 */
```

Cuando se requiera controlar el formato con el cual se imprime, se puede usar la instrucción:

```
System.out.printf("formato", nomVariable);
```

Donde: *formato* indica la forma en la cual se va a imprimir el contenido de nomVariable. Analice los siguientes ejemplos. En forma de comentarios, se proporciona una explicación junto a cada uno de ellos. También se puede probar el código almacenado en el archivo PruebasCapítulo1.java, subprograma *ejemplosImpresión ConFormato()*.

```
// Impresión con formato de números tipo float y double
double real = Math.sqrt (17.5);
/* Imprime ocupando 6 espacios en total, con 3 dígitos de precisión.
 * El punto ocupa un espacio. Justifica hacia la derecha.
 */
System.out.printf("%6.3f ", real);

// Impresión con formato de números tipo int, short, long y byte
int entero = 16305;
// Imprime ocupando 8 espacios. Justifica hacia la derecha.
System.out.printf("\n%8d ", entero);

// Imprime ocupando 8 espacios. Justifica hacia la izquierda.
System.out.printf("\n%-8d ", entero);

// Imprime ocupando 8 espacios. Agrega signos + o - al número.
System.out.printf("\n%+8d ", entero);

// Imprime ocupando 8 espacios. Completa con 0 los espacios faltantes.
System.out.printf("\n%08d ", entero);

// Impresión con formato de cadenas de caracteres.
String cadena ="Esta es una prueba";

// Con la S todas las letras de la cadena se imprimen en mayúsculas.
System.out.printf("\n%S ", cadena);

// Con la s se imprimen los caracteres como están.
System.out.printf("\n%s ", cadena);
```

```
// Imprime, ocupando 10 espacios, los primeros 6 caracteres de la cadena.
System.out.printf("\n%10.6s ", cadena);

// Impresión con formato de un carácter.
char letra = 'a';

// Con la C se imprime la letra mayúscula. En este caso la letra A.
System.out.printf("\n%C ", letra);

// Con la c se imprime la letra como está. En este caso la letra a.
System.out.printf("\n%c ", letra);
```

» Entrada de datos

También conocida como lectura de datos, es la operación que permite ingresar los datos que serán procesados para lograr el resultado buscado. La lectura implica una asignación, ya que el dato que se ingresa se asigna a la variable involucrada en la lectura. La representación mediante diagrama de flujo aparece en la figura 1.5. Con nomVar se hace referencia a la variable que se está leyendo. Si fueran más de una, se separan por comas.

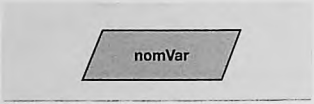


Figura 1.5 Lectura

En Java se usa la clase Scanner para leer datos desde el teclado. Dicha clase se importa de la biblioteca *util* (observe el programa 1.1). Se requiere la siguiente instrucción:

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

Con esta instrucción se declara un objeto de la clase Scanner (llamado variableLectura), el cual se usará posteriormente para leer los datos ingresados por el usuario desde el teclado. Los conceptos de clase y de objeto serán tratados con más detalle en el capítulo siguiente. Para leer los datos se pueden usar algunos de los métodos que se muestran a continuación:

```
variableEntera = variableLectura.nextInt( );
// Lee un número entero y lo asigna a variableEntera.

variableReal = variableLectura.nextDouble( );
// Lee un número de doble precisión y lo asigna a variableReal.

variableCadena = variableLectura.next( );
/* Lee una cadena de caracteres hasta que encuentra un espacio en blanco y la asigna a
* variableCadena.
*/

variableCadena = variableLectura.nextLine( );
/* Lee una cadena de caracteres hasta que encuentra el carácter de control
*  correspondiente al retorno (enter) y la asigna a variableCadena.
*/
variableLógica = variableLectura.nextBoolean( );
// Lee un valor booleano o lógico (true/false) y lo asigna a variableLógica.
```

Ejemplo 1.11

```
Scanner lee = new Scanner(System.in);
int claveAlumno;
double calif;
String nombreAlumno;

System.out.print("\nIngrese nombre: ");
nombreAlumno = lee.nextLine( );
/* Usamos nextLine( ) para leer nombres completos: nombre, apellido paterno
 * y apellido materno.
 */
```

```
System.out.print("Ingrese clave del alumno: ");
claveAlumno = lee.nextInt();

System.out.print("\nIngrese calificación: ");
calif = lee.nextDouble();
```

En ciertos casos, los datos pueden encontrarse almacenados en un archivo. Por lo tanto, la entrada debe hacerse desde el archivo y no desde el teclado. Para leer datos desde un archivo se usan las clases File y Scanner de Java. Con la primera se crea un objeto que se asocia al archivo físico que se quiere leer y, posteriormente, se indica que la lectura a través del Scanner se hará desde el archivo creado. Para usar la clase File se debe importar desde la librería *io* de Java.

La sintaxis usada es:

```
File nomVarArch = new File (nomArchivo);

Scanner nomVar = new Scanner (nomVarArch);
```

Al momento de crear el archivo y asociarse la variable (nomVarArch) con el archivo existente (nomArchivo) se puede provocar un error si el archivo no existe o está ubicado en otra dirección. En Java se debe manejar la excepción, y la manera de hacerlo será tema de una de las siguientes secciones.

Una vez declaradas las variables, la lectura se realiza con los mismos métodos empleados para la lectura desde el teclado. Existen métodos que resultan muy útiles para la lectura de archivos de los cuales no conocemos cuántos datos almacenan. Algunos de ellos son:

```
// Si existe un dato tipo double en el archivo, regresa true.
hasNextDouble()
// Si existe un dato tipo int en el archivo, regresa true.
hasNextInt()

/* Si existe otro dato en el archivo, regresa true. Generalmente se usa para cadenas o
 * caracteres.
 */
hasNext() o hasNextLine()
```

Por último, una vez que se ha terminado de leer los datos del archivo, es necesario cerrarlo. Para ello se utiliza el método close(), aplicando la sintaxis:

```
nomVar.close ();
```

» Asignación de valor a variables

Es la operación que permite darle un valor a una variable. En el momento que la variable recibe el nuevo valor, pierde cualquier otro valor que pudiera haber tenido. En los diagramas de flujo esta operación se representa como muestra la figura 1.6. Indica que a nomVar se le asigna el resultado de la expresión, pudiendo ser ésta una constante, otra variable o una operación que al ser evaluada da un valor compatible con el tipo de la variable.

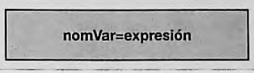


Figura 1.6 Asignación

En Java se usa el signo = para indicar una asignación. La variable que está en el lado izquierdo del signo igual debe estar previamente declarada, y las variables que se usen en el lado derecho deben estar declaradas y deben tener algún valor asignado.

Ejemplo 1.12

```
promedio = (cal1 + cal2 + cal3) / 3;
// En la variable promedio se asigna el resultado de la suma de los 3 números entre 3.

suma = suma + número;
/* Para que la variable suma se pueda usar en el lado derecho de la expresión requiere
 * que previamente se le haya asignado un valor. El resultado de la operación se asigna a
 * suma, la cual pierde su valor anterior.
 */

nombreAlumno = "Diego Ricalde";
// nombreAlumno es una variable tipo String. El valor se encierra entre comillas.


letra = 'n';
// letra es una variable tipo char. El valor se encierra entre comilla simple.

respuesta = true;
// respuesta es una variable tipo booleano.
```

» Operadores especiales

En Java existen unos operadores especiales que en ciertas situaciones pueden ser muy útiles. Dos muy usados son: ++ y --. El primero de ellos indica incremento (suma 1) y el segundo decremento (resta 1). Además, dependiendo del orden en que acompañen a la variable será el orden en el cual se realiza la operación.

- Siguiendo a la variable: variable++ o variable--

  La variable se usa y posteriormente, se incrementa o decrementa.

- Precediendo a la variable: ++variable o – –variable

  La variable se incrementa o decrementa, luego se usa.

Observe los ejemplos que aparecen más abajo. Los mismos pueden probarse usando el archivo PruebasCapítulo1.java, subprograma: *ejemplosOperadoresEspeciales()*.

```
n = 2;
System.out.println("n: " + n); // Imprime 2, el valor asignado

resultado = n++; // Asigna el valor de n (2) a resultado y luego incrementa n (3)
System.out.println("n: " + n + " - resultado: " + resultado);
// Imprime 3 (la n incrementada) y 2 (el valor asignado a resultado)

resultado = ++n; // Incrementa n (4) y luego la asigna a resultado (4)
System.out.println("n: " + n + " - resultado: " + resultado);
// Imprime 4 (la n incrementada) y 4 (el valor asignado a resultado)

resultado = n--; // Asigna el valor de n (4) y luego decrementa n (3)
System.out.println("n: " + n + " - resultado: " + resultado);
// Imprime 3 (la n decrementada) y 4 (el valor asignado a resultado)

resultado = --n; // Decrementa n (2) y luego la asigna a resultado (2)
System.out.println("n: " + n + " - resultado: " + resultado);
// Imprime 2 (la n decrementada) y 2 (el valor asignado a resultado)
```

Otros operadores especiales directamente relacionados con la asignación son: +=, -=, *=, /=, %=. Con estos operadores primero se ejecuta la operación indicada antes del = y luego se asigna. Observe los siguientes ejemplos. Los mismos pueden probarse usando el archivo PruebasCapítulo1.java, subprograma: *ejemplosAsignacionesEspeciales()*.

```
int n1, n2;

n1 = 3;
n2 = 2;
System.out.println("n1: " + n1 + " - n2: " + n2);
// Imprime 3 y 2, que son los valores asignados a las variables

n1 += n2; // Equivale a n1= n1 + n2
System.out.println("n1: " + n1 + " - n2: " + n2); // Imprime 5 y 2

n1 -= n2; // Equivale a n1= n1 - n2
System.out.println("n1: " + n1 + " - n2: " + n2); // Imprime 3 y 2

n1 *= n2; // Equivale a n1= n1 * n2
System.out.println("n1: " + n1 + " - n2: " + n2); // Imprime 6 y 2

n1 /= n2; // Equivale a n1= n1 / n2
System.out.println("n1: " + n1 + " - n2: " + n2); // Imprime 3 y 2

n1 %= n2; // Equivale a n1= n1 % n2
System.out.println("n1: " + n1 + " - n2: " + n2); // Imprime 1 y 2
```

» Variables que funcionan como contadores o acumuladores

Ciertas variables, por la función que desempeñan, reciben el nombre de contador o de acumulador. Se le llama *contador* cuando se usa para contar. Por ejemplo, contar cuántas calificaciones hay mayores que 6. Por su parte, el término *acumulador* se utiliza para indicar que en la variable se acumulan/almacenan resultados parciales para obtener un resultado final. Por ejemplo, una variable en la que se vayan sumando calificaciones para posteriormente calcular el promedio de las mismas.

En la figura 1.7 se presenta el diagrama de flujo correspondiente a la solución de un problema simple: se leen las calificaciones de 3 exámenes parciales y la calificación del examen final. Se calcula el promedio final como el promedio entre la calificación del final con el promedio de los parciales. Finalmente se imprime el resultado obtenido. El diagrama incluye varios de los conceptos vistos: lectura, asignación, expresiones aritméticas e impresiones.

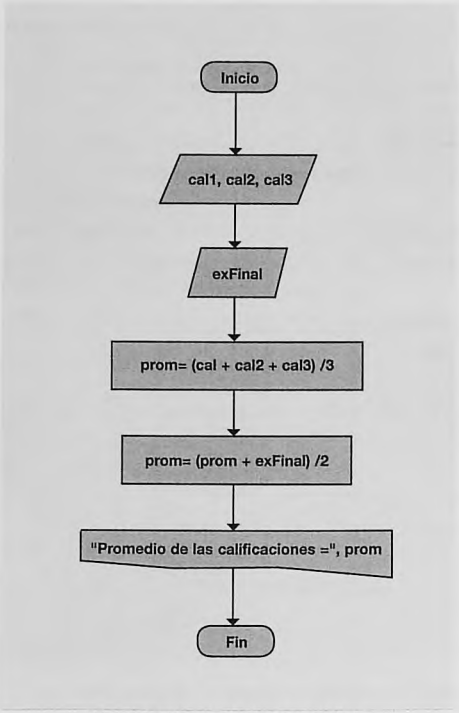


Figura 1.7 Ejemplo de diagrama de flujo

En el diagrama de la figura 1.7, además de los símbolos vistos para cada una de las operaciones estudiadas, se usan elipses para indicar el inicio y el fin del diagrama. En ambos casos se etiqueta, según corresponda, con las palabras Inicio (o Principio) y Fin (o Termina) respectivamente.

En el programa 1.1 se presenta el programa correspondiente al diagrama de la figura 1.7. En él se observan todos los elementos explicados más arriba.

Programa 1.1 **CalculaPromedioCalif.java**

```
package cap1;
import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Ejemplo de un programa en Java.
 * Programa 1.1
 */
public class CalculaPromedioCalif {

        public static void main(String[] args) {

                // Declaración de variables
                double cal1, cal2, cal3, exFinal, prom;
                Scanner lee = new Scanner(System.in);

                // Lectura de datos
                System.out.println("Ingrese las 3 calificaciones: ");
                cal1 = lee.nextDouble( );
                cal2 = lee.nextDouble( );
                cal3 = lee.nextDouble( );
                System.out.println("Ingrese la calificación del final: ");
                exFinal = lee.nextDouble( );

                /* Proceso: se calcula el promedio y se asigna el resultado
                * a la variable prom. Uso de ( ) para indicar que debe hacerse primero
                * la suma y luego la división.
                */
                prom = (cal1 + cal2 + cal3) / 3;
                prom = (prom + exFinal) / 2;

                // Salida de resultado
                System.out.println("\nPromedio de las calificaciones: " + prom + "\n");
        }
}
```

Observe que el código del programa 1.1 se corresponde al diagrama de la figura 1.7. Cada una de las operaciones del diagrama se tradujo usando el lenguaje Java y generando así el programa. En este último se agregó la declaración de las variables (que en el diagrama no es necesaria) y algunos comentarios que ayudan a comprender el código.

» Conversión de datos

En Java se puede forzar un dato o una expresión a cambiar de forma. Para ello se utiliza la siguiente sintaxis:

(nuevo tipo) variable o expresión

Ejemplo 1.13

```
double prom, número1;
int suma, n, número2;
...
prom = (double) suma/n;
/* El resultado de la división de dos enteros (la cual, en Java, da un entero) se convierte al tipo
de doble precisión. */

n = (int) Math.sqrt(17);
// Convierte a entero el resultado de la raíz cuadrada de 17. Se pierden los decimales.

número1 = 4.1823;
número2 = (int) número1;
// número 2 se queda con la parte entera de número1, en este caso con el valor 4
```

» Métodos de la clase *String*

Con respecto a los datos tipo String, se pueden aplicar métodos para cuando se requiera trabajar con ellos. Los métodos se invocan en el contexto requerido según el tipo de resultado que den y siguiendo la sintaxis:

nombreVariable.método([parámetros si los tuviera])

Donde:

- nombreVariable es una cadena tipo String
- método es el nombre del método que se quiere aplicar sobre la variable

Algunos de los métodos[4] más usados se presentan a continuación. Entre () se indica el tipo de resultado obtenido.

length( ) → regresa el total de caracteres que tiene la cadena (entero)

charAt(i) → regresa el carácter que ocupa la posición i de la cadena, considerando que los caracteres se enumeran del 0 en adelante (carácter)

equals(cad) → regresa true si cad es igual a la cadena que invoca el método (booleano)

compareTo(cad) → regresa un número positivo, negativo o cero, si la cadena es mayor, menor o igual a cad (entero)

indexOf(cad) → regresa la posición donde se encuentra la primer ocurrencia de cad dentro de la cadena que invocó al método (entero)

toUpperCase( ) → convierte la cadena a mayúsculas (cadena)

toLowerCase( ) → convierte la cadena a minúsculas (cadena)

concat(cad) → concatena (pega) a la derecha de la cadena que invoca el método el contenido de cad (cadena)

Con el operador + se puede lograr el mismo resultado que con el método concat. Se utiliza de la siguiente manera:

```
cadena = cadena + otraCadena;

o

cadena = cadena + "constante tipo cadena";
```

Ejemplo 1.14

Los mismos pueden probarse usando el archivo PruebasCapítulo1.java, subprograma: *ejemplosString( )*.

```
String nombreAlumno = "Diego Ricalde";
String cadena;
int n, res, pos;
char car;
boolean resp;

System.out.println("\nDesde ejemplosString:");
n = nombreAlumno.length( ); // A n se le asigna el valor 13
System.out.println("n: " + n);
```

[4] La lista completa de los métodos puede encontrarse en: http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/String.html

```
car = nombreAlumno.charAt(0); // A car se le asigna la letra D
System.out.println("car: " + car);
resp = nombreAlumno.equals("Diego Martínez"); // A resp se le asigna false
System.out.println("resp: " + resp);

res = nombreAlumno.compareTo("Diego Martín"); //A res se le asigna un valor positivo
System.out.println("res: " + res);
pos = nombreAlumno.indexOf("Ricalde"); // A pos se le asigna el valor 6
System.out.println("pos: " + pos);

System.out.println(nombreAlumno.toUpperCase( )); // Imprime DIEGO RICALDE
System.out.println(nombreAlumno.toLowerCase( )); // Imprime diego ricalde

nombreAlumno = nombreAlumno.concat(" Gutiérrez");
System.out.println(nombreAlumno);  // Imprime Diego Ricalde Gutiérrez

cadena = "Hola " + nombreAlumno + "\n";
// Imprime Hola Diego Ricalde Gutiérrez (y salta a la siguiente línea)
System.out.println(cadena);
```

1.3.2 Estructuras algorítmicas selectivas

Son las instrucciones que permiten bifurcar el flujo de la solución de un problema. Es decir, en un algoritmo, además de lecturas, asignaciones e impresiones se necesita indicar que, ante ciertas situaciones, se siguen conjuntos distintos de instrucciones.

Retomando el algoritmo para preparar gelatina de almendras, en el proceso se debe indicar que si tiene esencia de almendras, se deben agregar 5 gotas.

Java ofrece tres estructuras selectivas distintas: simple, doble y múltiple. El ejemplo anterior es un caso de una selección simple.

» Selección simple

Usada para indicar que sólo en caso de que una cierta condición se cumpla (V), entonces se debe proceder a ejecutar un determinado conjunto de instrucciones y, posteriormente, seguir con el flujo de instrucciones

con el cual se seguiría también en el caso que la condición no se cumpliera (F). En un diagrama de flujo, esta estructura se representa como se muestra en la figura 1.8.

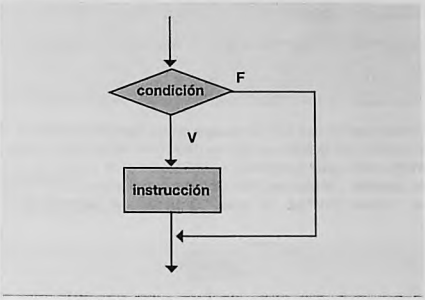


Figura 1.8 Estructura selectiva simple

La condición puede ser cualquier expresión relacional simple o varias expresiones relacionales combinadas por medio de operadores lógicos.

En Java se escribe como:

```
if (condición)
    instrucción;
```

- La *condición* siempre se escribe entre ().
- Si *instrucción* fuera más de una, entonces se encierran entre { }.
- Es conveniente dejar sangría debajo de la palabra if.

Ejemplo 1.15

```
if (totalAlum > 0)
        promedio = suma / totalAlum;

if (exFinal >= 6 && promParciales >= 6)
        System.out.println("El alumno cumple con los requisitos para aprobar la materia");
```

```
if (totalAlum > 0) {
        promedio = suma / totalAlum;
        System.out.println("El promedio de las calificaciones es: " + promedio);
}
```

Se retoma el diagrama de flujo de la figura 1.7 y se le agrega una nueva funcionalidad: determinar si un alumno aprueba una materia. El criterio que se aplica es que se debe tener el promedio de los parciales aprobatorio y que el examen final también debe estar aprobado. La calificación de la materia se obtiene como el promedio entre el promedio de los parciales y el examen final. Observe que si el alumno no cumple con las condiciones planteadas, entonces el proceso concluye. La figura 1.9 presenta el diagrama de flujo con la solución del problema planteado.

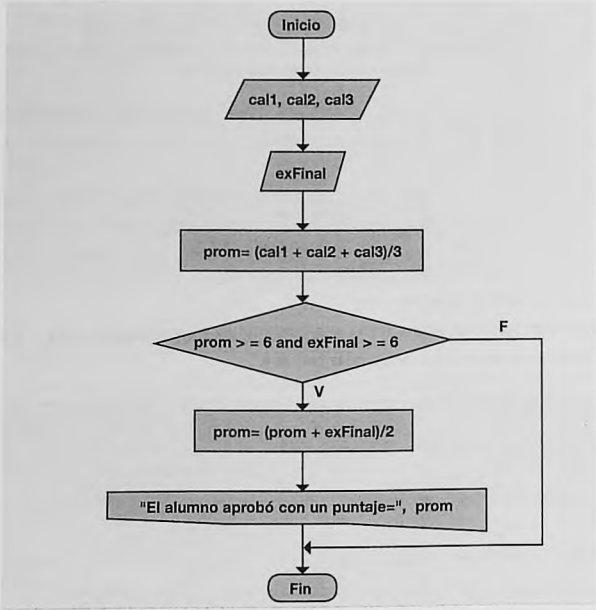


Figura 1.9 Diagrama de flujo con estructura selectiva simple

En el programa 1.2 se presenta el código en Java correspondiente al diagrama de la figura 1.9.

Programa 1.2 **UsoSelecciónSimple.java**

```
package cap1;
import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Ejemplo de uso de una selección simple (if)
 * Programa 1.2
 */
public class UsoSelecciónSimple {

        public static void main(String[] args) {
                // Declaración de variables
                double cal1, cal2, cal3, exFinal, prom;
                Scanner lee = new Scanner(System.in);

                // Lectura de datos
                System.out.println("Ingrese las calificaciones de los 3 parciales: ");
                cal1 = lee.nextDouble( );
                cal2 = lee.nextDouble( );
                cal3 = lee.nextDouble( );

                System.out.println("Ingrese la calificación del examen final: ");
                 exFinal = lee.nextDouble( );

                // Calcula promedio
                prom = (cal1 + cal2 + cal3) / 3;
```

```
        /* Evalúa que tanto el promedio de parciales como el examen final hayan
         * sido aprobados. En caso afirmativo, se imprime la calificación final.
        */
        if (prom >= 6 && exFinal >= 6){
                prom = (prom + exFinal) / 2;
                System.out.println("\nEl alumno aprobó con un puntaje = " + prom + "\n");
        }
    }
}
```

» Selección doble

Usada para indicar que, en caso de que una cierta condición se cumpla (V), se debe proceder a ejecutar un determinado conjunto de instrucciones, y en caso contrario (F) se deberá ejecutar un conjunto distinto de instrucciones. Es decir, al evaluar la condición hay una doble bifurcación del flujo: uno para cuando la condición es verdadera y otro para cuando la condición es falsa. En un diagrama de flujo, esta estructura se representa como se muestra en la figura 1.10.

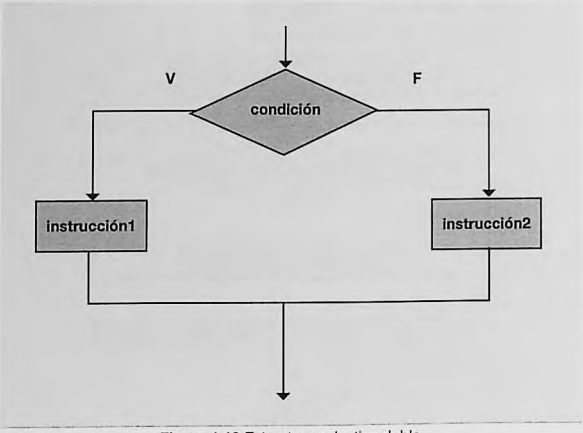


Figura 1.10 Estructura selectiva doble

La condición puede ser cualquier expresión relacional simple o varias expresiones relacionales combinadas por medio de operadores lógicos.

En Java se escribe como:

```
if (condición)
        instrucción1;
else
        instrucción2;
```

- La *condición* siempre se escribe entre ().
- Si *instrucción1* o *instrucción2* fuera más de una, entonces se encierran entre { }.
- Si *condición* resulta verdadera (V), entonces se ejecuta *instrucción1*.
- Si *condición* resulta falsa (F), entonces se ejecuta *instrucción2*.
- Es conveniente dejar sangría debajo de las palabras if y else.

Ejemplo 1.16

```
if (totalAlum > 0)
        promedio = suma / totalAlum;
else
        System.out.println("No hay alumnos, no es posible calcular el promedio");


if (totalAlum > 0) {
        promedio = suma / totalAlum;
        System.out.println("El promedio de las calificaciones es: " + promedio);
}
else
        System.out.println("No es posible calcular el promedio");
```

En el programa 1.3 se presenta un ejemplo de selección doble. Retomamos el programa 1.2 pero ahora, en caso de que el alumno no apruebe la materia, se imprimirá un mensaje adecuado.

Programa 1.3 **UsoSelecciónDoble.java**

```
package cap1;
import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Ejemplo de uso de una selección doble (if/else)
 * Programa 1.3
 */
public class UsoSelecciónDoble {

        public static void main(String[ ] args) {

                // Declaración de variables
                double cal1, cal2, cal3, exFinal, prom;
                 Scanner lee = new Scanner(System.in);

                // Lectura de datos
                System.out.println("Ingrese las calificaciones de los 3 parciales: ");
                cal1 = lee.nextDouble( );
                cal2 = lee.nextDouble( );
                cal3 = lee.nextDouble( );

                System.out.println("Ingrese la calificación del examen final: ");
                exFinal = lee.nextDouble( );

                 // Calcula promedio
                prom = (cal1 + cal2 + cal3) / 3;
```

```
            /* Evalúa que tanto el promedio de parciales como el examen final hayan
            * sido aprobados. Si es así, se imprime la calificación final. En caso
            * contrario se imprime un mensaje.
            */
            if (prom >= 6 && exFinal >= 6){
                  prom = (prom + exFinal) / 2;
                  System.out.println("\nEl alumno aprobó con un puntaje = " + prom + "\n");
            }
            else
                  System.out.println("El alumno no aprobó la materia\n");
      }
}
```

» Selección múltiple

Usada para múltiples flujos de operaciones. Se evalúa una variable (o una expresión), llamada selector, y, según el valor obtenido, se elige el flujo correspondiente. El selector puede ser un número entero o un valor tipo carácter. En un diagrama de flujo, esta estructura se representa como se muestra en la figura 1.11. En cada una de las ramas se indican los posibles valores que puede tomar el selector. Así, si el selector toma el valor1, se ejecutará la instrucción1; si toma el valor2, se ejecutará la instrucción2, y así sucesivamente. La última rama (por omisión) es opcional. Se usa cuando se necesita decir qué hacer en caso de que el selector no tome ninguno de los valores considerados en los demás casos.

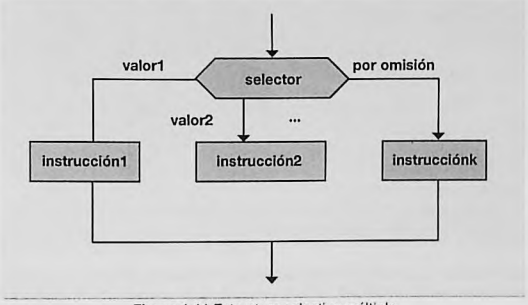


Figura 1.11 Estructura selectiva múltiple

El selector puede ser una variable tipo int o char, o cualquier expresión que, al ser evaluada, produzca un valor tipo int o char.

En Java se escribe como:

```
switch (selector) {
        case valor1: instrucción1;
                        break;
        case valor2: instrucción2;
                        break;
          ...
        default: instrucciónk;
}
```

- El *selector* siempre se escribe entre ().
- Si *instrucción1*, *instrucción2*, ..., *instrucciónk* fueran más de una, NO se requiere encerrarlas entre { }.
- Cada instrucción o cada bloque de instrucciones correspondientes a cada uno de los casos termina con un *break*. Tal directiva permite que, si el selector resultó igual a alguno de los casos, ya no se siga evaluando.
- Es conveniente dejar sangría debajo de la palabra switch y en cada caso.

Ejemplo 1.17

```
switch (categ) {
        case 1: aumento = 0.10;
                break;
        case 2: aumento = 0.08;
                break;
        case 3: aumento = 0.06;
                break;
        default: aumento = 0.03;
}
```

En este ejemplo se supone que categ es una variable tipo int. Si su valor es 1, entonces se asigna 0.1 a *aumento*. Si fuera 2, se le asigna el valor 0.08 y, en el tercer caso, el valor 0.06. Para cualquier otro valor se asigna 0.03.

```
switch (num % 4) {
        case 0: y = Math.pow(x, 3);
                break;
        case 1: y = Math.sqrt(x);
                break;
        case 2: y = Math.pow(x,2) / 23.5;
                break;
        case 3: y = Math.pow(5, 4) * 2.5;
                break;
}
```

En este ejemplo al hacer un número entero (num) módulo 4 se pueden obtener sólo 4 posibles valores. Es decir, el residuo de la división entera de un número entre 4 sólo puede ser 0, 1, 2 o 3. No se justifica tener la cláusula default en el switch.

```
switch (zona) {
        case 'E' :
        case 'O' :System.out.println("Diríjase a la sucursal Distrito Federal\n");
                break;
        case 'N' : System.out.println("Diríjase a la sucursal Monterrey\n");
                break;
        case 'S' : System.out.println("Diríjase a la sucursal Tuxtla G.\n");
                break;
        default:  System.out.println("No tenemos una sucursal en su zona.\n");
}
```

En el último ejemplo el selector es una variable tipo char. Se establece un flujo dependiendo del valor de zona, indicando que si el valor es E u O, entonces se imprimiría el mensaje: "Diríjase a la sucursal Distrito Federal". Es decir, es el mismo flujo para dos casos distintos.

En el programa 1.4 se presenta un ejemplo de selección múltiple. El problema consiste en calcular e imprimir el nuevo precio de un automóvil, considerando la categoría a la que pertenece. El programa recibe como datos el precio actual y la categoría del automóvil. Si la categoría es 10 o 20, el precio se incrementa en un 12%; si

la categoría es 30 el aumento es del 10% y si la categoría es 40 el aumento es del 8%. En cualquier otro caso, el incremento es del 5%.

Programa 1.4 **UsoSelecciónMúltiple.java**

```
package cap1;
import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Ejemplo de uso de una selección múltiple (switch).
 * El programa calcula e imprime el precio actualizado de un automóvil, de
 * acuerdo con la categoría del mismo.
 * Programa 1.4
 */
public class UsoSelecciónMúltiple {

        public static void main(String[ ] args) {

            // Declaración de variables
            double precioActual, precioFinal;
            int categoría;
            Scanner lee = new Scanner(System.in);

            // Lectura de datos
            System.out.print("\nPrecio actual del automóvil: ");
            precioActual = lee.nextDouble( );
            System.out.print("\nCategoría del automóvil: ");
            categoría = lee.nextInt( );

            switch (categoría) {
```

```
                    case 10:
                    case 20: precioFinal = precioActual * 1.12;
                            break;
                    case 30: precioFinal = precioActual * 1.10;
                            break;
                    case 40: precioFinal = precioActual * 1.08;
                            break;
                    default: precioFinal = precioActual * 1.05;
                }

                System.out.println("\nEl precio actualizado es: $" + precioFinal + "\n");

        }
}
```

1.3.3 Estructuras algorítmicas repetitivas

Son las instrucciones que permiten repetir una secuencia de instrucciones. En ciertas situaciones se necesita ejecutar una o más instrucciones varias veces. Para hacerlo se podrían volver a escribir, pero esta solución es muy limitada. Para lograr una solución general se usan las estructuras repetitivas o los ciclos.

Retomando el algoritmo para preparar gelatina de almendras, en el proceso se debe indicar que las almendras se muelen hasta obtener un polvo fino. La operación de "moler" se repite hasta alcanzar el resultado deseado.

Java ofrece tres estructuras repetitivas distintas: for, while y do-while. Cada una de ellas tiene características específicas que deben tenerse en cuenta al momento de usarlas.

Ciclo for: se utiliza cuando se conoce de antemano el número de veces que debe repetirse. El diagrama de flujo correspondiente al for se muestra en la figura 1.12. Se usa una variable de control (vC), la cual se inicializa con un valor inicial (valorInicial) y se incrementa hasta llegar al valor final (valorFinal). Por lo tanto, conociendo el valor inicial, el valor final y el incremento se puede determinar cuántas veces se ejecutará el ciclo.

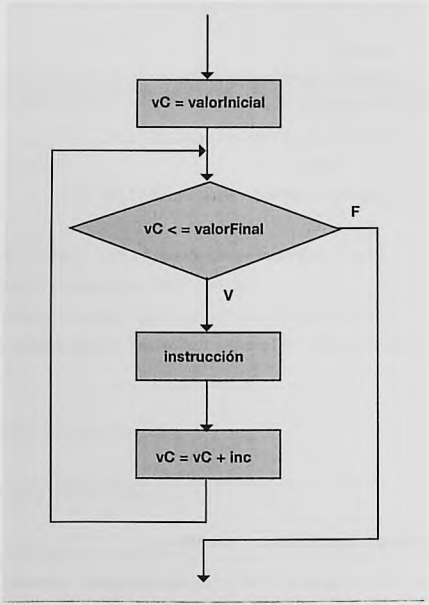


Figura 1.12 Estructura repetitiva *for*

En Java se escribe como:

```
for (vC = valorInicial; vC <= valorFinal; vC= vC + inc)
        instrucción;
```

- Si el *incremento* es 1, se usa el operador ++ (vC++).
- *instrucción* es la instrucción (conjunto de instrucciones), que se van a ejecutar repetidamente.
- Si *instrucción* es más de una, entonces se escriben entre {}.
- La vC, al terminar el ciclo, es igual a valorFinal + inc.
- Es conveniente dejar sangría debajo de la palabra for.

Una variante del for presentado es el for decreciente. En este caso el valor inicial es mayor que el valor final y la variable de control decrece (en lugar de incrementarse).

En Java se escribe como:

```
for (vC = valorInicial; vC > = valorFinal; vC = vC – dec)
        instrucción;
```

- Si el *decremento* es 1, se usa el operador – – (vC – –)
- La vC, al terminar el ciclo, es igual a valorFinal - dec

En el programa 1.5 se presenta un ejemplo de ciclo for. El problema es una generalización del programa 1.1 ya que se calcula e imprime el promedio de un grupo de calificaciones. La entrada es el total de calificaciones y cada una de las mismas. Las calificaciones, a medida que se leen, se acumulan en la variable *suma*. Posteriormente, si se leyó al menos una calificación, se calcula y se imprime el promedio. En caso contrario se imprime un mensaje. En esta solución se usa un ciclo y una decisión doble.

Programa 1.5 | **UsoCicloFor.java**

```
package cap1;
import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Ejemplo de uso del ciclo for
 * Programa 1.5
 */
public class UsoCicloFor {

        public static void main(String[] args) {
        Scanner lee = new Scanner(System.in);
        double cal, suma, prom;
        int totCal, i;

        System.out.print("\nIngresa total de calificaciones: ");
        totCal = lee.nextInt( );
```

```
        /* La variable suma debe iniciarse en 0, ya que se usará para acumular
        * las calificaciones y necesita tener un valor asignado cuando se
        * ejecute (por primera vez): suma + cal.
        */
        suma = 0;
        for (i = 1; i <= totCal; i++){
            System.out.print("\nCalificación " + i + ": ");
            cal = lee.nextDouble( );
            suma = suma + cal;
        }

        if (totCal > 0){
            prom = suma / totCal;
            System.out.println("\nEl promedio de las calificaciones es: " + prom +
            "\n");
        }
        else
            System.out.println("\nNo se puede calcular el promedio. \n");
        }
}
```

Ciclo while: se utiliza cuando se quiere repetir una o varias instrucciones cuyo número de veces depende del cumplimiento de una cierta condición. Retomando el ejemplo de la gelatina de almendras, se muelen *mientras* haya trozos grandes. Es importante que haya al menos una instrucción que altere la condición, si no se caería en un ciclo infinito (sin fin). Tales ciclos tienen la particularidad de que podrían no ejecutarse ni una vez si la condición tuviera un valor igual a *false* inicialmente. En la figura 1.13 se muestra su representación gráfica según los símbolos correspondientes a los diagramas de flujo.

En Java se escribe como:

```
while (condición)
    instrucción;
```

- La *condición* siempre se escribe entre ().
- La *instrucción* es la operación o el conjunto de operaciones que se quiere repetir.
- Si *instrucción* es más de una, se escriben entre { }.
- La *instrucción* se ejecuta mientras la condición sea verdadera.
- En *instrucción* debe haber al menos una instrucción que altere la condición, garantizando que se hará falsa. En caso contrario, se tendría un ciclo infinito.
- El ciclo podría no ejecutarse si *condición* es false inicialmente.
- Es conveniente dejar sangría debajo de la palabra *while*.

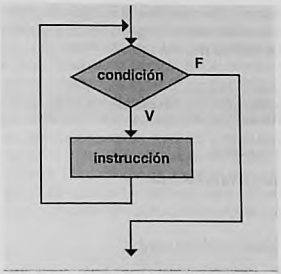


Figura 1.13 Estructura repetitiva *while*

En la figura 1.14 se presenta el diagrama de flujo que calcula e imprime la serie de Ullman. Esta serie se forma a partir de cualquier número entero positivo de la siguiente manera:

1. Si el número es par, se divide entre 2.
2. Si el número es impar, se multiplica por 3 y se le suma 1.
3. Se repiten los pasos 1 y 2 hasta que el número sea igual a 1.

Según el algoritmo dado, a partir de un número entero positivo se genera una secuencia de números (enteros y positivos) que siempre termina con el valor 1.

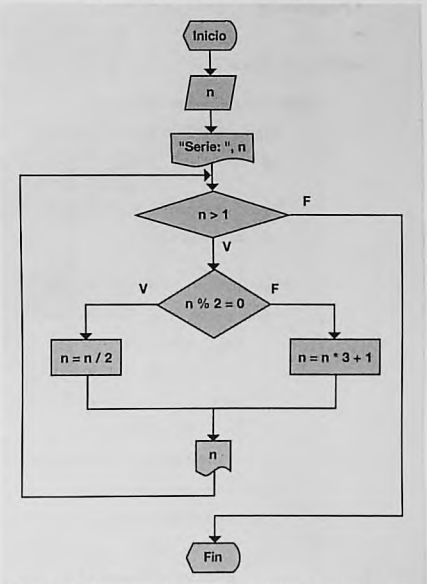


Figura 1.14 Diagrama de flujo para calcular la serie de Ullman

En el programa 1.6 se presenta el código correspondiente al diagrama de flujo anterior. Observe que, como en los casos anteriores, el programa resulta ser la traducción (utilizando Java) de la lógica expresada en el diagrama de flujo de manera gráfica.

Programa 1.6 **UsoCicloWhile.java**

```
package cap1;
import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Ejemplo de uso del ciclo while - Serie de Ullman
 * Programa 1.6
 */
public class UsoCicloWhile {

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

                System.out.print("Ingresa un número entero > 0: ");
                número = lee.nextInt();

                System.out.print("\n\nSerie de Ullman: " + número);
                while (número > 1){
                        if (número % 2 == 0)
                                número = número / 2;
                        else
                                número = número * 3 + 1;
                        System.out.print("-" + número);
                }
                System.out.println("\n\n");
        }
}
```

El ciclo while resulta muy útil cuando no se conoce el número de datos a procesar, y el fin de los mismos se indica por medio de un valor "que se distingue" de los demás. Por ejemplo, considere que se hizo una encuesta en la calle preguntando al encuestado el último nivel académico alcanzado. Los encuestadores registran la respuesta en un formulario diseñado para tal fin o en algún dispositivo electrónico. Al terminar el día, se deben procesar las respuestas. Esto se podría hacer sin saber cuántas encuestas en total se levantaron. Si las posibles respuestas son 1 (primaria), 2 (secundaria), 3 (preparatoria), 4 (licenciatura) y 5 (posgrado), se puede establecer que un -1 indica que ya no hay información a procesar. El -1, o su equivalente, reciben el nombre de *bandera de fin datos*. Usando un while se leerían y procesarían las respuestas *mientras* la respuesta leída sea mayor o igual a 1. En cuanto el usuario ingrese el -1, el proceso se detendría. El diagrama de flujo de la figura 1.15 muestra la lógica que se sigue, en general, en estos casos. Hay una lectura, luego el ciclo while en cuya condición aparece la variable leída y la última operación dentro del ciclo es otra lectura, valor con el que vuelve a evaluarse la condición. Entre la condición y la última lectura irían todas las operaciones que se deben repetir.

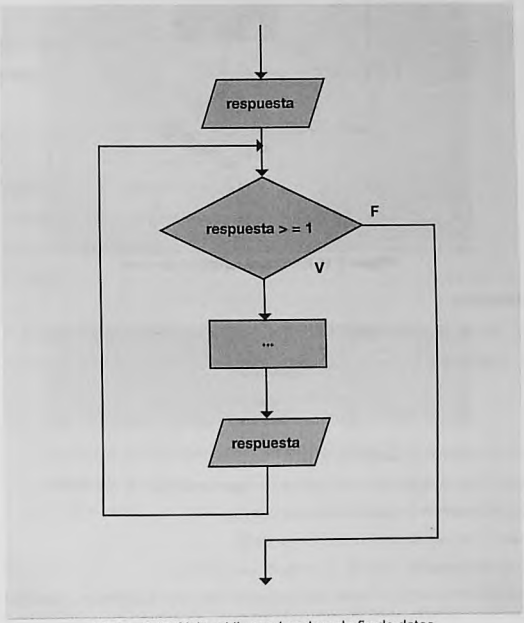


Figura 1.15 Ciclo *while* con bandera de fin de datos

Ciclo do-while: se utiliza cuando se quiere repetir una o varias instrucciones y el número de veces depende del cumplimiento de cierta condición. Retomando el ejemplo de la gelatina de almendras, se muelen *mientras* haya trozos grandes. La diferencia con el ciclo while es que en ése se evalúa la condición y luego se ejecuta la instrucción; mientras que en éste primero se ejecuta la instrucción y luego se evalúa. Por lo tanto, un do-while siempre se ejecuta al menos una vez. Es importante que el ciclo, lo mismo que en el while, tenga al menos una instrucción que altere la condición, si no se tendría un ciclo infinito (no tiene fin). La figura 1.16 muestra su forma gráfica, de acuerdo a los símbolos usados en los diagramas de flujo.

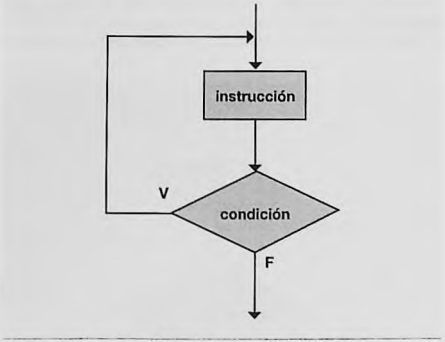


Figura 1.16 Estructura repetitiva *do-while*

En Java se escribe como:

```
do {
        instrucción;
} while (condición);
```

- La *condición* siempre se escribe entre ().
- La *instrucción* es la operación o el conjunto de operaciones que se quiere repetir.
- La *instrucción* siempre se ejecuta al menos una vez.
- Si *instrucción* es más de una, se escriben entre {}.
- La *instrucción* se ejecuta mientras la condición sea verdadera.
- En *instrucción* debe haber al menos una instrucción que altere la condición, garantizando que se hará falsa. En caso contrario, se tendría un ciclo infinito.
- En conveniente dejar sangría debajo de la palabra *do*.

En el programa 1.7 se presenta un ejemplo de ciclo do-while. Se retomó el programa 1.5 ahora usando un ciclo do-while para validar el total de calificaciones leído. Si se ingresa 0 o un valor negativo el ciclo se repetirá y volverá a pedir el dato. En caso de que el valor leído sea mayor que 0, entonces la condición del ciclo se hará falsa y se interrumpirá su ejecución. Al validar la entrada de esta manera, podemos suprimir la decisión doble usada para evitar una posible división entre cero.

Programa 1.7 | **UsoCicloDoWhile.java**

```
package cap1;
import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Programa 1.7
 */
public class UsoCicloDoWhile {

    public static void main(String[] args) {
    Scanner lee = new Scanner(System.in);
    double cal, suma, prom;
    int totCal, i;

    /* Al menos se debe leer una vez. Si se dan 1 o más calificaciones, se sale del
    * do-while. Si el usuario ingresa 0 o menos, entonces se vuelve a pedir el dato.
    */
     do {
          System.out.print("\nIngresa total de calificaciones: ");
          totCal = lee.nextInt();
    } while (totCal <= 0);

    suma = 0;
    for (i = 1; i <= totCal; i++){
```

```
                System.out.print("\nCalificación " + i + ": ");
                cal = lee.nextDouble( );
                suma = suma + cal;
            }

            prom = suma / totCal;
            System.out.println("\nEl promedio de las calificaciones es: " + prom + "\n");
        }
    }
```

1.3.4 Manejo de excepciones

Una excepción es una situación no deseada que se presenta durante la ejecución de un programa. Java permite el manejo de distintos tipos de excepciones por medio de ciertas clases, siendo la principal la clase Exception. Existe una jerarquía de clases que representan los distintos tipos de excepciones que pueden presentarse. De esta manera se pueden agrupar las excepciones por tema. Considerando que aún no se han visto los temas de clases y herencia de clases, en este capítulo se presentan los conceptos básicos necesarios para el manejo de excepciones, tratando todas las posibles excepciones con su tipo más general: Exception. Para una mayor información sobre las clases y sus métodos, dedicadas al manejo de excepciones, se sugiere consultar la documentación de Java.

Las clases, como se verá en el siguiente capítulo, cuentan con métodos que dan la funcionalidad a las mismas. La clase Exception de Java tiene varios métodos, entre los que destacan:

- getMessage(): regresa un mensaje relacionado con la excepción.
- toString(): regresa una descripción de la excepción formada por el nombre de la clase de la excepción, seguida por el resultado del método getMessage().
- printStackTrace(): imprime la secuencia de ejecución hasta el punto donde se presentó la excepción.

La instrucción utilizada para el manejo de las excepciones tiene la sintaxis:

```
try{
        instrucción;
} catch (Exception e){
        instrucción;
}
finally { //opcional
        instrucción;
}
```

- El bloque *try* es para las instrucciones que se van a ejecutar en una situación normal (sin errores). Puede estar formado por una o más instrucciones.
- El bloque *catch* permite manejar la excepción. Incluye las instrucciones que se quieren ejecutar en caso de que se produzca la excepción. La variable e es del tipo de la clase Exception y, por lo tanto, sobre ella se pueden aplicar los métodos mencionados.
- El bloque *finally* es opcional y se usa para indicar las instrucciones que se quieren ejecutar, se presente o no la excepción.

En el programa 1.8 se muestra un ejemplo en el cual el manejo de excepciones se utiliza para evitar un error en tiempo de ejecución por división entre cero.

Programa 1.8 | **ManejoExcepcionesExpArit.java**


```
package cap1;
import java.util.*;

/**
 * Ejemplo de manejo de excepciones en una expresión aritmética. Evita que una
 * división entre 0 produzca un error durante la ejecución.
 * @author Silvia Guardati
 * Programa 1.8
 */
public class ManejoExcepcionesExpArit {

    public static void main(String[] args) {
        int numerador, denominador, resultado;
        Scanner lee = new Scanner(System.in);

        System.out.print("\nIngrese numerador: ");
        numerador = lee.nextInt();
        System.out.print("Ingrese denominador: ");
        denominador = lee.nextInt();
```

```
            /* En el bloque try se intenta dividir e imprimir el resultado.
            * En caso de que el denominador sea 0, en el bloque catch se indica
            * qué hacer. En este caso se imprime un mensaje propio acompañado
            * del mensaje obtenido por el método getMessage( ).
            */
            try{
                  resultado = numerador / denominador;
                  System.out.println("\nResultado:  " + resultado);
            } catch (Exception e){
                  System.out.println("\nNo se pudo dividir. Causa: " + e.getMessage( ));
                  // System.out.println("\nNo se pudo dividir. Causa: " + e.toString( ));
                  // e.printStackTrace( );
            }
      }
}
```

Un ejemplo de ejecución es:

```
Ingrese numerador: 25
Ingrese denominador: 0

No se pudo dividir. Causa: / by zero
```

Si cambiamos la instrucción del bloque catch por:

```
System.out.println("\nNo se pudo dividir. Causa: " + e.toString( ));
```

el resultado sería:

```
Ingrese numerador: 25
Ingrese denominador: 0

No se pudo dividir. Causa: Java.lang.ArithmeticException: / by zero
```

Por último, si cambiamos la instrucción del bloque catch por e.printStackTrace(); el resultado sería:

```
Ingrese numerador: 25
Ingrese denominador: 0
Java.lang.ArithmeticException: / by zero
at libropoo.ManejoExcepciones.main(ManejoExcepciones.Java:27)
```

En el programa 1.9 se presenta otro ejemplo, ahora tratando las excepciones en el caso de la lectura de un archivo.

Programa 1.9 | **ManejoExcepcionesArchivo.java**

```
package cap1;
import java.io.File;
import java.util.*;

/**
 * Ejemplo de manejo de excepciones en la lectura de archivo. Evita que si falla
 * la apertura del archivo se produzca un error durante la ejecución.
 * @author Silvia Guardati
 * Programa 1.9
 */
public class ManejoExcepcionesArchivo {

        public static void main(String[] args) {

                /* En el bloque try se intenta abrir un archivo de nombre "Datos". Si es posible,
                * mientras haya valores que leer, estos se leen y se suman. Si la apertura
                * del archivo falla, pasa al bloque catch donde se toma la excepción y,
                * en este caso, se imprime el mensaje que arroja el método getMessage().
                * Por último, en el bloque finally se despliega un mensaje se haya o no
                * leído del archivo.
                * */
```

```
        try {
            File archivo = new File("Datos.txt");
            Scanner lee = new Scanner(archivo);
            double suma = 0;
            while (lee.hasNextDouble( ))
                suma = suma + lee.nextDouble( );
            System.out.println("Suma de los valores leídos: " + suma);
            lee.close( );
        } catch(Exception e){
            System.out.println("Archivo: " + e.getMessage( ));
        } finally {
            System.out.println("Lectura de un archivo con manejo de excepciones ");
        }
    }
}
```

Se ejecuta el programa considerando que el archivo "Datos" tiene los siguientes valores:

2.5
3.4
4.2
5.6
6.8

El resultado obtenido es:

```
Suma de los valores leídos: 22.5

Lectura de un archivo con manejo de excepciones
```

Si cambiamos el nombre del archivo por uno que no existe, el resultado obtenido es:

```
Archivo: Datos.txt (El sistema no puede encontrar el archivo especificado)
Lectura de un archivo con manejo de excepciones
```

Observe que en ambos casos se imprime la línea indicada en el bloque finally.

En el programa 1.10 se presenta otro ejemplo del manejo de excepciones. En este caso, para validar que una entrada sea un número entero.

Programa 1.10 **ManejoExcepcionesEntradaDatos.java**

```
package cap1;
import java.util.Scanner;

    /**
     * Ejemplo de manejo de excepciones en la entrada de datos. Evita que el ingreso
     * de un valor que no sea un número entero produzca un error durante la ejecución.
     * @author Silvia Guardati
     * Programa 1.10
     */
    public class ManejoExcepcionesEntradaDatos {

        public static void main(String[] args) {
            Scanner lee = new Scanner(System.in);
            int número = 0;
            boolean bandera = false;

            /* En el bloque try se intenta leer un número entero. Si esto sucede se
            * altera la bandera para salir del ciclo e interrumpir la entrada de datos.
            * En el bloque catch se imprime un mensaje indicando el tipo de dato esperado
            * y se hace una lectura para prepararse para el siguiente intento.
            */
            while (!bandera)
                try {
                    System.out.print("Ingresa un número entero: ");
                    número = lee.nextInt();
                    bandera = true;
                } catch (Exception e){
                    System.out.println("\n¡Debes ingresar un número entero!");
                    lee.next();
                }
            System.out.println("El valor leído es: " + número);
        }
}
```

Un ejemplo de ejecución es:

```
Ingresa un número entero: 10
El valor leído es: 10
```

En este caso se ejecuta el bloque try con éxito.

Otra posible ejecución es:

```
Ingresa un número entero: 10.0
¡Debes ingresar un número entero!
Ingresa un número entero: !0
¡Debes ingresar un número entero!
Ingresa un número entero: 10
El valor leído es: 10
```

En el segundo intento se da un número con parte decimal, por lo tanto se ejecuta el bloque *catch* para tomar la excepción. En el siguiente se da un carácter, por lo que vuelve a ejecutarse el bloque *catch*. Finalmente, se da un número entero, por lo que se ejecuta el *try* alterándose la bandera y saliendo del ciclo.

El manejo de excepciones se complementa con el lanzamiento de excepciones y con la creación de excepciones propias. Ambos temas se verán en otros capítulos.

1.4 PROGRAMACIÓN MODULAR

La programación modular es una de las buenas prácticas de la ingeniería de software,[5] y está basada en el principio de *divide y vencerás*. Este principio se usa mucho en la solución de problemas complejos y/o de gran tamaño, sin importar la naturaleza de dichos problemas. Por ejemplo, si se quiere construir un edificio es más fácil pensarlo como varios proyectos que, integrados, nos dan todo el edificio. Así, estará el equipo que se encargará de la construcción de paredes y techos, otro que se concentrará en la instalación eléctrica, otro en la plomería y así, diferentes grupos, hasta lograr terminar la obra. En este ejemplo, al dividir el todo en partes hace posible que algunos de los grupos puedan trabajar simultáneamente y, además, permite asignar ciertas tareas a grupos especializados, lo cual ayuda a garantizar la calidad del producto final.

Lo mismo sucede en la industria del software. Si la generación de un producto se divide en componentes, entonces cada componente estará a cargo de un equipo de desarrollo y se podrá tener varios equipos trabajando en paralelo. Además, se pueden asignar determinados componentes a aquellos equipos que cuenten con los especialistas necesarios.

[5] Disciplina encargada del estudio y aplicación de metodologías y procesos para el diseño e implementación de productos de software.

Existen diferentes maneras de plantear la división de la solución de un problema en partes. En esta sección presentaremos el uso de subprogramas o módulos. Los subprogramas tradicionalmente reciben el nombre de funciones si regresan un resultado, y si no lo hacen se les denomina procedimientos. Con el enfoque de la programación orientada a objetos, se llama métodos, de manera indistinta, a todos los subprogramas.

La creación de subprogramas ofrece varias ventajas:

- Promueve la abstracción de la solución, permitiendo una solución más comprensible.
- Facilita esconder código irrelevante para la lógica general de la solución.
- Permite que los módulos se puedan asignar a equipos distintos y, por lo tanto, se pueda trabajar en paralelo.
- Permite reusar código. Un subprograma ya probado que resuelve una tarea específica podrá volver a emplearse en la solución de otro problema, evitando así duplicar código.
- Facilita la prueba del código. Es mucho más fácil revisar y, si corresponde, corregir varios subprogramas (pequeños y simples) en lugar de revisar y corregir un solo programa grande y complejo.

Cohesión: **este concepto hace referencia a que un subprograma debe hacer una sola tarea. Es decir, que todo el código que se escriba en un subprograma debe resolver un único problema. Por ejemplo, un subprograma no sería cohesivo si dadas las calificaciones de un grupo de alumnos calculara el promedio y la calificación más alta. En este caso, lo mejor sería tener un subprograma que calcule el promedio y otro que encuentre la calificación más alta.**

En el lenguaje Java los subprogramas se escriben por medio de la siguiente sintaxis:

```
public static tipoResultado nombreSubprog([tipo nomParam1, ..., tipo nomParam2]){
        // Cuerpo del subprograma
}
```

donde:

- public: indica que el subprograma o método es público. Se podrá usar desde cualquier clase (programa) del proyecto o desde otros proyectos si se importa la clase previamente. Variantes de este modificador se estudiarán en el capítulo 2.
- static: indica que el subprograma (o método) no se asocia a un objeto. Este concepto adquiere sentido en el siguiente capítulo cuando se estudien las clases y los objetos.
- tipoResultado: es el tipo de resultado que arroja el subprograma, pudiendo ser cualquiera de los tipos válidos en Java.
- nombreSubprograma: representa el nombre del subprograma. El nombre debe estar relacionado con lo que hace el método. Se sugiere que empiece con un verbo y, si se requiere mayor información, se continúe con un sustantivo. Se escribe aplicando la notación "camello".
- parámetros: los [] indican que son opcionales. Es decir, hay subprogramas que pueden no tener parámetros. Si los hubiera se especifica de qué tipo de dato son y su nombre, todos separados por coma.

- Cuerpo del subprograma: es el conjunto de instrucciones necesarias para alcanzar el resultado por el cual se escribe el subprograma. Se pueden usar cualquiera de las instrucciones válidas en Java.

La primera línea (desde la palabra public hasta el cierre de paréntesis) se conoce como el *encabezado* o la *firma del método*. Es importante conocer la firma del método porque es por medio de ella que los programas/subprogramas se comunican.

Cuando se pasa una variable —que almacene cualquiera de los tipos de datos estudiados hasta el momento— como parámetro, si se modifica dentro del subprograma, al regresar el control al programa (o subprograma) que invocó dicha variable no conserva los cambios. En este caso, se dice que los parámetros *pasan por valor*, es decir, no se proporciona su referencia o dirección, sino sólo el contenido o valor. Como se verá en otros capítulos, cierto tipo de datos sí puede modificarse en los subprogramas. Cuando esto sucede, se dice que *pasan por referencia*, ya que lo que se proporciona es la referencia o dirección de una variable.

En el programa 1.11 se presenta un ejemplo de uso de subprogramas. Considere que se quiere conocer el promedio de las edades de un grupo de alumnos. Para ello se deben leer las edades y calcular el promedio. Se utilizaron dos subprogramas. El primero de ellos lee y valida el total de edades, asegurándose que sea un número entero mayor que 0. El segundo se encarga de leer cada una de las edades y calcula el promedio, regresándolo como resultado.

Buenas prácticas

- ☑ Un subprograma debe ser cohesivo.
- ☑ Un subprograma debe tener un tamaño no mayor al de una o dos pantallas, una o dos páginas de código.
- ☑ El nombre del subprograma debe empezar con un verbo en minúscula. Si sigue alguna otra palabra, ésta debería empezar con mayúscula.
- ☑ El nombre del subprograma debe indicar qué hace el subprograma.
- ☑ El número de parámetros no debe ser mayor a siete.
- ☑ El nombre de los parámetros debe ser claro.

Programa 1.11 | **EjemploSubprograma1.java**

```
package cap1;
import java.util.Scanner;

/**
 * Ejemplo del uso de subprogramas/métodos. La solución del problema está formada
 * por dos subprogramas y el método principal que es el que tiene el control.
 * @author Silvia Guardati
* Programa 1.11
*/
public class EjemploSubprograma1 {
```

```
/* Subprograma que lee un número entero que representa el total de
* edades que se desea promediar. Por medio del try catch
* se valida la entrada, evitando un error en caso de que se introduzca
* un dato que no sea un número entero. Se retoma el código del programa 1.10
*/
public static int leeTotalDatos(){
        Scanner lee = new Scanner(System.in);
        int número = 0;
        boolean bandera = false;

        while (!bandera || número <= 0)
                try {
                        System.out.print("Ingresa el total de edades (>0): ");
                        número = lee.nextInt();
                        bandera = true;
                } catch (Exception e){
                        System.out.println("\n¡Debes ingresar un número entero!");
                        lee.next();
                }
        return número;
}

/* Subprograma que lee n edades y calcula y regresa el promedio de las
* mismas. Recibe como parámetro el total de edades, regresa un número
* tipo double que es el promedio de las edades.
*/
public static double calculaPromedio(int n){
        Scanner lee = new Scanner(System.in);
        double edad, promedio;
        int i;
```

```
            promedio = 0;
            for (i= 1; i <= n; i++){
                System.out.println("Ingresa la edad " + i + ": ");
                edad = lee.nextDouble( );
                promedio = promedio + edad;
            }

            promedio = promedio/n;
            return promedio;
        }

        // Método principal
        public static void main(String[ ] args) {
        int n;
        double promedio;

        n = leeTotalDatos( );
        promedio = calculaPromedio(n);
        System.out.println("\nEl promedio de las edades es: " + promedio);
        }
}
```

La firma del primer subprograma es public static int leeTotalDatos(), la cual indica que el método se identifica con el nombre leeTotalDatos, que regresa un resultado entero y que no recibe parámetros.

La firma del segundo subprograma es public static double calculaPromedio(int n), la cual indica que el método se identifica con el nombre calculaPromedio, que regresa un número tipo double y que recibe como parámetro un número entero.

El programa 1.12 es una generalización del anterior. Se usa una variable tipo cadena de caracteres (String) como parámetro para generar una salida con información para el usuario. Esa información se controla desde el main, generalizando así el uso de los subprogramas.

Programa 1.12 | **EjemploSubprograma2.java**

```
package cap1;
import java.util.Scanner;

/**
* Ejemplo del uso de subprogramas/métodos. La solución del problema está formada
* por dos subprogramas y el método principal que es el que tiene el control.
* @author Silvia Guardati
* Programa 1.12
*/
public class EjemploSubprograma2 {

    /* Subprograma que lee y regresa un número entero, mayor que cero.
    * Recibe un parámetro para indicar los datos que se quieren leer.
    * Por medio del try catch se valida la entrada, evitando un error en caso
    * de que se introduzca un dato que no sea un número entero.
    * Se retoma el código del programa 1.11
    */
    public static int leeTotalDatos(String mensaje){
        Scanner lee = new Scanner(System.in);
        int número = 0;
        boolean bandera = false;

        while (!bandera || número <= 0)
            try {
                System.out.print("\nIngresa el total de " + mensaje + " ( > 0): ");
                número = lee.nextInt( );
                bandera = true;
            } catch (Exception e){
                System.out.println("\n¡Debes ingresar un número entero!");
```

```
                    lee.next();
                }
        return número;
}

/* Subprograma que lee n datos y calcula y regresa el promedio de los
 * mismos. Recibe como parámetro el total de datos y una cadena que indica
 * qué representan los datos (edades, precios, calificaciones, etc.).
 * Regresa un número tipo double que es el promedio de los datos.
 */
public static double calculaPromedio(int n, String mensaje){
        Scanner lee = new Scanner(System.in);
        double dato, promedio;
        int i;

        promedio = 0;
        for (i= 1; i <= n; i++){
                System.out.println("Ingresa " + mensaje + " " + i +": ");
                dato = lee.nextDouble();
                promedio = promedio + dato;
        }

        promedio = promedio/n;
        return promedio;
}

// Método principal
public static void main(String[] args) {
        int n;
        double promedio;
        /* Se utilizan los subprogramas para leer el total de edades, cada una
         * de ellas y calcular el promedio.
```

```
        */
        n = leeTotalDatos("Edades");
        promedio = calculaPromedio(n, "edad");
        System.out.println("\nEl promedio de las edades es: " + promedio);

        /* Se utilizan los subprogramas para leer el total de calificaciones,
         * cada una de ellas y calcular el promedio.
         */
        n = leeTotalDatos("Calificaciones");
        promedio = calculaPromedio(n, "calificación");
        System.out.println("\nEl promedio de las calificaciones es: " + promedio);
    }
}
```

1.5 PRUEBAS DE SOFTWARE

Desde hace varios años el tema de la calidad ocupa un lugar importante en la industria del software. Por calidad se entiende entregar los productos en el tiempo estimado, usando los recursos estimados y libres de errores. A pesar de que parece obvio, no resulta tan fácil de cumplir. La mayoría de las empresas dedicadas al desarrollo de software no satisfacen uno o más de estos puntos.

Probar primero la solución diseñada y luego la implementación de la misma es clave para garantizar la calidad del producto generado. La meta de las pruebas es que los productos: 1) cumplan el objetivo por el cual se solicitaron y 2) que lo hagan bien. Por ejemplo, si el programa debe calcular e imprimir el total a pagar por la compra de n productos, se deberá probar que: 1) calcula el total, considerando el IVA y los descuentos —si corresponden— y lo imprime; y 2) realiza las operaciones involucradas (suma de precios, cálculo del IVA, etc.) de manera correcta y con la precisión esperada. Si bien las pruebas no garantizan terminar el producto a tiempo y con los recursos planeados, sí contribuyen a entregar un producto que cumpla con todos los requisitos señalados y que los mismos estén libres de errores. Además, las pruebas bien planeadas y aplicadas en todas las fases del desarrollo de software ayudan a encontrar y, en consecuencia, a corregir defectos desde etapas tempranas, lo cual resulta menos costoso, en tiempo y dinero, que hacerlo una vez terminado el producto. Por lo tanto, también contribuyen a cumplir con el plazo y el presupuesto.

Existen diversas técnicas de pruebas dependiendo del objetivo que persigan, así como de la etapa en la cual se aplican. En este capítulo presentaremos dos de ellas, las cuales sirven para introducir el tema y se pretende también concientizar desde la formación de los futuros ingenieros de software acerca de la importancia de cuidar la calidad desde el inicio de cualquier desarrollo.

1.5.1 Mapa de memoria o prueba de escritorio

Este tipo de pruebas pertenece a la categoría de pruebas estáticas, ya que su aplicación no requiere ejecutar el programa. La prueba se realiza "en papel", revisando el diagrama de flujo o el código, siguiendo la lógica de la solución para distintas posibles entradas, llamadas casos de prueba. Como en toda prueba, se intenta encontrar errores, no demostrar que la solución es correcta. Por lo tanto, es necesario que los casos de prueba contemplen la mayor cantidad posible de escenarios que puedan presentarse.

A los mapas de memoria o prueba de escritorio también se les conoce como pruebas de caja blanca, porque implican conocer la lógica de la solución (el diagrama de flujo o el código). En el caso de las pruebas de caja negra, simplemente se compara el resultado esperado con el obtenido, sin analizar la lógica interna.

Un mapa de memoria normalmente es una tabla en la cual las columnas representan las variables usadas en la solución y los renglones los valores que van tomando durante la simulación de la ejecución. Retomando el problema cuya solución fue representada por medio del diagrama de flujo de la figura 1.14, a continuación se muestra el mapa de memoria correspondiente. Observe que es un mapa formado por una única variable, n, y una columna dedicada a las impresiones. Para este problema se eligieron 4 casos de prueba: n = 7, n = 6, n = 1 y n = -3.

Tabla 1.2 Mapa de memoria: Serie de Ullman–caso de prueba 1 (n = 7)

n	Imprime
7	7
22	22
11	11
34	34
17	17
52	52
26	26
13	13
40	40
20	20
10	10
5	5
16	16
8	8
4	4
2	2
1	1

Tabla 1.3 Mapa de memoria: Serie de Ullman–caso de prueba 2 (n = 6)

n	Imprime
6	6
3	3
10	10
5	5
16	16
8	8
4	4
2	2
1	1

Tabla 1.4 Mapa de memoria: Serie de Ullman–caso de prueba 3 (n = 1)

n	Imprime
1	1

Tabla 1.5 Mapa de memoria: Serie de Ullman–caso de prueba 4 (n = -3)

n	Imprime
-3	-3

En el último caso, como no se valida el dato de entrada, se genera un resultado inconsistente, ya que se imprime que la serie está formada por el -3. Este caso de prueba demuestra que la solución no es completa, ya que para valores negativos no debería generar la serie de Ullman.

A continuación se presenta el mapa de memoria correspondiente al programa 1.7. Se realizaron tres pruebas, la primera de ellas con un total de calificaciones igual a 0, lo que provoca otra lectura. En el segundo caso se ingresa un valor negativo, lo que también resulta en otra lectura. Por último, se ingresa el 4 que indica el total de calificaciones, lo cual permite avanzar en la solución.

Tabla 1.6 Mapa de memoria: Cálculo del promedio–caso de prueba 1

Total cal	i	cal	suma	prom	Imprime	Comentario
0						Al ingresar 0, se vuelve a pedir.
-3						Al ingresar un número < 0, se vuelve a pedir.
4			0			
	1	8	8			
	2	7	15			
	3	9	24			
	4	8	32			
	5			8	8	

Como se puede observar, en esta solución no se aplica ninguna validación sobre los datos ingresados como calificaciones. Por lo tanto, se permite ingresar valores negativos o fuera de rangos (ver tabla 1.7). El caso de prueba 2 pone en evidencia que la solución no contempla qué hacer en caso de datos inválidos, lo que genera resultados inconsistentes. Una solución robusta exigiría conocer el rango de calificaciones aceptables (de 0 a 10 o de 0 a 100, sólo enteros, etc.) y, posteriormente, validar cada entrada antes de procesarse.

Tabla 1.7 Mapa de memoria: Cálculo del promedio–caso de prueba 2

Total cal	i	cal	suma	prom	Imprime	Comentario
3			0			
	1	-3	-3			No detecta calificación negativa
	2	21	18			No detecta calificación fuera de rango.
	3	4.5	22.5			
	4			7.5	7.5	

1.5.2 Pruebas unitarias

Durante el desarrollo de software se aplican distintas pruebas según el grado de avance del producto. Las *pruebas unitarias* tienen por objetivo probar, de manera aislada, componentes desarrollados. Por lo tanto, se aplica este tipo de pruebas a subprogramas, a programas, a métodos de una clase o a clases. Una vez probados los componentes se pasa a la integración de los mismos, la cual consiste en lograr que trabajen juntos. Para probar que trabajan juntos correctamente, se aplican las *pruebas de integración*. Posteriormente, luego de integrar todos los componentes y hechas las pruebas de integración en cada caso, se realiza la *prueba de sistema*. Es decir, se prueba al sistema funcionando como un único producto de software. Pasada esta prueba, se continúa con las *pruebas de aceptación*. Éstas normalmente se realizan en el ambiente de operación (o en uno similar) y junto con el usuario/cliente, ya que se trata de probar el sistema desde la perspectiva del negocio.

Las *pruebas unitarias*, como ya se mencionó, consisten en determinar que un componente cumple correctamente con el requisito planteado. NetBeans ofrece una herramienta (de Java) llamada JUnit que permite diseñar pruebas unitarias de manera amigable y fácil. Es importante mencionar que la herramienta sólo facilita la aplicación de las pruebas, pero el éxito de las mismas siempre dependerá de que se elija convenientemente qué probar y los datos de prueba adecuados. En el capítulo 2 se regresa a este tema.

1.6 RESUMEN

En este capítulo se presentaron las bases necesarias para empezar a diseñar e implementar soluciones algorítmicas a problemas. Se explicaron las estructuras algorítmicas selectivas y repetitivas, y la programación modular. Haciendo uso de estos elementos el lector puede resolver problemas para los cuales existe una solución algorítmica.

Todos los temas estudiados a lo largo del capítulo se complementaron con ejemplos que ayudan al estudiante a comprenderlos y a aplicarlos en la solución de problemas.

1.7 EJERCICIOS

1.1 Analice y resuelva las siguientes expresiones:

a. 3 + 8 * 5 - 6 / 3

b. 2.5 * 2 * 3 - 4 / 2 + 8

c. 2 * (6 – 2.5) / 3

d. Math.pow(3.5, 2) + 10 * (12 – 8)

e. 23 > 18

f. 16 – 12 * 2 <= Math.sqrt(27.5)

g. 3 + 2.5 * 4 / 3 != 4 * 5.2 – 3.8

h. (25 / 2 + 3.4 == 18 /3 + 4 * 2) || (23 < 15 * 2 * Math.abs(-2))

i. (8.5 / 3.2 + 6.5 – 2.3 * 5 / 6 >= 15.8) && (21.4 < 81 /4 + 2)

1.2 Dados los lados de un rectángulo, calcular e imprimir el área y el perímetro.

Datos: base y altura

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.3 Dados los lados de un triángulo, calcular e imprimir el área del mismo usando la fórmula:

área = $\sqrt{(\text{per} * (\text{per} - \text{lado1}) * (\text{per} - \text{lado2}) * (\text{per} - \text{lado3}))}$

donde:

- per = (lado1 + lado2 + lado3) / 2
- lado1, lado2 y lado3 son los tres lados del triángulo (los tres deben ser positivos)

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.4 Dados tres números (n1, n2 y n3), identificar e imprimir el menor de ellos.

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.5 Considere que se dispone de la producción de carbón (en toneladas) de los últimos 12 meses. Con estos datos se quiere calcular e imprimir el promedio anual de toneladas.

Datos: $pCar_1$, $pCar_2$, ..., $pCar_{12}$

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.6 Dados los precios de ventas de n productos, encontrar e imprimir el de mayor precio.

Datos: n, p_1, p_2, ..., p_n

donde:

- n es el total de productos, n> 0.
- pi, es el precio del producto i ($1 \leq i \leq n$).

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado

1.7 Retome el ejercicio anterior. Considere ahora que los datos no son precios de productos (siendo todos los precios números positivos) sino números en general. Para encontrar el valor más grande, ¿puede usar la solución encontrada en el ejercicio anterior? ¿Por qué sí/no? ¿Existen otras alternativas para resolver este tipo de problemas?

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.8 Retome el ejercicio 6. Se tienen los mismos datos, pero ahora se quiere obtener la siguiente información:

- **a.** El mayor precio.
- **b.** El menor precio.
- **c.** El promedio de los precios.
- **d.** Total de precios mayores que $200.

¿Puede usar la solución encontrada en los ejercicios anteriores? ¿Por qué sí/no? ¿Existen otras alternativas para resolver este tipo de problemas?

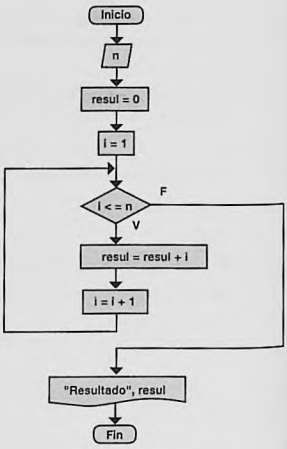
Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.9 Se deben encontrar las raíces reales de una expresión cuadrática, teniendo como datos los valores de los coeficientes (a, b, c).

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.10 Analice el diagrama de flujo que aparece a la derecha.

- **a.** Realice el mapa de memoria para n = -3 y para n = 5.
- **b.** Identifique las estructuras de control usadas.
- **c.** Prográmelo utilizando Java.



1.11 En una concesionaria automotriz se tiene información sobre las ventas realizadas a lo largo del último mes. Se conoce el total de ventas y el precio de venta de cada unidad. También se sabe que la concesionaria paga una comisión a sus vendedores de acuerdo con el precio de la unidad.

Si precio ≤ 100,000, la comisión es del 1.5%

Si precio > 100,000 y precio ≤ 250,000, la comisión es del 1.8%

Si precio > 250,000, la comisión es del 2.3%

Datos: n, $precio_1$, $precio_2$, ..., $precio_n$

Donde:

- n es el total de automóviles vendidos (n > 0)

- $precio_i$ es el precio de la venta i ($1 \leq i \leq n$) y $precio_i > 0$

Estos datos deben ser procesados de tal manera que se pueda generar e imprimir la siguiente información:

a. Comisión pagada por cada venta.
b. Total pagado por comisiones.
c. Total de autos vendidos con un precio menor o igual a $100,000.
d. Determinar si hubo alguna categoría de autos (por precio) que no se haya vendido en el último mes.

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.12 Durante el mes de diciembre, diariamente, se aplicó una encuesta a transeúntes en general preguntándoles cuántas latas de refrescos consumen al día. Como resultado se tiene la respuesta de los encuestados en cada uno de los 31 días del mes. El número de encuestados por día es variable.

Datos:

$lt_{1,1}$, $lt_{1,2}$, $lt_{1,3}$, ..., -1

$lt_{2,1}$, $lt_{2,2}$, $lt_{2,3}$, ..., -1

...

$lt_{31,1}$, $lt_{31,2}$, $lt_{31,3}$, ..., -1

Donde:

- $lt_{i,j}$ es el total de latas consumidas en el día i por el encuestado j.
- En cada uno de los días, cuando ya no se tengan datos, se dará un -1 como bandera de fin de datos.
- $lt_{i,j}$ igual a 0 indica que ese encuestado no consumió latas de refresco ese día.

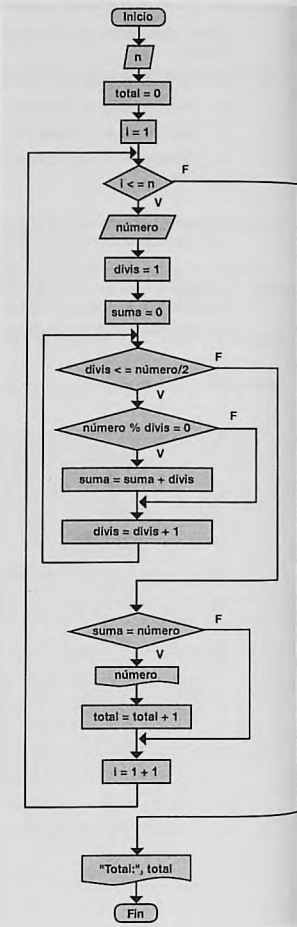
Estos datos deben ser procesados de tal manera que se genere e imprima la siguiente información:

a. Total de latas de refresco consumidas por día.
b. Promedio diario de latas consumidas.
c. ¿Cuántas personas encuestadas el día 6 no consumieron refrescos?
d. Total de encuestados que contestaron no haber consumido refresco.
e. ¿Cuántos días tuvieron al menos un encuestado con un consumo > 5 latas?

Realice el diagrama de flujo y el programa correspondiente a la solución de este problema. Pruebe su solución con un conjunto de datos adecuado.

1.13 Analice el diagrama de flujo que aparece a la derecha.

- **a.** Realice el mapa de memoria para n = 3 y para número = 24, 6, 58.
- **b.** ¿Qué obtiene? ¿Cuándo imprime a "número"?
- **c.** Identifique las estructuras de control usadas.
- **d.** Prográmelo utilizando Java.



1.14 Analice el diagrama de flujo anterior. Luego:

- **a.** Realice el mapa de memoria para n = –2 y para número = 4, 6.
- **b.** ¿El resultado sigue siendo correcto?
- **c.** Si su respuesta es no, ¿qué le modificaría?
- **d.** Prográmelo utilizando Java.

1.15 Sobre el diagrama de flujo del problema 13.

- **a.** Realice el mapa de memoria para n = 3 y para número = 4, -8, 6.
- **b.** ¿El resultado sigue siendo correcto y completo?
- **c.** Si su respuesta es no, ¿qué le modificaría?
- **d.** Prográmelo utilizando Java.

1.16 Escriba un subprograma que reciba como parámetro un número entero positivo (n > 0) y regrese como resultado el factorial de dicho número. Recuerde que el factorial de un número se calcula como:

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

1.17 Escriba un subprograma que reciba como parámetro un número entero positivo (n > 0) y regrese una cadena formada por la serie de Fibonacci, desde 1 hasta el número dado. Recuerde que los números de Fibonacci se calculan de la siguiente manera:

Fibonacci(0)= 0

Fibonacci(1)= 1

Fibonacci(2)= Fibonacci(1) + Fibonacci(0)

Fibonacci(3)= Fibonacci(2) + Fibonacci(1)

...

Fibonacci(n)= Fibonacci(n-1) + Fibonacci(n-2)

1.18 Escriba un subprograma que reciba un número entero positivo (n > 0) y regrese true si dicho número es primo. En caso contrario, deberá regresar false. Recuerde que un número primo es aquel que es divisible sólo entre la unidad y entre sí mismo.

1.19 Escriba un subprograma que reciba un número entero positivo (n > 0) y regrese true si dicho número es perfecto. En caso contrario, deberá regresar false. Un número es perfecto si es igual a la suma de sus divisores (excluyendo al mismo número e incluyendo la unidad). El 6 es un ejemplo de número perfecto, ya que 6 = 1 + 2 + 3.

1.20 Escriba un programa para probar los subprogramas anteriores.

1.21 Retome el programa 1.6 y modifíquelo de tal manera que si el número leído es menor que 1:

a. Imprima un mensaje adecuado e interrumpa la ejecución.

b. Fuerce al usuario a ingresar un número entero mayor que 1.

1.22 Retome el programa 1.7 y modifíquelo de tal manera que sólo acepte calificaciones comprendidas en el rango de 0 a 10.

1.23 Retome el programa 1.12, y modifique el método calculaPromedio() de tal manera que entre sus parámetros reciba los límites inferior y superior del intervalo de valores, en el cual deben estar los datos a leer. Por ejemplo, para leer las calificaciones podría recibir 0 y 10 o 0 y 100. Posteriormente, modifique el método main para que se lean esos valores y se proporcionen al método que obtiene el promedio. La firma del método debería verse como se muestra a continuación:

```
public static double calculaPromedio(int n, String mensaje, double limInf, double limSup)
```

PRINCIPIOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

2

Contenido

Competencias

- Explicar las características y los conceptos básicos de la Programación Orientada a Objetos.
- Explicar los conceptos de sobrescritura y sobrecarga.
- Presentar las interfaces como alternativa para establecer el comportamiento esperado de una clase.
- Explicar el concepto de herencia y su uso en la solución de problemas.

2.1 INTRODUCCIÓN

Para diseñar una solución y posteriormente escribir un programa se puede seguir diversos paradigmas. Uno de los más usados actualmente es el que se denomina *orientado a objetos*, el cual sirve de base tanto para el análisis y diseño de programas como para la implementación de los mismos. En este libro se emplea este enfoque, en particular el denominado Programación orientada a objetos (POO).

Cuando se analiza un problema, generalmente se identifican los datos involucrados y las operaciones que debe aplicarse a estos datos para resolverlo. Con el enfoque orientado a objetos, los datos y las operaciones se agrupan de tal manera que forman un todo que se denomina **clase**. Es decir, una clase incluye los datos que describen o definen un concepto, así como todas las operaciones asociadas a dichos datos.

En la POO se distinguen **clases** y **objetos**. Las primeras se utilizan para representar conceptos (concretos o abstractos), mientras que los objetos representan instancias de aquéllas. A la vez, una clase está formada por **atributos** y **métodos**. Los atributos describen el concepto representado, es decir, son cada una de las características del mismo, mientras que los métodos son las operaciones permitidas por la clase para realizarse sobre los atributos.

La figura 2.1[1] muestra un ejemplo de una clase que representa, de manera simplificada, el concepto "Perro". En este caso, un perro se define por medio de tres atributos: nombre, raza y edad. Es decir, todo perro quedará definido por estos elementos. También se incluyó un método que permite actualizar la edad del perro. Los símbolos que aparecen en el diagrama se explicarán con detalle más adelante en este mismo capítulo.

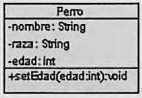


Figura 2.1 Clase *Perro*

La figura 2.2 muestra un ejemplo de una instancia de la clase *Perro*. Se tiene un objeto llamado *miPerro*, el cual representa a un perro en particular, pero no a todos los perros, como en el caso de la clase. Además, este perro tiene valores concretos para cada uno de los atributos: su nombre es *Charrúa*, su raza es *labrador* y tiene tres años de edad.

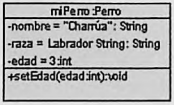


Figura 2.2 Objeto de la clase *Perro*

La POO se caracteriza porque permite la **abstracción** y el **encapsulamiento** de la información, así como establecer relaciones de **herencia** entre distintos elementos. También tiene un recurso llamado **polimorfismo**, que da cierta flexibilidad con la cual no se contaba en el paradigma estructurado. A continuación se explican estos cuatro aspectos de la POO.

[1] Las figuras se elaboraron con la herramienta TopCoder UML <http://community.topcoder.com/tc?module=Static&d1=dev&d2=umltool&d3=description)

1. **Abstracción.** Es la propiedad que determina que un objeto o concepto se pueda representar, considerando sus elementos más importantes e ignorando aquellos que resultan irrelevantes. De esta manera el diseñador de la solución se concentra en los aspectos del concepto que puedan impactar en la solución.
2. **Encapsulamiento.** Esta característica de la POO hace referencia a que en una clase se incluyen todos los elementos que la componen, de tal manera que forman una especie de "caja negra" que contiene todo lo que se necesita y que no requiere que los usuarios de la misma conozcan su estructura. Es decir, oculta los detalles de implementación de la clase. Además, obliga al usuario a utilizar los métodos provistos para el acceso a los datos, ayudando así a mantener la seguridad de los mismos.
3. **Herencia.** Esta propiedad es muy útil para compartir elementos entre clases y para representar jerarquías de conceptos. Es decir, dada una clase que representa un concepto se puede definir una o más clases como clases derivadas de la primera y que representan conceptos más específicos. Considere el ejemplo de la figura 2.3, en el cual se observa una clase llamada *Mamífero*, que representa a cualquier animal mamífero, y una clase más específica, llamada *Perro*, que representa sólo a los mamíferos que sean perros. En este caso, un perro hereda todos los atributos de la clase general, y además tiene sus propios atributos.

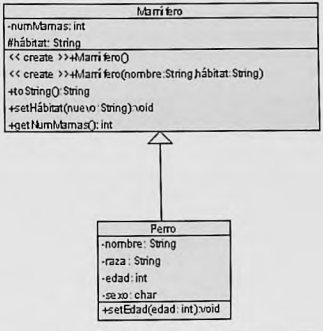


Figura 2.3 Objeto de la clase *Perro*

4. **Polimorfismo.** Esta característica permite que variables que almacenan la referencia de un objeto puedan tomar distintas formas. Es decir, un objeto no puede cambiar de forma, pero una variable polimórfica puede hacer referencia a distintos tipos de objetos.

2.2 CLASES

Una *clase* describe un conjunto de elementos que se "parecen" por su estructura y por su comportamiento. La *estructura* hace referencia a aquellos datos (características) comunes al concepto que se está describiendo; el *comportamiento*, a las actividades que realizan los individuos que la forman y a las operaciones que involucran a los datos de los mismos. Por ejemplo, con la clase *Alumno* se representa el conjunto de individuos que asisten formalmente a alguna escuela y que están inscritos en algún programa académico. Todos los alumnos tienen un nombre, una dirección, una clave que los identifica y un promedio, entre otras características, todo lo cual define la estructura de la clase. Además, los alumnos pueden realizar actividades como inscribirse o darse de baja de una materia y se les puede cambiar la dirección y/o el promedio; todas estas tareas definen el comportamiento de la clase.

Los datos que forman la estructura reciben el nombre de **atributos**, y el comportamiento se define por medio de **métodos**. Retomando el ejemplo de la clase *Alumno*, los atributos son el nombre, la dirección, la clave y el promedio; los métodos son los que permiten actualizar el domicilio y el promedio, y los que permiten inscribir o dar de baja alguna materia.

Los componentes o **miembros** (atributos y métodos) de la clase van a estar acompañados de un **modificador** que determina el tipo de acceso que se puede tener sobre ellos. En *Java* se utilizan tres tipos diferentes de modificadores: **private**, **protected** y **public**.

Private. Es el modificador que da mayor protección a los miembros de una clase. Cuando acompaña a un atributo o método indica que se puede tener acceso a dicho componente sólo por medio de otros miembros de la misma clase. En la figura 2.3 el signo menos (–) junto al atributo *numMamas* lo marca como privado. Por tanto, este atributo podrá conocerse o modificarse a través de los métodos propios de la clase *Mamífero*. Este tipo de modificador es el que permite el encapsulamiento u ocultamiento de los miembros de una clase.

Protected. Este modificador se usa principalmente cuando se tiene la relación de herencia entre clases. Cuando acompaña a un atributo o método indica que se puede tener acceso a dicho componente sólo por medio de otros miembros de la misma clase y por miembros de clases derivadas. En el caso del lenguaje Java, el acceso también alcanza a otras clases del mismo paquete. En la figura 2.3, el signo cardinal *(#)* junto al atributo *hábitat* lo marca como protegido. Por tanto, este atributo podrá conocerse o modificarse por medio de los métodos de la clase *Mamífero* y de su derivada *Perro*.

Public. Este modificador es el que permite que un miembro de una clase pueda ser usado por otras clases. Cuando acompaña a un atributo o método, indica que se puede tener acceso a dicho componente desde cualquier otra clase. En la figura 2.3, el signo *más* (+) junto al método *setHábitat* lo marca como público. Por tanto, dicho método podrá invocarse desde la clase *Mamífero*, desde su derivada *Perro* o desde cualquier otra clase usuaria.

2.2.1 Representación de una clase en UML

El **lenguaje UML (Unified Modeling Language)** se utiliza para modelar, de manera estándar, diferentes elementos. Los diagramas de clases son una de las vistas que ofrece UML.

Con un *diagrama de clases* se representa una o varias clases y las relaciones entre ellas, si las hubiera. En la sección previa, la figura 2.1 representa la clase *Perro*, con todos sus miembros (atributos y métodos), mientras que la figura 2.3 representa las clases *Mamífero* y *Perro*, y una relación de herencia entre ellas.

El diagrama de una clase está formado por tres partes. En la primera de ellas se escribe el nombre de la clase, la primera letra mayúscula, centrado. Es una buena práctica elegir el nombre de la clase de tal manera que exprese claramente el concepto que se está representando. La segunda parte es para los atributos, los cuales se expresan indicando primero su modificador, luego el nombre seguido de dos puntos y, por último, el tipo del atributo. Para los modificadores se usan los signos –, #, + explicados previamente. Siguiendo con la clase *Perro*, la línea:

-nombre: String

indica que el atributo se llama *nombre*, es privado y es de tipo cadena de caracteres.

Los nombres de los atributos se escriben con minúsculas y deben expresar la característica correspondiente de la clase. La tercera parte del diagrama está destinada a los métodos. Cada método se indica con un modificador (siendo el *public* el más usado), seguido del nombre del método, sus parámetros –si los tuviera–, dos puntos y el tipo de resultado que obtiene. Es una buena práctica llamar a los métodos usando un verbo y alguna otra palabra que, juntos, expresen el objetivo del mismo. Siguiendo con la clase *Perro*, la línea:

+setEdad(edad:int): void

indica que el método *setEdad* es público, tiene un parámetro llamado *edad*, que es de tipo entero y, además, no regresa resultado. La palabra *void*, en Java, se usa para especificar que un método no regresa ningún tipo de resultado.

2.2.2 Definición de una clase en Java

La definición de una clase en Java empieza con las palabras reservadas public[2] class y el nombre dado a la clase. Luego se declara cada uno de los atributos y, por último, se define los métodos. A continuación se presenta la sintaxis usada para la definición de una clase.

```
public class NombreClase {
    modificador tipo atributo;
    modificador tipo atributo;
    ...
    modificador tipo método ([parámetros]) {
    ...
    }

    modificador tipo método ([parámetros]) {
    ...
    }
}
```

[2] Pueden usarse otros modificadores. Se volverá sobre este punto más adelante.

Donde *modificador* puede ser *private*, *protected* o *public*, con las implicaciones ya analizadas; *tipo* hace referencia al tipo del atributo o al tipo del resultado que obtiene el método. Además, los métodos pueden tener parámetros, en cuyo caso los mismos se enlistan separados por comas, indicando el tipo y el nombre del parámetro:

tipo nombreParam1, tipo nombreParam2

La primera línea de la declaración del método recibe el nombre de **firma del método**. Es muy importante este concepto porque es por medio de los métodos que los objetos se comunican y, por tanto, la firma establece cómo se llevará a cabo esa comunicación.

Buenas prácticas

- ☑ El nombre de las clases comienza con una letra mayúscula. Si estuviera formado por más de una palabra, la inicial de cada una de ellas se escribe con mayúscula.
- ☑ Generalmente, el nombre de la clase es un sustantivo singular.
- ☑ El nombre de la clase debe expresar claramente el concepto que se está representando.
- ☑ El nombre de los atributos se escribe con letras minúsculas.
- ☑ El nombre de los atributos debe expresar claramente la característica que está representando cada uno de ellos.
- ☑ El nombre de los métodos es un verbo seguido de alguna o varias palabras que den más información acerca de lo que hace el método. El verbo se escribe con letras minúsculas y la inicial de la siguiente palabra en mayúscula.
- ☑ Crear un archivo fuente por cada clase.
- ☑ El nombre del archivo es igual al de la clase que almacena.

A continuación se presenta un ejemplo en el cual se incluye el diagrama de clase UML de la clase *Alumno* (figura 2.4) y su correspondiente codificación en Java en el programa 2.1. El método *Alumno()*, llamado igual que la clase, es un tipo especial de método –denominado constructor– que por su importancia se tratará en la siguiente sección.

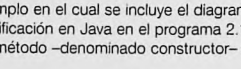


Figura 2.4 Clase *Alumno*

Programa 2.1 **Alumno.java**[3]

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.1
 * Se define, de manera muy simple, a un alumno por medio de tres atributos y
 * algunos métodos.
 */

public class Alumno{
   private String nombre;
   private String carrera;
   private double promedio;

   /**
   * @Constructor por omisión
   */
   public Alumno(){
   }

   /**
   * @Actualiza la carrera de un alumno
   * Recibe una cadena como parámetro. El parámetro tiene el mismo nombre que
   * el atributo, por lo tanto se usa la palabra this para indicar que al atributo
   * se le asigna el valor del parámetro.
   */
```

[3] En el método *setCarrera()* se usa la palabra reservada *this* para referenciar el atributo correspondiente al objeto que invoca al método. Es decir, se especifica que al atributo *carrera* del objeto que invoca al método se le debe asignar el valor del parámetro *homónimo*. Si el parámetro tuviera otro nombre, entonces el *this* puede omitirse. En el método *setPromedio()* se tiene el mismo caso.

```
    public void setCarrera(String carrera){
        this.carrera = carrera;
    }

    /**
    * @Actualiza el promedio de un alumno
    * Recibe un número como parámetro. El parámetro tiene el mismo nombre que
    * el atributo, por lo tanto se usa la palabra this para indicar que al atributo
    * se le asigna el valor del parámetro.
    */
    public void setPromedio(double promedio){
        this.promedio = promedio;
    }

    /**
    * @Regresa el promedio del alumno.
    */
    public double getPromedio(){
        return promedio;
    }
}
```

Una vez definida la clase, para trabajar con un objeto que sea una ocurrencia de la misma se debe: (1) declarar y, posterior o simultáneamente, (2) instanciar el objeto de la siguiente manera:

```
NombreClase nombreObjeto;  // (1) Declaración
nombreObjeto = new NombreClase ( ) ;  // (2) Instanciación
```

Si los pasos (1) y (2) se realizan simultáneamente, se escriben:

```
NombreClase nombreObjeto = new NombreClase ( ) ;  // Declaración e instanciación
```

Retomando el ejemplo del programa 2.1, se hace:

```
Alumno unAlumno;
unAlumno = new Alumno( );
```

o bien:

```
Alumno unAlumno = new Alumno( );
```

Para invocar un método asociado a un objeto se aplica la sintaxis:

```
nombreObjeto.nombreMétodo
```

Siguiendo con el ejemplo del programa 2.1, para cambiar el nombre de la carrera que cursa el alumno identificado por el objeto *unAlumno*, se hace:

```
unAlumno.setCarrera(nomCarrera); // Se le asigna el valor de la variable nomCarrera
```

o

```
unAlumno.setCarrera("Ingeniería en Computación"); // Se le asigna una constante
```

Las operaciones que involucra el método se aplican sobre el objeto que lo invoca.

2.2.3 Constructores

El constructor es un tipo especial de método que se invoca al instanciar un objeto, el cual crea y, a veces, inicializa al objeto creado. Los constructores se llaman igual que la clase, pueden tener o no parámetros y no regresan resultado. La declaración de un constructor puede estar acompañada de alguno de los modificadores ya analizados (*private, protected, public*). Si no se indica ninguno, se establece por omisión como *public*.

Los constructores pueden ser por omisión o con parámetros. Los primeros hacen referencia a constructores que no reciben parámetros y que, normalmente, no tienen instrucciones en su cuerpo. Estos constructores sólo crean el objeto. Los segundos reciben parámetros y sus valores se asignan a los atributos del objeto. Por tanto, estos constructores no sólo crean el objeto, sino que también lo inicializan. En algunos casos, en el constructor por omisión se puede asignar valores por omisión a los atributos.

En cada clase debe haber al menos un constructor. Si no se especifica, entonces Java agrega automáticamente un constructor por omisión. Este tipo de constructores no tiene parámetros, ni instrucciones en su cuerpo. Es decir, es un método vacío.

Cuando en una clase se incluye un constructor con parámetros, entonces el constructor por omisión de Java desaparece. Si se quiere mantener, se debe definir explícitamente en la clase.

Muy importante

- ☑ Los constructores se llaman igual que la clase.
- ☑ Una clase puede tener más de un constructor, en cuyo caso deben tener distintos parámetros para distinguirse entre sí.
- ☑ Los constructores no regresan resultados.
- ☑ Si en la clase no se incluye un constructor, Java agrega uno por omisión. Éste desaparece en el momento que el programador incluye uno con parámetros.
- ☑ En general, es conveniente incluir un constructor por omisión y uno o más constructores con parámetros (según los posibles usos previstos al diseñar la clase).

A continuación, retomando la clase del programa 2.1, se presentan algunos ejemplos de constructores.[4]

```
/**
 * @Constructor  por omisión. El constructor no recibe parámetros y no contiene instrucciones.
*/
public Alumno(){
}

/**
 * @Constructor con parámetros. Recibe dos datos tipo cadena de caracteres y un número.
 * Contiene las instrucciones requeridas para asignar los valores recibidos a través de los
 * parámetros a los atributos de la clase.
*/
public Alumno(String nom, String car, double prom){
   nombre = nom;
   carrera = car;
   promedio = prom;
}

/**
 * @Otro constructor con parámetros. Recibe un dato tipo Alumno. Invoca, por medio del
 * this y de los parámetros requeridos, al constructor anterior.
*/
```

[4] En el constructor que recibe tres parámetros no se necesita asociar a cada nombre de atributo la palabra *this* porque los parámetros tienen un nombre distinto al atributo. Sin embargo, en el constructor que sólo recibe el nombre del alumno, como el parámetro se llama igual que el atributo, se debe usar el *this* junto al atributo.

```
public Alumno(Alumno alum){
   this(alum.nombre, alum.carrera, alum.promedio);
}


/**
 * @Otro constructor con parámetros. Recibe un dato tipo String.
 * Se crea un objeto asignando valor sólo al atributo nombre.
*/
public Alumno(String nombre){
   this.nombre = nombre;
}
```

Para declarar e instanciar objetos a partir de los constructores dados en el ejemplo, se procede de la siguiente manera:

```
Alumno alumLic = new Alumno( );
```

En este caso la variable *alumLic* referencia a un objeto tipo *Alumno*. Al declarar e instanciar el objeto se invoca al constructor por omisión. Los atributos *nombre*, *carrera* y *promedio* almacenarán los valores *null*, *null* y *0*, respectivamente, ya que son los valores que Java asigna por omisión.

```
Alumno alumIng = new Alumno("Juan del Campo", "Ingeniería", 8.6);
```

En este caso la variable *alumIng* referencia a un objeto tipo *Alumno*. Al declarar e instanciar el objeto se invoca al constructor que tiene dos cadenas y un número como parámetros. Los atributos *nombre*, *carrera* y *promedio* toman los valores *Juan del Campo*, *Ingeniería* y *8.6*, respectivamente.

```
Alumno otroAlum = new Alumno(alumIng);
```

En este caso la variable *otroAlum* referencia a un objeto tipo *Alumno*. Al declarar e instanciar el objeto se invoca al constructor que tiene un objeto de su mismo tipo como parámetro. Por tanto, los atributos *nombre*, *carrera* y *promedio* toman los mismos valores que tienen dichos atributos en el objeto dado como argumento.

```
Alumno alumno = new Alumno("Lucía Gómez");
```

En este caso la variable *alumno* referencia a un objeto tipo *Alumno*. Al declarar e instanciar el objeto se invoca al constructor que tiene sólo una cadena como parámetro. Por tanto, el objeto tiene asignado un valor en el atributo *nombre*, quedando los otros dos indefinidos, por el momento.

Si se quiere asignar valores por omisión a los atributos, se puede hacer en la declaración de los mismos. Siguiendo con el ejemplo de la clase *Alumno*, se podría asignar al nombre la cadena vacía, es decir, a la carrera el nombre *Ingeniería* y al promedio un 0. Así, al declarar e instanciar un objeto con el constructor por omisión, se asignarán estos valores a los atributos del objeto creado, en lugar de los dados por Java por omisión.

```
public class Alumno{
    private String nombre = "";
    private String carrera = "Ingeniería";
    private double promedio = 0.0;
    ...
}
```

En general, se recomienda que si la asignación es simple, como en el ejemplo, se haga en la declaración o en el constructor por omisión. En cambio, si requiere algún procesamiento, entonces deberá incluirse necesariamente en el constructor.

2.2.4 Ejemplo de una clase en Java

A continuación se presenta un ejemplo de una clase implementada en Java en la cual se incluyen, además de los atributos, los constructores y algunos métodos. En la figura 2.5 se muestra el diagrama UML de la clase *Rectángulo* formada por dos atributos numéricos –cada uno de los lados que definen a un rectángulo–, dos constructores y dos métodos, uno para el cálculo del área y otro para el cálculo del perímetro.

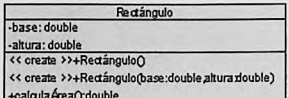


Figura 2.5 Clase *Rectángulo*

En el programa 2.2 se presenta el código correspondiente a la *clase* de la figura 2.5. A los atributos se les da el valor *0* por omisión, es decir, cuando se instancia un objeto con el constructor por omisión. También la clase tiene un constructor con parámetros, el cual permite que al instanciar un objeto se le asigne valores específicos a los atributos del mismo. Por último, se definieron dos métodos que corresponden a operaciones válidas sobre los atributos de la clase. En este caso, el cálculo del área y del perímetro del rectángulo.

Programa 2.2 **Rectángulo.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.2
 * Se define un rectángulo por medio de su base y de su altura. Además, se incluyen
 * métodos para el cálculo del área y del perímetro.
 */

public class Rectángulo{
   private double base;
   private double altura;

   /**
    * @Constructor por omisión. Los atributos toman los valores dados por omisión.
    */
   public Rectángulo(){
      base = 0.0;
      altura = 0.0;
   }

   /**
   * @Constructor con parámetros. Los atributos toman los valores dados a través
   * de los parámetros.
   */
   public Rectángulo(double base, double altura){
      this.base = base;
      this.altura = altura;
   }

   /**
   * Calcula el área de un rectángulo.
   */
   public double calculaArea(){
      return base * altura;
   }

   /**
   * Calcula el perímetro de un rectángulo.
   */
   public double calculaPerímetro(){
      return 2 * (base + altura);
   }
}
```

Para ofrecer un ejemplo completo de definición de una clase y el uso de la misma se presenta el programa 2.3. Este programa es la solución de un problema particular, cuya descripción se da a continuación.

Descripción: Se tiene un salón en forma rectangular, de 7 metros de largo por 6.5 metros de ancho. Además, se cuenta con dos alfombras, también rectangulares, que se colocarán sobre el piso de dicho salón. Se desea saber qué parte del piso quedará cubierta y qué parte no, para ayudar a decidir si se compran o no otras alfombras.

Programa 2.3 | **Alfombras.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.3
 * Programa de aplicación de la clase Rectángulo. Se representan las alfombras
 * y el piso del salón como rectángulos, y usando el método calculaArea( ) se
 * calculan los datos solicitados: total de m2 del salón que quedan cubiertos
 * por las alfombras y total de m2 que no alcanzan a cubrirse.
 */

public class Alfombras{

  public static void main (String [ ] args){

     Rectángulo alfom1, alfom2, piso;
     double areaCub, areaDescub;

     // Instanciación de los objetos, usando el constructor con parámetros.
     alfom1 = new Rectángulo(3.8, 4.6);
     alfom2 = new Rectángulo(4.5, 2.3);
          piso = new Rectángulo(7.0, 6.5);

     // Uso del método de la clase Rectángulo para calcular área.
     areaCub = alfom1.calculaArea( ) + alfom2.calculaArea( );  // Área cubierta
     areaDescub = piso.calculaArea( ) - areaCub;              // Área sin cubrir

     System.out.println("\nÁrea del salón cubierta con alfombras: " + areaCub + " m2");
     System.out.println("\nÁrea del salón sin cubrir con alfombras: " + areaDescub + " m2\
     n\n");
     }
}
```

Al ejecutar el programa 2.3 se obtienen los siguientes resultados:

Área del salón cubierta con alfombras: 27.83 m²

Área del salón sin cubrir con alfombras: 17.67 m²

2.2.5 Miembros estáticos de una clase

Hemos visto que a partir de una clase se crean objetos y que cada objeto tiene sus propias instancias de los atributos de la clase, ocupando estos diferentes posiciones de memoria. Sin embargo, en algunos casos se necesita que uno o varios de sus atributos sean comunes a todos los objetos. Esto se logra declarando dichos miembros como estáticos, usando el *modificador static*. Los miembros así declarados reciben el nombre de *miembros estáticos* o *variables de clase* y se asocian con la clase, no con los objetos. Un miembro estático puede ser manipulado por cualquiera de los objetos y también directamente a través de la clase, sin crear objetos. Esto último se hizo en el capítulo anterior, al declarar los subprogramas como métodos estáticos de una clase. Para ejecutar dichos métodos, estando en la misma clase, sólo se usan sus nombres y parámetros, si los tienen. Si los métodos fueron definidos en una clase distinta de la cual se los quiere emplear, se debe seguir la sintaxis:

NombreClase.nombreMétodo( )

Con respecto a los atributos estáticos, analicemos el siguiente ejemplo. Supongamos que se crean cuentas bancarias y a cada una se le asigna un número –a medida que se crea– comenzando con el 1. Se requiere un atributo que tome un valor distinto para cada objeto, de tal manera que cada cuenta tenga un número que la identifique. Por otra parte, se necesita llevar el número de cuentas creadas para saber cuál es el siguiente número a asignar. El primero de los valores será una variable de instancia, mientras que el segundo será una variable de clase. Observe el siguiente código, que presenta un ejemplo de atributo estático.

```
/**
 * @author Silvia Guardati
 * Ejemplo de atributo estático
 */
public class Cuenta{
   private static int totalCuentas = 0; // Atributo estático
   private int numCta;
   private String nomTitular;
   private double saldo = 0.0;

   public Cuenta(){
      numCta = ++totalCuentas; // Se incrementa la variable y luego se asigna
   }
```

```
    public Cuenta(String nom, double sal){
        this();
        nomTitular = nom;
        saldo = sal;
    }

    public String toString(){
        return "Titular: " + nomTitular + "\nNúmero de cuenta: " + numCta;
    }
...
}
```

Al crearse el primer objeto tipo *Cuenta*, se le asignará el valor 1. Si posteriormente se crea otra cuenta se le dará el valor 2, y así sucesivamente.

Siguiendo con el ejemplo, se le puede incluir un método estático que permita conocer el total de cuentas que se emitieron:

```
public static int getTotalCuentas( ) {
    return totalCuentas;
}
```

Para invocar este método no se requiere crear un objeto de la clase, sino que se llama por medio del nombre de la clase (recuerde que es un miembro estático asociado a la clase, no a los objetos).

```
n = Cuenta.getTotalCuentas( );
```

Asimismo, se pueden declarar constantes estáticas, anteponiendo la palabra *static* a la palabra *final*. Es importante tener en cuenta que se les debe asignar un valor en la declaración o en el constructor y que el mismo no podrá cambiar durante la ejecución. Por ejemplo:

```
private static final TOTAL_ESTADOS = 32;
```

2.2.6 Otros modificadores de una clase – Anidación de clases

Al definir una clase se puede usar los modificadores *private* y *protected*, siempre que la clase se declare dentro de otra clase. A este tipo de clases se les conoce como **clases interiores o clases anidadas**, y a las clases que las contienen se les llama **clases envolventes o exteriores**.

Las clases interiores pueden ser declaradas como *private*, *protected* y *public*. Si es el primer modificador, entonces la clase podrá ser usada sólo dentro de la clase envolvente. Si es de tipo *protected*, entonces podrá ser usada en la clase *envolvente*, en cualesquiera de sus clases derivadas y dentro de clases del mismo paquete.

Por último, si es *public* no se restringe su uso. Las instancias de la clase interior se crean a través de objetos de la clase envolvente. Es importante tener en cuenta que objetos de las clases interiores sólo existen dentro de instancias de las clases envolventes. Se emplea la siguiente notación:

```
ClaseEnvolvente.ClaseInterior objInterior = objExterior.new ClaseInterior( );
```

siendo objExterior un objeto previamente instanciado de la ClaseEnvolvente.

Las clases anidadas permiten agrupar clases relacionadas por los conceptos que representan y aumentan el encapsulamiento. En ciertos casos pueden ayudar a mantener el código más legible y, por lo tanto, más mantenible.

En de una clase interior se tiene acceso a todos los miembros de la clase envolvente, aunque estos sean privados. Por el contrario, las clases envolventes no tienen acceso a los miembros de sus clases interiores, aunque estos sean públicos.

Por otra parte, a una clase interior se la puede declarar *estática* (*static*). En este caso la clase pasa a ser un miembro estático de la clase envolvente. Una clase estática no tiene acceso a los miembros de la clase envolvente. Sin embargo, ofrece la facilidad de que pueden crearse objetos sin depender de objetos de la clase envolvente. En estos casos, la anidación de clases se justifica porque promueve el encapsulamiento de clases. La sintaxis para crear objetos de la clase estática es:

```
ClaseEnvolvente.ClaseEstática objEstática = new ClaseEnvolvente.ClaseEstática( );
```

La palabra *this* empleada en una clase interior hace referencia al objeto interior. Para referenciar al objeto de la clase envolvente se escribe: ClaseEnvolvente.this.

Analice los siguientes programas, 2.4 y 2.5. En el primero de ellos se definieron una clase envolvente, tres anidadas (una con cada modificador) y tres estáticas (una con cada modificador). El ejemplo, aunque sencillo, pretende ilustrar todos los conceptos explicados más arriba.

Muy importante

- ☑ Clase interior privada, sólo se conoce dentro de la clase envolvente.
- ☑ Clase interior protegida o pública, se conoce dentro de la clase envolvente y de sus clases derivadas.
- ☑ Objetos de la clase interior, sólo existen dentro de objetos de la clase exterior.
- ☑ La clase interior tiene acceso a todos los miembros de la clase exterior.
- ☑ La clase exterior no tiene acceso a los miembros de la clase interior, aunque estos sean públicos.

Programa 2.4 **ClaseEnvolvente.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.4
 * Ejemplo de clases anidadas y estáticas. Definición y acceso a sus miembros.
 */
public class ClaseEnvolvente {

  private int ate1;

  public ClaseEnvolvente() {
  }

  public ClaseEnvolvente(int ate1) {
    this.ate1 = ate1;
  }

  /* Este método crea una instancia de la clase ClaseInterior1 y regresa una
   * cadena con los datos del objeto creado.
   */
  public String haceAlgo(){
    ClaseInterior1 objCI = new ClaseInterior1(1);
    return "\nDesde ClaseEnvolvente se imprime un objeto de ClaseInterior1: " + objCI;
  }
  /* Este método crea una instancia de la clase ClaseEstática1 -que es estática y
   * privada. Regresa una cadena con los datos del objeto creado.
   */
  public String haceOtraCosa(){
    ClaseEstática1 objCEst1 = new ClaseEstática1();
    return "\nDesde ClaseEnvolvente se imprime un objeto de ClaseEstática1: " + objCEst1;
  }

  @Override
  public String toString() {
    return "\nEl atributo de ClaseEnvolvente: " + ate1;
  }

  /* Se define una clase anidada y privada, por lo tanto sólo podrá ser usada
   * dentro de la clase envolvente o exterior. Analice el método haceAlgo().
   */
  private class ClaseInterior1 {
    private int ati1;
```

```
    private ClaseInterior1() {
    }

    private ClaseInterior1(int ati1) {
      this.ati1 = ati1;
    }

    // Observe que se tiene acceso al atributo de la clase envolvente que es privado.
    public String toString() {
      return "\nAtributo de ClaseInterior1: " + ati1 + " - atributo de la clase exterior: " + ate1;
    }
}

/* Se define una clase anidada y protegida, por lo tanto podrá ser usada
 * dentro de la clase envolvente y sus derivadas.
 * El acceso a un objeto de este tipo será a través de un objeto de la clase exterior.
 * Analice el código del programa 2.5.
 */
protected class ClaseInterior2{
      private int ati2;

      protected ClaseInterior2() {
      }

      protected ClaseInterior2(int ati2) {
        this.ati2 = ati2;
      }

      public String toString() {
      return "\nAtributo de ClaseInterior2: " + ati2;
      }

      /* Regresa el objeto de la clase exterior por medio del cual se tuvo acceso
       * a esta clase.
       */
      protected ClaseEnvolvente getClaseEnvolvente(){
        return ClaseEnvolvente.this;
      }

      // Regresa el objeto de la clase interior.
      protected ClaseInterior2 getClaseInterior2(){
        return this;
      }
}
```

```
/* Se define una clase anidada y pública.
 * El acceso a un objeto de este tipo será a través de un objeto de la clase exterior.
 * Analice el código del programa 2.5.
 */
public class ClaseInterior3{
      private int ati3;

      public ClaseInterior3() {
      }

      public ClaseInterior3(int ati3) {
         this.ati3 = ati3;
      }

      public String toString() {
      return "\nAtributo de ClaseInterior3: " + ati3;
      }
}

/* Se define una clase estática y privada. Analice el código del método haceOtraCosa()
 * y del programa 2.5.
 */
private static class ClaseEstática1{
      private int ats1;

      private ClaseEstática1() {
      }

      private ClaseEstática1(int ats1) {
         this.ats1 = ats1;
      }

      public String toString() {
         return "\nAtributo de ClaseEstática1: " + ats1;
      }
}

// Se define una clase estática y protegida. Analice el código del programa 2.5.
protected static class ClaseEstática2{
      private int ats2;

      protected ClaseEstática2() {
      }
```

```
        protected ClaseEstática2(int ats2) {
            this.ats2 = ats2;
        }

        public String toString() {
        return "\nAtributo de ClaseEstática2: " + ats2;
        }
    }

    // Se define una clase estática y pública. Analice el código del programa 2.5.
    public static class ClaseEstática3{
        private int ats3;

        public ClaseEstática3() {
        }

        public ClaseEstática3(int ats3){
            this.ats3 = ats3;
        }

        public String toString() {
            return "\nAtributo de ClaseEstática3: " + ats3;
        }
    }
}
```

Programa 2.5 | **PruebaClasesAnidadas.java**

```
package cap2;

// Se importa la clase ClaseEstática3 para simplificar la notación.
import cap2.ClaseEnvolvente.ClaseEstática3;

/**
 * @author Silvia Guardati
 * Programa 2.5
 * Ejemplo de uso de clases anidadas y estáticas definidas en el programa 2.4.
 */
public class PruebaClasesAnidadas {

    public static void main(String[ ] args) {
        ClaseEnvolvente objCE1, objCE2;
        objCE1 = new ClaseEnvolvente(50);
```

```
// Se imprime el objeto recién creado
System.out.println("\nDesde el main: " + objCE1);

/* Al involcar el método haceAlgo( ) se creará un objeto de tipo ClaseInterior1
 * y se imprimirá. La clase ClaseInterior1 se declaró como privada, por lo tanto
 * no se tiene acceso a ella fuera de la clase envolvente.
 */
System.out.println(objCE1.haceAlgo( ));

/* Por medio de un objeto de la clase envolvente se crea un objeto de una de sus
 * clases anidadas. En este caso, la clase protegida ClaseInterior2.
 */
ClaseEnvolvente.ClaseInterior2 objCI2 = objCE1.new ClaseInterior2(2);
System.out.println("\nDesde el main: " + objCI2);

/* Por medio de un objeto de la clase envolvente se crea un objeto de una de sus
 * clases anidadas. En este caso, la clase pública ClaseInterior3.
 */
ClaseEnvolvente.ClaseInterior3 objCI3 = objCE1.new ClaseInterior3(3);
System.out.println("\nDesde el main: " + objCI3);

/* Como no se importó la clase ClaseEstática2, se debe seguir la notación
 * ClaseEnvolvente.ClaseEstática2.
 */
ClaseEnvolvente.ClaseEstática2 objCEst2 = new ClaseEnvolvente.ClaseEstática2(20);
System.out.println("\nDesde el main: " + objCEst2);

/* Como se importó la clase ClaseEstática3, se puede omitir el nombre de la
 * clase envolvente, al declarar e instanciar el objeto.
 */
ClaseEstática3 objCEst3 = new ClaseEstática3(10);
System.out.println("\nDesde el main: " + objCEst3);

/* La clase ClaseEstática1 no puede usarse directamente en este programa
 * porque es privada. Se puede crear un objeto de su tipo por medio de un
 * objeto de la clase envolvente. Se invoca el método haceOtraCosa( ), el
 * cual crea un objeto y lo regresa en forma de cadena.
 */
System.out.println("\nDesde el main: " + objCE1.haceOtraCosa( ));

/* Este método regresa el objeto de la clase envolvente usado para instanciar
 * el objeto de la clase ClaseInterior2.
 */
objCE2 = objCI2.getClaseEnvolvente( );
System.out.println("\nDesde el main: " + objCE2);
```

```
        // Este método regresa el objeto de la clase ClaseInterior2.
        System.out.println("\nDesde el main: " + objCI2.getClaseInterior2());
    }
}
```

En el programa 2.6 se presenta un ejemplo de una clase (exterior) con una clase anidada. La primera representa a un polígono regular y la otra a un triángulo. Esta última se utiliza como auxiliar para el cálculo del área del polígono. En este caso, anidando las clases, se están encapsulando clases relacionadas.

Programa 2.6 **PolígonoRegular.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.6
 * Ejemplo de clases anidadas. Se declara la clase Polígono con una clase interna,
 * Triángulo, la cual se usará como auxiliar para el cálculo del área.
 */
public class PolígonoRegular {
  /* Un polígono regular es un polígono que tiene todos sus lados iguales.
   * En esta representación, un polígono queda definido por el total de lados,
   * el tamaño de sus lados y por el apotema (segmento que une el centro y la
   * mitad de cada lado del polígono).
   */
  private int totalLados;
  private double lado, apotema;

  public PolígonoRegular() {
  }

  public PolígonoRegular(int totalLados, double lado, double apotema) {
    this.totalLados = totalLados;
    this.lado = lado;
    this.apotema = apotema;
  }

  /* El área de un polígono regular se puede calcular a partir del área de los
   * n triángulos que lo forman, siendo n el total de lados del polígono.
   */
  public double calculaÁrea(){
    double área;
    Triángulo triangInterno = new Triángulo(lado, apotema);
```

```
        área = totalLados * triangInterno.calculaÁrea();
        return área;
      }

      /* Clase interna a la clase Polígono. En este caso se encapsula dentro de la
       * clase exterior y, al ser privada, no podrá ser usada fuera de ella.
       */
      private class Triángulo {
        private double base, altura;

        private Triángulo(double base, double altura) {
          this.base = base;
          this.altura = altura;
        }

        private double calculaÁrea(){
          return base * altura / 2.0;
        }
      }
    }
```

2.3 SOBRESCRITURA Y SOBRECARGA

La sobrescritura y la sobrecarga son facilidades que ofrece la programación orientada a objetos y ayudan a ganar generalidad al codificar. Estas actividades permiten que un método tenga asociadas distintas funcionalidades, dependiendo de los datos que se le proporcionen. Existen algunas diferencias entre estas dos operaciones que deben tenerse en cuenta.

- **Sobrescribir:** se sobrescribe un método cuando se usa el mismo nombre, se recibe la misma lista de parámetros y se obtiene el mismo tipo de resultado. Lo único que cambia es el proceso que contiene el método. El efecto que se logra es que el método se usa siempre en el mismo contexto independientemente de lo que haga. Un ejemplo muy útil es la sobrescritura del método *toString()* que pertenece a la clase *Object*, de Java. Este método no recibe parámetros y regresa una cadena de caracteres *(String)* como resultado. El siguiente código es la sobrescritura de este método dentro de la clase *Rectángulo*, vista anteriormente.

```
/**
* Sobrescritura del método toString().
* @return String
*/
```

```
public String toString(){
  String cad = "";

  cad = "\nLado 1: " + lado1 + "\nLado 2: " + lado2 + "\n\n";
  return cad;
}
```

En el método se declara un objeto de la clase *String* y se inicializa con la cadena vacía. Luego se le asigna una concatenación de cadenas –salto de línea y el texto *Lado i:*– y valores de los atributos –*lado1 y lado2*–. Si se quisiera desplegar en pantalla los valores del objeto *alfom1*, se haría:

```
System.out.println (alfom1);
```

lo cual daría como resultado:

```
Lado 1: 3.8

Lado 2: 4.6
```

A continuación se muestra otro ejemplo. Se retoma la clase *Alumno* del programa 2.1 y se sobrescribe el método *toString()*, para de poder desplegar en pantalla fácilmente los valores de sus atributos. En este caso no se hace uso de un objeto auxiliar tipo *String*, sino que se regresa directamente el resultado de la concatenación de las cadenas con los contenidos de los atributos.

```
/**
* Sobrescritura del método toString().
* @return String
*/
public String toString (){
  return "\n" + nombre + "  " + carrera + "  " + promedio + "\n";
}
```

Una vez sobrescrito el método, en un programa de aplicación de la clase *Alumno* se puede declarar e instanciar el objeto *alumIng* por medio de la instrucción que se muestra a continuación:

```
Alumno alumIng = new Alumno("Juan del Campo", "Ingeniería", 8.6);
```

Para imprimir el objeto *alumIng* se usa la instrucción que aparece más abajo:

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

cuyo resultado es la siguiente salida en pantalla:

```
Juan del Campo  Ingeniería  8.6
```

Es importante volver a señalar la flexibilidad que representa la sobrescritura. En los ejemplos analizados el código para desplegar un objeto tipo *Rectángulo* es el mismo que para desplegar un objeto tipo *Alumno*. Esto permite que el ingeniero pueda concentrarse en la solución del problema y deja el detalle de la implementación a la clase correspondiente.

```
System.out.println (alumIng); //alumIng objeto de la clase Alumno

System.out.println (alfom1); //alfom1 objeto de la clase Rectángulo
```

El método *equals* se puede sobrescribir, respetando la firma que tiene en la clase *Object*. Para ello, debe recibir como parámetro un dato tipo *Object* y en el método se debe convertir a la clase (en la que está), para luego hacer la comparación. Como se requieren conceptos no vistos todavía, se volverá sobre este punto posteriormente, en este mismo capítulo.

- **Sobrecargar:** se sobrecarga un método cuando se usa el mismo nombre, pero se recibe una lista diferente de parámetros y puede obtenerse o no el mismo tipo de resultado. Por tanto, el proceso implícito en el método se aplica a distintos tipos de datos, lo cual determina que el mismo varíe. El efecto que se logra es que el método se usa sobre elementos de diferente naturaleza, pudiendo dar, además, resultados de distintos tipos. Ejemplos muy ilustrativos y útiles son la sobrecarga de los métodos *equals()*, que pertenece a la clase *Object*, y *compareTo()*, que pertenece a la interface *Comparable*, ambos del lenguaje Java. El primero de ellos generalmente se sobrecarga para determinar si dos objetos de la misma clase son iguales, dando un valor verdadero, si lo son, o falso en caso contrario. Por su parte, el método *compareTo()* se sobrecarga para comparar dos objetos de la misma clase, dando como resultado un 0 si son iguales, un entero positivo si el primero es mayor que el segundo, o un entero negativo en otro caso. A continuación se presenta el código correspondiente a la sobrecarga de estos métodos dentro de la clase *Alumno* y *Rectángulo*, respectivamente.

```
/**
* Sobrecarga del método compareTo( ). El método determina si un objeto tipo Alumno
* es mayor, igual o menor que otro objeto del mismo tipo. La comparación sólo se
* lleva a cabo con el atributo nombre, y será muy útil para manejar colecciones
* ordenadas de objetos.
* @return int
*/
public  int compareTo(Alumno a){
   return nombre.compareTo(a.nombre);
}
```

El resultado arrojado por este método es un entero positivo, negativo o igual a cero, según el nombre del objeto sea mayor, menor o igual que el nombre del objeto dado como parámetro. La comparación de los nombres se hace de acuerdo a los valores del código ASCII. Sin embargo, es importante destacar que la comparación puede hacerse sobre atributos que sean numéricos, en cuyo caso se usan los operadores relacionales "<", ">","==".

```
/**
* Sobrecarga del método equals( ). El método determina si dos objetos del tipo
* Rectángulo son iguales. Regresa true si lo son y false en caso contrario.
* @return boolean
*/
public boolean equals(Rectangulo r){
   boolean resp;

   resp = ((r.lado1 == this.lado1 &&  r.lado2 == this.lado2) ||
           (r.lado1 == this.lado2 &&  r.lado2 == this.lado1));
   return resp;
}
```

En el programa 2.3 puede incluirse la siguiente instrucción para comparar y determinar si dos alfombras tienen el mismo tamaño. En este caso, el parámetro *r* toma el valor del objeto *alfom2* y los atributos *lado1* y *lado2*, junto al *this*, son del objeto *alfom1*, dado que a éste se le asoció el método.

```
if (alfom1.equals(alfom2))
   System.out.println ("\n\nLas alfombras tienen el mismo tamaño\n");
else
   System.out.println ("\n\nLas alfombras NO tienen el mismo tamaño\n");
```

Considerando los valores con los cuales se instanciaron los objetos *alfom1* y *alfom2*, la salida del programa será:

```
Las alfombras NO tienen el mismo tamaño
```

Al presentar la sobrecarga se señaló que un método sobrecargado puede o no dar el mismo tipo de resultado. A continuación se presenta el método *equals()* sobrecargado de dos maneras distintas, variando el tipo de resultado que arroja. En ambos casos el objetivo del método es el mismo, es decir, determinar si dos objetos tipo *Alumno* son iguales. La primera de las implementaciones no se recomienda, ya que altera la naturaleza misma del método; sólo se da para ilustrar este concepto.

```
/**
* Sobrecarga del método equals( ). Regresa un 1 si el nombre, carrera y promedio de
* los dos alumnos son iguales, y un 0 en caso contrario.
* @return int
*/
public  int equals(Alumno a){
   int resp = 0;
```

```
    if (a.nombre.equals(nombre) &&  a.carrera.equals(carrera) && a.promedio == promedio)
      resp =  1;
    return resp;
  }

  /* *
  * Sobrecarga del método equals( ). Regresa true si el nombre, carrera y promedio de
  * los dos alumnos son iguales, y false en caso contrario.
  *  @return boolean
  * */
  public  boolean equals(Alumno a){
    return (a.nombre.equals(nombre) &&  a.carrera.equals(carrera) &&
            a.promedio == promedio);
  }
```

En el ejemplo que se muestra a continuación se sobrecarga el método *suma*, de tal manera que puede sumar números enteros, números de doble precisión y números complejos.[5] Observe que los métodos se usan de manera indistinta para cada uno de los tipos de datos.

```
// Método que regresa la suma de los números enteros recibidos
  public static int suma(int num1, int num2){
    return num1 + num2;
  }

  // Método que regresa la suma de los números recibidos
  public static double suma(double num1, double num2){
    return num1 + num2;
  }

  // Método que regresa la suma de los números complejos recibidos
  public static Complejo suma(Complejo num1, Complejo num2){
    double real, imag;
    real = num1.getReal( )+ num2.getReal( );
    imag = num1.getImaginario( ) + num2.getImaginario( );
    Complejo res = new Complejo(real, imag);
    return res;
  }

  // Ejemplo de uso del método "suma" sobrecargado
    System.out.println("\nSuma de enteros: " + suma(12, 3));
    System.out.println("\nSuma de reales: " + suma(12.3, 8.5));
```

[5] La clase *Complejo* puede consultarse en el programa 2.30, Complejo.java.

```
Complejo c1 = new Complejo(4.2,8.7);
Complejo c2 = new Complejo(5.1,18.7);
System.out.println("\nSuma de Complejos: " + suma(c1, c2));
```

La tabla 2.1 presenta un resumen de las diferencias y de las similitudes más importantes entre estos dos conceptos.

Tabla 2.1 Similitudes/diferencias entre sobrescribir y sobrecargar

	Sobrescribir	Sobrecargar
Nombre del método	Igual	Igual
Lista de parámetros	Igual	Diferente
Tipo de resultado	Igual	Puede ser igual o no

2.4 INTERFACES

Una interface describe el comportamiento deseado para una o varias clases. Por tanto, está formada por un conjunto de especificaciones de métodos, debiendo incluir los parámetros y la salida, pero no la implementación de los mismos. Las clases, usadas para implementar las interfaces, son las encargadas de desarrollar cada uno de los métodos. Consecuentemente, las interfaces no se instancian. Es decir, no se tendrán variables declaradas e instanciadas con el tipo de las interfaces, sino que se trabaja con las clases definidas a partir de ellas.

En las interfaces también se pueden declarar constantes. Cada una de las clases que implementa a la interface tendrá acceso a las mismas. De esta manera se permite compartir un conjunto de valores por varias clases, definiéndolos sólo en la interface.

Muchas clases, aunque no estén relacionadas entre sí, pueden implementar la misma interface, codificando los métodos de acuerdo con las necesidades propias de cada una de ellas. De esta forma, una interface constituye una especie de "contrato" entre clases. Asegura que todas las clases concretas tengan los métodos que formalmente se declararon en ella, pero les deja "la libertad" para implementarlos según las características particulares de cada una.

Una clase puede implementar varias interfaces. En este caso deberá implementar todos los métodos de dichas interfaces. Se usa el operador *instanceof* para determinar si un objeto de una clase está implementando una interface dada.

Muy importante

- ☑ Las interfaces están formadas por declaraciones formales de métodos, los cuales no están implementados.
- ☑ Las interfaces también pueden incluir constantes.
- ☑ Las clases concretas deben implementar todos los métodos de la interface involucrada.
- ☑ Las interfaces no se instancian.
- ☑ Una clase puede implementar muchas interfaces.
- ☑ Una interface puede ser implementada por muchas clases.
- ☑ En el lenguaje UML se indica que una clase implementa a una interface por medio de una línea punteada.
- ☑ En Java se usa la palabra reservada *interface* para definir una interface.
- ☑ En Java se indica que una clase implementa a una interface por medio de la palabra *implements*.
- ☑ El nombre de la interface empieza con mayúscula.
- ☑ Cada *interface* se guarda en un archivo.

En Java las interfaces se declaran usando la palabra reservada *interface*. Para indicar que una clase implementa a una interface, se usa la palabra reservada *implements*. Si la clase implementa a más de una, entonces los nombres de las mismas se separan por coma.

En la figura 2.6 se presenta un ejemplo de interface, junto con una clase que la implementa. La interface *Mascota* contiene tres métodos *come()*, *juega()* y *duermeEn()*. De esta manera se está estableciendo que cualquier mascota debe desarrollar estas tres actividades. Es decir, se está describiendo el comportamiento esperado para cualquier mascota. La clase *Tortuga* implementa a la interface y, por lo tanto, debe implementar los tres métodos mencionados, de acuerdo con las características propias de las tortugas. Además de estos métodos, la clase incluye atributos y métodos propios. Los programas 2.7 y 2.8 muestran el código de la interface y de la clase respectivamente. Recuerde que la interface se guarda en un archivo y la clase en otro.

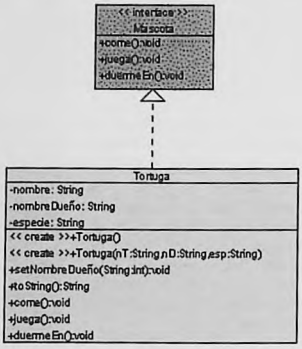


Figura 2.6 Ejemplo de interface

Programas 2.7 y 2.8 | **Mascota.java** | **Tortuga.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.7
 * Definición de una interface, la cual establece el comportamiento esperado para cualquier
 * mascota. Es decir, toda mascota debe comer, dormir en algún lugar y jugar.
 */
public interface Mascota {
  public void come();
  public void duermeEn();
```

```
    public void juega();
}


package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.8
 * Definición de la clase Tortuga que implementa la interface Mascota.
 */

public class Tortuga implements Mascota{
  private String nombre;
  private String nombreDueño;
  private String especie;

  public Tortuga(){
  }

  public Tortuga(String nom, String nomD, String esp){
    nombre = nom;
    nombreDueño = nomD;
    especie = esp;
  }

  /**
  @Se implementa el método de la interface.
  */
  public void come(){
    System.out.println("\nLas tortugas comen hojas de lechuga y pequeños peces.\n");
  }

  /**
  @Se implementa el método de la interface.
  */
  public void duermeEn(){
    System.out.println("\nLas tortugas duermen en peceras.\n");
  }

  /**
  @Se implementa el método de la interface.
  */
  public void juega(){
```

```
        System.out.println("\nLas tortugas no juegan con sus dueños.\n");
    }

    public void setNombreDueño(String nuevoNom){
        NombreDueño = nuevoNom;
    }

    public String toString(){
        StringBuilder cad = new StringBuilder();

        cad.append("\nNombre de la tortuga: " + nombre + " y le pertenece a: " + nombreDueño);
        cad.append("\n" + nombre + " es de la especie: " + especie);
        return cad.toString();
    }
}
```

Si en alguna clase aplicación se escribiera el siguiente código:

```
Tortuga miMascota = new Tortuga("Manuelita", "Matías", "japonesa");
System.out.println(miMascota);

miMascota.come();

miMascota.duermeEn();

miMascota.juega();
```

se generaría una salida como la que se muestra a continuación:

```
Nombre de la tortuga: Manuelita y le pertenece a: Matías
Manuelita es de la especie: japonesa

Las tortugas comen hojas de lechuga y pequeños peces.

Las tortugas duermen en peceras.

Las tortugas no juegan con sus dueños.
```

En la figura 2.7 se presenta un esquema de varias clases implementando una interface. Se retoma la interface *Mascota* y se declaran tres clases, *Tortuga*, *Gato* y *Perro*, que implementan a la misma. Por tanto, las tres clases deberán tener codificados los métodos enunciados en la interface, adaptándolos a las características propias del tipo de mascota que está describiendo cada una de ellas. Los programas 2.7, 2.8, 2.9 y 2.10 corresponden a este esquema.

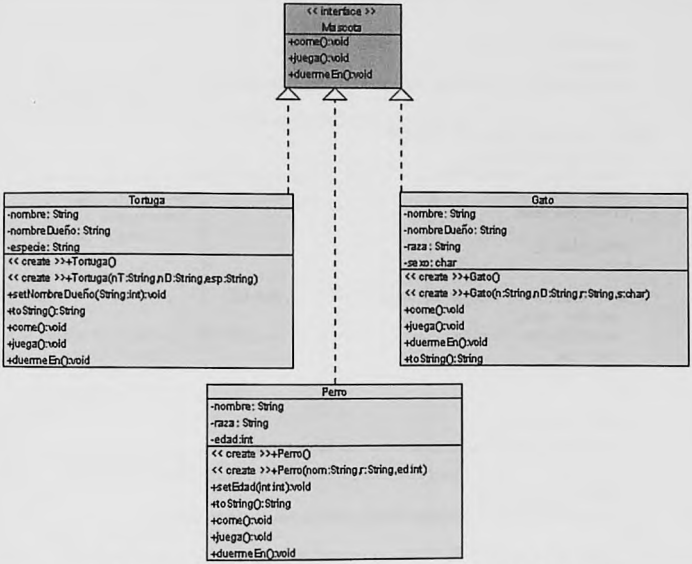


Figura 2.7 Ejemplo de varias clases implementando una interface

Programa Mascota.java, Tortuga.java | **Gato.java 2.9** | **Perro.java 2.10**

```
public interface Mascota{
/* El código es el presentado en el programa 2.7 */
}

public class Tortuga implements Mascota{
  /* El código es el presentado en el programa 2.8 */
}

package cap2;
```

```
/**
 * @author Silvia Guardati
 * Programa 2.9
 * Definición de la clase Gato que implementa la interface Mascota.
 */

public class Gato implements Mascota{
   private String nombre;
   private String nombreDueño;
   private String raza;
   private char sexo;

   public Gato(){
   }

   public Gato(String nom, String nomDue, String ra, char s){
      nombre = nom;
      nombreDueño = nomDue;
      raza = ra;
      sexo = s;
   }

   public void come(){
      System.out.println("\nLos gatos comen croquetas para gatos.");
   }

   public void duermeEn(){
      System.out.println("\nLos gatos duermen en sus camas o en un sillón.");
   }

   public void juega(){
      System.out.println("\nA los gatos les gusta jugar con ovillos de lana.");
   }

   public String toString(){
      StringBuilder cad = new StringBuilder();

      cad.append("\n\nNombre del gato: " + nombre + " y le pertenece a: " + nombreDueño);
      cad.append(cad + "\n" + nombre + " es: ");
         if (sexo == 'f' || sexo == 'F')
      cad.append("hembra ");
         else
      cad.append("macho ");
      cad.append("y de la raza: " + raza);
      return cad.toString();
   }
}
```

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.10
 * Definición de la clase Perro que implementa la interface Mascota.
 */

public class Perro implements Mascota{
   private String nombre;
   private String raza;
   private int edad;

   public Perro( ){
   }

   public Perro(String nom, String ra, int e){
      nombre = nom;
      raza = ra;
      edad = e;
   }

   public void come( ){
      System.out.println("\nLos perros comen croquetas para perros.");
   }

   public void duermeEn( ){
      System.out.println("\nLos perros duermen en sus camas o sobre la cama de su dueño.");
   }

   public void juega( ){
      System.out.println("\nA los perros les gusta jugar con pelotas y correr detrás de los niños.");
   }

   public String toString ( ){
      StringBuilder cad = new StringBuilder( );

      cad.append("\n\nNombre del perro: " + nombre);
      cad.append("\nEs de la raza: " + raza + " y tiene: " + edad + " años.");
      return cad.toString( );
   }
}
```

Si en una aplicación se escribe el siguiente código (vea el archivo *PruebaInterface.java*):

```
Tortuga miMascota = new Tortuga("Manuelita", "Julián", "japonesa");
System.out.println(miMascota);
miMascota.come( );
miMascota.duermeEn( );
miMascota.juega( );

Gato tuMascota = new Gato("Minino", "Mauricio", "Persa", 'm');
System.out.println(tuMascota);
tuMascota.come( );
tuMascota.duermeEn( );
tuMascota.juega( );

Perro suMascota = new Perro("Charrúa", "Labrador", 4);
System.out.println(suMascota);
suMascota.come( );
suMascota.duermeEn( );
suMascota.juega( );
```

se genera una salida como la que se muestra a continuación:

```
Nombre de la tortuga: Manuelita y le pertenece a: Julián
Manuelita es de la especie: japonesa

Las tortugas comen hojas de lechuga y pequeños peces.

Las tortugas duermen en peceras.

Las tortugas no juegan con sus dueños.


Nombre del gato: Minino y le pertenece a: Mauricio
Minino es: macho y de la raza: Persa

Los gatos comen croquetas para gatos.

Los gatos duermen en sus camas o en un sillón.

A los gatos les gusta jugar con ovillos de lana.


Nombre del perro: Charrúa
Es de la raza: Labrador y tiene: 4 años.
```

Los perros comen croquetas para perros.

Los perros duermen en sus camas o sobre la cama de su dueño.

A los perros les gusta jugar con pelotas y correr detrás de los niños.

En Java existe la Interface **Comparable** que tiene un único método llamado *compareTo()*, el cual regresa un número entero positivo, negativo o igual a 0. Es positivo cuando el objeto que invoca al método es mayor que el que se pasa como parámetro, negativo si es menor y cero cuando los dos son iguales.

$$\text{obj1.compareTo(obj2)} = \begin{cases} > 0 & \text{si obj1} > \text{obj2} \\ = 0 & \text{si obj1} = \text{obj2} \\ < 0 & \text{si obj1} < \text{obj2} \end{cases}$$

Cuándo usar interfaces...

- ☑ Para indicar, de manera general, el comportamiento esperado de una o más clases.
- ☑ Para declarar un conjunto de métodos que deben estar presentes en una o más clases.
- ☑ Para representar similitudes entre clases, sin necesidad de establecer relaciones entre ellas.
- ☑ Para simular herencia múltiple al declarar una clase que implementa a múltiples interfaces. El tema de herencia se tratará en la siguiente sección.
- ☑ Para determinar la interface de programación de un objeto, sin especificar los detalles de implementación.

El criterio para determinar si los objetos son mayores, menores o iguales entre sí depende de cada clase y del atributo (o los atributos) sobre los cuales se quiera establecer orden. Por ejemplo, si son alumnos, posiblemente el orden sea por la clave; en el caso de personas en general, podría ser por orden alfabético, según sus nombres. Implementar el método de la interface es muy útil cuando se requiere una colección ordenada de objetos de una cierta clase, ya que garantiza que dicha clase tiene un método *compareTo()*. Por tanto, los objetos de dicha clase podrán usarse en cualquier contexto donde se necesite manejar datos ordenados porque podrán ser comparados entre ellos.

El tema de las interfaces y su uso se retomará en los capítulos dedicados al estudio de las estructuras de datos.

2.5 HERENCIA

La *herencia* es la propiedad que permite compartir miembros entre clases. Además, por medio de ella se definen clases generales que pueden usarse como base para definir clases más específicas, sin tener que volver a incluir los miembros que se comparten. Por tanto, la herencia favorece la abstracción y generalización de ciertos conceptos y su posterior uso para diseñar elementos más específicos, estableciendo así jerarquía de conceptos. Asimismo, promueve el reuso, ya que las clases que heredan pueden hacer uso de los miembros heredados, sin tener que volver a codificarlos.

Se identifican tres tipos distintos de herencia según el número de clases que participen y la relación entre las mismas. La figura 2.8 presenta un esquema de cada uno de los tipos de herencia y en seguida se explican las principales características.

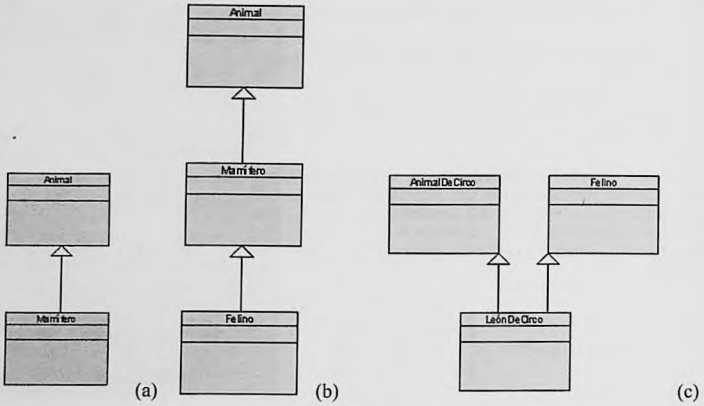


Figura 2.8 Tipos de herencia

a. **Herencia simple:** es la relación que se define entre dos clases, una de las cuales es la **clase base** o **súper clase** y la otra se denomina **subclase** o **clase derivada**. La súper clase representa el concepto más general, en este caso a todos los animales, mientras que la subclase representa a un subconjunto de los animales, los mamíferos. La clase derivada hereda todos los miembros de la clase base. Por tanto, un objeto de la clase *animal* tendrá sólo los atributos que puedan generalizarse a todos los animales; por ejemplo, son seres vivos que se mueven y otras características que les son propias. Por su parte, los mamíferos tienen todos los atributos de los animales –porque los heredan– más sus propios atributos que los distinguen de otros animales que no son mamíferos; por ejemplo, el número de glándulas mamarias.

b. **Herencia de múltiples niveles:** es la relación que se establece entre tres o más clases de manera lineal, definiendo así desde conceptos generales hasta conceptos cada vez más específicos. A medida que se desciende en las clases se logra un mayor grado de especificidad. En el ejemplo se define la clase *Animal*, haciendo referencia a todos los animales, luego *Mamífero* para representar a una subclase de animales y, por último, la clase *Felino*, que es una subclase de la clase *Mamífero*. De esta manera se está representando que los felinos son una clase derivada de los mamíferos, los cuales a su vez son una subclase de animales. Por tanto, un objeto de la última clase hereda los miembros de las otras dos y, además, puede tener atributos propios.

c. **Herencia múltiple:** es la relación que se establece entre tres o más clases de tal manera que una de ellas hereda de por lo menos otras dos. En este caso, la clase derivada tendrá, además de sus propios miembros, los heredados de todas sus súper clases. En el ejemplo de la figura 2.8, la clase *LeónCirco* hereda de *AnimalCirco*, que representa a todos los animales que se usan en los circos con fines de entretenimiento, y de *Felino*, que representa a todos los animales que pertenecen a esta familia. Por tanto, un objeto de la clase *LeónCirco* hereda las características de ambas clases y tendrá además sus propios atributos.

La herencia simple y la de múltiples niveles se pueden implementar fácilmente en Java. Sin embargo, la múltiple requiere el uso de otros recursos.

2.5.1 Herencia simple

La herencia simple es la que se establece entre dos clases cuando una de ellas comparte sus miembros con la otra. En este caso, a la clase que recibe los atributos y métodos se le da el nombre de subclase o clase derivada, y a la otra –de la cual se toman– clase base o súper clase.

En la figura 2.9 se retoma el ejemplo anterior, pero con más detalles en las especificaciones de las clases. La clase *Animal* tiene dos atributos, constructores y dos métodos. Por su parte, la clase *Mamífero* tiene un atributo, constructores, dos métodos propios y, además, todos los miembros heredados de la súper clase.

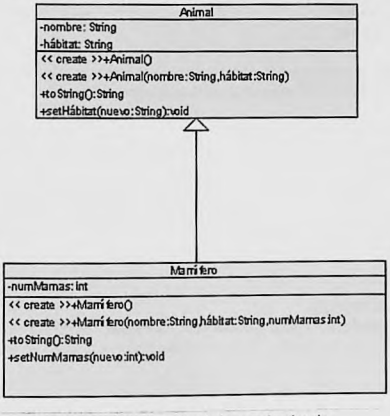


Figura 2.9 Ejemplo de herencia simple

Para indicar la relación de herencia entre clases se usa la palabra *extends*, junto al nombre de la súper clase en el encabezado de la clase derivada. Con respecto a los constructores, en el de la clase derivada se debe invocar –por medio de la palabra *super*– al constructor de la clase base. La elección del constructor queda determinada por los parámetros que se usen. En caso de no hacerlo, el compilador invoca al constructor por omisión. En consecuencia, si la súper clase no tuviera un constructor por omisión se produce un error de compilación. La llamada al constructor de la base debe ser la primera línea del constructor de la derivada. La implementación de la clase base no cambia.

La palabra *super* también se usa para invocar a miembros de la súper clase. Es muy útil cuando en la clase derivada se sobrescriben métodos de la clase base, ya que generalmente es para ampliar las capacidades de los mismos. Los programas 2.11 y 2.12 presentan el código correspondiente al diagrama de clases UML de la figura 2.9.

Programas 2.11 y 2.12 | **Animal.java** | **Mamífero.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.11
 * Definición de la clase Animal para ejemplificar la herencia simple.
 */

public class Animal {
   private String nombre;
   private String hábitat;

   public Animal(){
   }

   public Animal(String nom, String hab){
      nombre = nom;
      hábitat = hab;
   }

   public void setHábitat(String nuevo){
      hábitat = nuevo;
   }

   public String toString (){
      return "Nombre: " + nombre + "\nHabita en: " + hábitat + "\n";
   }
}
```

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.12
 * Definición de la calse Mamífero como clase derivada de la clase Animal.
 */
public class Mamífero extends Animal{
   private int numMamas;

/* Constructor por omisión. El compilador se encarga de invocar al constructor
 * por omisión de la súper clase.
 */
```

```
    public Mamífero(){
    }

  /* Constructor con parámetros. La primera línea de código corresponde a la
   * invocación del constructor con parámetros de la súper clase.
   */
    public Mamífero(String nom, String hab, int numM) {
      super(nom, hab);
      numMamas = numM;
    }

    public void setMamas(int num){
      numMamas = num;
    }

  /* Despliega los valores de los atributos de un mamífero, incluyendo aquellos que hereda
   * de la súper clase. Usa la palabra super para invocar un método de la clase base.
   */
    public String toString (){
      StringBuilder cad = new StringBuilder();

      cad.append(super.toString ());
      cad.append("Número de mamas: " + numMamas + "\n");
      return cad.toString();
    }
}
```

A continuación se presenta el código (programa 2.13) para ejemplificar la creación de objetos de las clases definidas en el programa anterior y el uso de los métodos.

Programa 2.13 | **UsaHerencia1.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.13
 * Ejemplo de herencia simple.
 */

public class UsaHerencia1 {
  public static void main (String [] args) {
```

```
        // Se crea un objeto tipo Animal y se le asignan valores a sus atributos.
        Animal  x = new Animal("mosca", "planeta Tierra");

        /* El constructor con parámetros de la clase Mamífero recibe los valores para instanciar
         * todos los atributos: los heredados y el específico de la subclase.
         */
        Mamífero cerda = new Mamífero("Rosita", "Granja", 14);

        System.out.println ("\n\nDatos del animal \n" + x);
        System.out.println ("\nDatos del mamífero\n" + cerda);

        /* Se asocia al objeto cerda –de la clase Mamífero– un método heredado
         * de la clase Animal.
         */
        cerda.setHábitat("Galpones de confinamiento");
        System.out.println ("\nDatos del mamífero\n" + cerda);

        /* Se asocia al objeto cerda –de la clase Mamífero– un método específico
         * de su clase.
         */
        cerda.setMamas(16);
        System.out.println ("\nDatos del mamífero\n" + cerda);
    }
}
```

Una vez ejecutada esta pequeña aplicación, en pantalla se verá desplegada la siguiente información:

```
Datos del animal
Nombre: mosca
Habita en: planeta Tierra

Datos del mamífero
Nombre: Rosita
Habita en: Granja
Número de mamas: 14

Datos del mamífero
Nombre: Rosita
Habita en: Galpones de confinamiento
Número de mamas: 14
```

Datos del mamífero
Nombre: Rosita
Habita en: Galpones de confinamiento
Número de mamas: 16

Los programas 2.14 y 2.15 presentan el código de otro ejemplo de herencia simple entre clases. En este caso, se tiene la súper clase *Persona* y la clase derivada *Estudiante*. En la primera se incluyen las características y comportamiento comunes a cualquier persona, sin importar el tipo de actividad que desempeña. En la segunda se tienen aquellos elementos que son propios de los alumnos universitarios. De esta manera, un alumno tiene todas las características de las personas más las específicas de su propia clase.

Programas 2.14 y 2.15 | **Persona.java** | **Estudiante.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.14
* Se define, de manera muy simple, el concepto Persona.
 */
public class Persona{
   private String nombre = "";
   private String fechaNacimiento = "";    //día-mes-año
   private String domicilio = "";
   private char sexo;

   public Persona(){

   }

   public Persona(String n, String fn, String d, char s){
      nombre = n;
      fechaNacimiento = fn;
      domicilio = d;
      sexo = s;
   }

   public String toString (){
      return "Nombre:\t" + nombre + "\nFecha Nacimiento:\t" + fechaNacimiento +
            "\nDomicilio:\t" + domicilio + "\nSexo:\t" + sexo + "\n";
   }

   public void setDomicilio(String d){
      domicilio = d;
```

```
    }
    public String getNombre(){
      return nombre;
    }

    public String getFechaNacimiento() {
      return fechaNacimiento;
    }

    public String getDomicilio(){
      return domicilio;
    }

    public char getSexo(){
      return sexo;
    }
}


package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.15
 * Se define, de manera muy simple, el concepto Estudiante como una subclase de
 * la clase Persona. Un estudiante tiene atributos propios, además de todos los
 * heredados de la súper clase.
 */
public class Estudiante extends Persona{
   private static int serie = 1000;  // Atributo estático
   private int clave;
   private String carrera;

   public Estudiante() {
      super();
      serie++;
      clave = serie; // Asigna una clave numérica única a cada estudiante.
   }

   public Estudiante(String n, String fn, String d, char s){
      super(n, fn, d, s);
      serie++;
      clave = serie; // Asigna una clave numérica única a cada estudiante.
   }
```

```
public Estudiante(String n, String fn, String d, char s, String c){
   this(n, fn, d, s);
   carrera = c;
}

public String toString (){
   return super.toString () + "Clave:\t" + clave + "\nCarrera:\t" + carrera + "\n";
}

public int getClave(){
   return clave;
}

public String getCarrera(){
   return carrera;
}
```

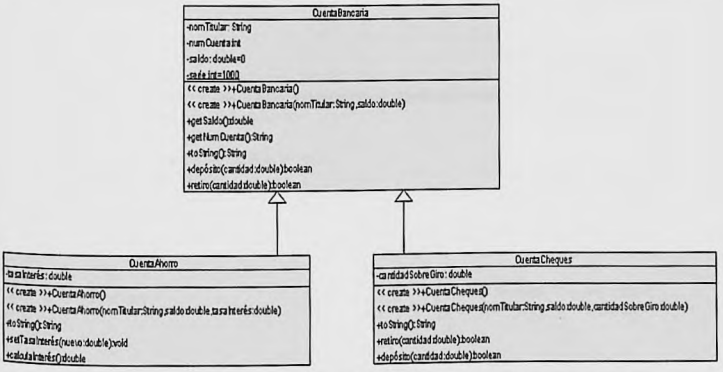


Figura 2.10 Ejemplo de herencia simple

En la figura 2.10 se presenta otro diagrama de clases con herencia simple. En este caso, de una clase base heredan dos clases derivadas. Se representa un esquema simplificado de cuentas bancarias, por medio de tres clases. En primer lugar, se definió una clase general *CuentaBancaria*, que tiene tres atributos, los constructores y algunos métodos para el manejo de las operaciones más importantes, por ejemplo, depósitos y retiros. Luego, la clase *CuentaAhorro* –subclase de la clase *CuentaBancaria*– para aquellas cuentas que reciben

dinero de clientes para ahorro y tienen como atributo el porcentaje de interés que pagan. Además, tiene algunos métodos propios. Por último, la clase *CuentaCheque* –subclase de la clase *CuentaBancaria*– para las cuentas bancarias que manejan cheques. Esta clase tiene como atributo propio la cantidad autorizada para sobregiros. El valor de ese atributo se calcula a partir del saldo –heredado de *CuentaBancaria*–. Las dos últimas clases heredan todos los miembros de la súper clase; sin embargo, en el caso de *CuentaCheque* se requiere sobrescribir los métodos *depósito()* y *retiro()* para que los mismos se adapten a las peculiaridades de esas operaciones en ese tipo de cuentas.

En este ejemplo se ve claramente que la herencia permite representar jerarquía de conceptos. En este caso, una *CuentaBancaria* es un concepto más general que una *CuentaAhorro*, que es un subtipo de la anterior. Asimismo, muestra cómo favorece el reuso de código, ya que en el caso de la *CuentaAhorro* ésta usa los métodos *depósito()* y *retiro()* heredados de la súper clase.

Los programas 2.16, 2.17 y 2.18 muestran el código correspondiente a la implementación de cada una de las tres clases del esquema anterior. Además, en el programa 2.19 se presenta una aplicación sencilla de las mismas.

Programa 2.16 | **CuentaBancaria.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.16
 * Definición de una clase que representa, de manera muy simplificada,
 * una cuenta bancaria.
 */

public class CuentaBancaria{
   private static int serie = 1000;
   private int numCta;
   private String nomTitular;
   private double saldo = 0.0;

   public CuentaBancaria(){
      numCta = ++serie;
   }

   public CuentaBancaria(String nom, double sal){
      this();
      nomTitular = nom;
      saldo = sal;
   }

   /* Se define como protegido para que sólo sus clases derivadas puedan
    * hacer uso de él.
```

```
*/
protected void setSaldo(double saldo) {
   this.saldo = saldo;
}

public int getNumCta() {
   return numCta;
}

public double getSaldo() {
   return saldo;
}

public String toString() {
   StringBuilder cad = new StringBuilder();

   cad.append("\n\nNúmero de cuenta: " + numCta + "\nNombre del titular: " + nomTitular);
   cad.append("\nSaldo: $" + saldo + "\n");
   return cad.toString();
}

/*
* Registra un depósito en la cuenta bancaria. Para ello valida que el monto a
* depositar sea mayor que 0. Regresa true si la operación se efectúa con
* éxito y false en caso contrario.
*/
public boolean depósito(double monto){
   boolean resp = false;

   if (monto > 0){
         saldo = saldo + monto;
         resp = true;
       }
   return resp;
}

/*
 * Registra un retiro de la cuenta bancaria. Si el monto es un valor mayor
 * o igual a cero y el saldo de la cuenta es mayor o igual que el monto a
 * retirar, actualiza el saldo y regresa true.
 * En caso contrario no altera el saldo y regresa false.
*/
public boolean retiro(double monto){
   boolean resp = false;
```

```
            if (monto > 0 && saldo >= monto){
               saldo = saldo - monto;
               resp = true;
            }
         return resp;
      }
   }
```

Programa 2.17 | **CuentaCheques.java**

```
package cap2;/**
 * @author Silvia Guardati
 * Programa 2.17
 * Ejemplo de herencia simple. Se define la clase CuentaCheques como una clase
 * derivada de la clase CuentaBancaria.
 */

public  class CuentaCheques extends CuentaBancaria{
   private  double canSobreGiro = 0.0;

   public  CuentaCheques( ){
      super( );
      canSobreGiro = getSaldo( ) * 0.1;
   }

   public  CuentaCheques(String  nom, double sal){
      super(nom, sal);
      canSobreGiro = getSaldo( ) * 0.1;
   }

   public  String  toString ( ){
      StringBuilder  cad = new StringBuilder( );

      cad.append(super.toString ( ));
      cad.append("\nCantidad sobre giro: " + canSobreGiro + "\n");
      return cad.toString( );
   }

   /*
   * Registra la operación de retiro. Para ello debe sobrescribir el método retiro de
   * la súper clase, ya que en el caso de una cuenta de cheques esta operación varía.
```

```
    * Un retiro se autoriza si el monto es menor o igual que el saldo, o bien, si el saldo más
    * la "cantidad autorizada para sobregiro" es mayor que el monto.
    */
    public  boolean retiro(double monto){
       boolean resp = false;

       if (super.retiro(monto)){
          canSobreGiro = getSaldo( ) * 0.1;
          resp = true;
       } else
          if ((getSaldo( ) + canSobreGiro) >= monto){
             setSaldo(0);
             canSobreGiro = 0;
             resp = true;
          }
       return resp;
    }

    /*
     * Registra la operación de depósito. Para ello debe sobrescribir el método depósito
     * de la súper clase, ya que en el caso de una cuenta de cheques esta operación varía.
     * Un depósito implica no sólo la actualización del saldo sino también la de la cantidad
     * autorizada para sobregiros.
     */
    public  boolean depósito(double monto){
       boolean resp = false;

            if (super.depósito(monto)){
               canSobreGiro = getSaldo( ) * 0.1;
               resp = true;
            }
       return resp;
    }
}
```

Programa 2.18 | **CuentaAhorro.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.18
```

```
/* Ejemplo de herencia simple. Se define la clase CuentaAhorro como una clase
 * derivada de la clase CuentaBancaria.
 */

public class CuentaAhorro extends CuentaBancaria{
   private double tasaInterés;

   public CuentaAhorro( ){
   }

   public CuentaAhorro(String nom, double sal, double inter){
      super(nom, sal);
      tasaInterés = inter;
   }

   public String toString ( ){
      StringBuilder cad = new StringBuilder( );

      cad.append(super.toString ( ));
      cad.append("\nInterés pagado: " + tasaInterés + "%\n");
      return cad.toString( );
   }

   public double getTasaInterés( ) {
      return tasaInterés;
   }

   public void setTasaInterés(double tasaInterés) {
      this.tasaInterés = tasaInterés;
   }

   /*
   * Calcula el interés a pagar sobre el saldo de la cuenta. Como el atributo
   * heredado es privado, debe usar el método getSaldo( ) para tener acceso a
   * su valor.
   */
   public double calculaInterés( ){
      return getSaldo( ) * tasaInterés / 100;
   }
}
```

Programa 2.19 | **UsaCuentas.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.19
 * Ejemplo de herencia simple.
 * Uso de las clases definidas en los programas 2.16, 2.17 y 2.18.
 */

public  class UsaCuentas{

   public  static void  main (String [] args) {
   double cant;

   /*
    * Se declara e instancia un objeto de cada uno de los tipos de cuentas definidas por
    * medio de las clases derivadas.
    */
   CuentaAhorro tuCuenAho = new CuentaAhorro("Isidoro Cañones", 5000,8.5);
   CuentaCheques suCuenChe = new CuentaCheques("Luna Martínez", 30000);

   /* Despliegue de la información de las cuentas bancarias. */
   System.out.println(tuCuenAho);
   System.out.println(suCuenChe);

   /* Calcula el interés a pagar en la cuenta de ahorro.
    * Se imprime el resultado obtenido por el método.
    */
   cant = tuCuenAho.calculaInterés( );
   System.out.println("\n\nA la cuenta: " + tuCuenAho + "Se le pagó: $" + cant +
                      " de interés\n");

   /* Se realiza -si el saldo lo permite- un retiro de la cuenta de ahorro y se muestra el
    * saldo actualizado. En caso de fracaso se despliega un mensaje.
    * Se invoca a métodos heredados de la clase CuentaBancaria: getNumCta( ) y getSaldo( ).
    */
   if (tuCuenAho.retiro(725.32))
         System.out.println("\n\nRetiro realizado en: " + tuCuenAho.getNumCta( ) +
                     "\nNuevo saldo: $" + tuCuenAho.getSaldo( ) + "\n" );
   else
         System.out.println("\n\nRetiro no efectuado en: " + tuCuenAho.getNumCta( ) +
                     " por saldo insuficiente: $" + tuCuenAho.getSaldo( ) + "\n" );
```

```
/* Se realiza un depósito en la cuenta de ahorro -si el monto es correcto- y se despliega el
 * nuevo saldo. En caso contrario se muestra un mensaje.
 */
if (suCuenChe.depósito(1000))
        System.out.println("\n\nDepósito realizado en: " + suCuenChe.getNumCta( ) +
                "\nNuevo saldo: $" + suCuenChe.getSaldo( ) + "\n" );
else
        System.out.println("\n\nDepósito no efectuado en: " + suCuenChe.getNumCta( )
                + " Error en el dato.\n");

/* Se realiza un retiro de la cuenta de cheque -si el saldo y la cantidad asignada para
 * sobregiro lo permite- y se muestra el saldo actualizado. Para ello se invoca al método
 * sobrescrito en la subclase. En caso de fracaso se despliega un mensaje.
 */
if (suCuenChe.retiro(4100))
        System.out.println("\n\nRetiro realizado en: " + suCuenChe.getNumCta( ) +
                "\nNuevo saldo: $" + suCuenChe.getSaldo( ) + "\n" );
else
        System.out.println("\n\nRetiro no efectuado en: " + suCuenChe.getNumCta( ) +
                " por saldo insuficiente: $" + suCuenChe.getSaldo( ) + "\n" );
  }
}
```

En el ejemplo presentado se aprecia la importancia que tiene poder sobrescribir métodos y de esta forma generalizarlos a distintas clases de objetos. La súper clase *CuentaBancaria* tiene los métodos *retiro()* y *depósito()*. Las clases derivadas los heredan, pero en el caso de la clase *CuentaCheques* se deben sobrescribir para ajustarlos a los requerimientos propios de dicha clase.

2.5.2 Herencia de múltiples niveles

La herencia de múltiples niveles hace referencia a un esquema de clases donde una clase derivada se utiliza como base para definir otra clase. Esta situación puede repetirse, provocando más niveles de herencia entre las clases involucradas. A medida que se definen clases a partir de otras clases se está representando información cada vez más específica, quedando en el nivel superior el concepto más general y en el inferior el más particular.

Retomando el ejemplo que se muestra en la figura 2.9, se presenta un diagrama de clases con herencia de múltiples niveles, en el cual se tienen tres clases relacionadas: *Animal*, *Mamífero* y *Felino*, las dos últimas describiendo conceptos que son casos específicos de su correspondiente súper clase. Observe el lector que se podría seguir definiendo otras subclases, por ejemplo, *FelinoSalvaje* y *FelinoDoméstico*, a partir de la clase *Felino*, para representar a aquellos que viven en estado salvaje –tigres, leones, linces, etc.– y a aquellos que conviven con el ser humano –gatos domésticos–, respectivamente.

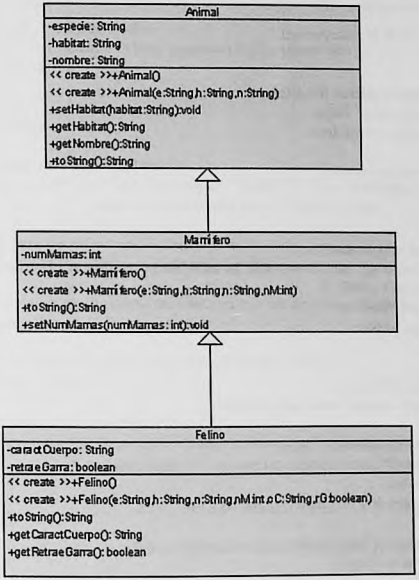


Figura 2.11 Herencia de múltiples niveles

En el programa 2.20 se presenta el código de la clase *Felino*, mientras que el código completo de las clases *Animal* y *Mamífero* se encuentra en los programas 2.11 y 2.12, respectivamente.

Programa 2.20 **Felino.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.20
 * Ejemplo de herencia de múltiples niveles.
 * Se define la clase Felino como derivada de la clase Mamífero, siendo ésta una
```

```
 * clase derivada de la clase Animal.
 */

public  class Felino extends Mamífero{
   private  String  caracCuerpo;
   private  boolean retraeGarras;

   // Constructor por omisión.
   public  Felino(){
   }

   // Constructor con parámetros.
   public  Felino(String  nom, String  hab, int nuM, String  carCue, boolean reGa){
     super(nom, hab, nuM);
     caracCuerpo = carCue;
     retraeGarras = reGa;
   }

   public  String  toString ( ){
     StringBuilder cadena = new StringBuilder( );

     cadena.append(super.toString( ));
     cadena.append("Caracteristicas del cuerpo:  " + caracCuerpo + "\n");
     if (retraeGarras)
        cadena.append("Este felino sí retrae las garras.  \n");
     else
        cadena.append("Este felino no retrae las garras.  \n");
     return cadena.toString( );
   }
}
```

En el programa 2.21 se presenta un ejemplo sencillo de herencia de múltiples niveles, haciendo uso de las clases definidas previamente.

Programa 2.21 | **UsaHerencia2.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.21
```

```
* Uso de herencia de múltiples niveles.
* Se define un objeto de la clase Felino –programa 2.20–, se imprime y
* se modifica dicho objeto.
*/

public  class UsaHerencia2 {

   public  static void  main (String [] args) {
      Felino tigre = new Felino("Tigre de Bengala", "India", 4, "Crece hasta los 3 metros de
                                 largo y puede alcanzar hasta 280 kilos", true);

      /* Imprime datos del objeto tipo Felino. */
      System.out.println ("\n\nDatos del felino\n\n" + tigre);

      /* Actualiza el número de mamas mediante un método de la clase Mamífero. */
      tigre.setMamas(6);

      /* Actualiza el hábitat mediante un método de la clase Animal. */
      tigre.setHábitat("Nepal");

      /* Imprime datos del objeto tipo Felino. */
      System.out.println ("\n\nDatos actualizados del felino\n\n" + tigre);
      }
}
```

Una vez ejecutada esta pequeña aplicación, en pantalla se verá desplegada la siguiente información:

```
Datos del felino

Nombre: Tigre de Bengala Habita en: India
Número de mamas: 4
Características del cuerpo: Crece hasta los 3 metros de largo y puede alcanzar hasta 280 kilos
Este felino sí retrae las garras.


Datos actualizados del felino

Nombre: Tigre de Bengala Habita en: Nepal
Número de mamas: 6

Características del cuerpo: Crece hasta los 3 metros de largo y puede alcanzar hasta 280 kilos
Este felino sí retrae las garras.
```

En los siguientes programas se presenta el código de otro ejemplo de herencia de múltiples niveles. Se define la clase Árbol y la clase *Frutal* a partir de ésta. Posteriormente se define la clase *Cítrico* como derivada de la clase *Frutal*, indicando una clase más específica de árboles frutales.

Programas 2.22, 2.23 y 2.24 | **Árbol.java** | **Frutal.java** | **Cítrico.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.22
 * Ejemplo de herencia de múltiples niveles.
 */

public class Árbol{
   private String nombre;
   private double altura;
   private String regiónClima;

   public Árbol(){
   }

   public Árbol(String nom, double alt, String rC){
      nombre = nom;
      altura = alt;
      regiónClima = rC;
   }

   public String toString (){
      return "Datos del Árbol: \n" + "Nombre:\t" + nombre + "\nAltura:\t" + altura + "\nRegión-
     Clima: " + regiónClima + "\n";
   }
}

package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.23
 * Ejemplo de herencia de múltiples niveles.
 */

public class Frutal extends Árbol{
   private String tiempoCosecha;
```

```
    private String nombreFruto;
    private String color;

    public Frutal(String nom, double alt, String rC, String tC, String nF, String col) {
      super(nom, alt, rC);
      tiempoCosecha = tC;
      nombreFruto = nF;
      color = col;
    }

    public String toString (){
      return super.toString ( ) + "Datos del fruto: \nNombre: " + nombreFruto + "\nColor: "
                                + color + "\nSe cosecha en: " + tiempoCosecha + "\n";
    }
}

package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.24
 * Ejemplo de herencia de múltiples niveles.
 * La clase Cítrico no tiene atributos, se define para contar con un tipo más específico
 * de frutas.
 */

public class Cítrico extends Frutal{

    public Cítrico(String nom, double alt, String rC, String tC, String nF, String col){
      super(nom, alt, rC, tC, nF, col);
    }

    public String toString( ) {
      return super.toString( ) + "Pertenece a la categoría de frutas cítricas\n";
    }
}
```

Como se puede observar en el programa 2.24, en la clase *Cítrico* no se incluyeron atributos. La clase se deriva a partir de la clase que representa a los árboles frutales, con el objeto de tener una categoría de fruta más específica: los cítricos, sin tener la necesidad de agregar nuevos atributos. También se podría definir a partir de la clase *Frutal* otras subclases para representar a las frutas del bosque (aquellas que crecen en arbustos, como la frambuesa) o a las frutas de grano (aquellas que tienen numerosas semillas, como la granada). A su vez, la clase *Árbol* se pudo usar para derivar otras clases; por ejemplo, *Forestal* (que representa a los árboles que tienen valor comercial por su madera) y *Ornamental* (que representa a los árboles usados para decoración de jardines y parques).

En el programa 2.25 se presenta un ejemplo de uso de las clases previamente definidas. Se crean tres objetos, uno de tipo *Árbol*, otro de tipo *Frutal* y el último de tipo *Cítrico*. Los dos últimos tienen los mismos atributos porque, como ya se mencionó, los árboles de cítricos no tienen ninguna característica adicional con respecto a los árboles frutales en general, según nuestra representación.

Programa 2.25 | **UsaHerencia3.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.25
 * Ejemplo de uso de herencia de múltiples niveles, basado en los programas 2.22, 2.23 y 2.24
 */
public class UsaHerencia3 {

   public static void main(String[] args) {

      Árbol unCedro = new Árbol("cedro", 11.8, "Ecuador");
      Frutal unManzano = new Frutal("manzano", 10.6, "Río Negro, Argentina", "enero-febrero",
      "manzana", "roja");
      Cítrico unLimonero = new Cítrico("limonero", 4.3, "México", "todo el año", "limón",
      "amarillo");

      System.out.println(unCedro);
      System.out.println(unManzano);
      System.out.println(unLimonero);
   }
}
```

Al ejecutarse el programa 2.25 se obtienen los siguientes resultados:

```
Datos del Árbol:
Nombre: cedro
Altura:   11.8
Región-Clima: Ecuador

Datos del Árbol:
Nombre: manzano
Altura:   10.6
Región-Clima: Río Negro, Argentina
Datos del fruto:
Nombre: manzana
Color: roja
Se cosecha en: enero-febrero
```

Datos del Árbol:
Nombre: limonero
Altura: 4.3
Región-Clima: México
Datos del fruto:
Nombre: limón
Color: amarillo
Se cosecha en: todo el año
Pertenece a la categoría de frutas cítricas

2.5.3 Herencia múltiple

La herencia múltiple se da cuando una clase hereda de más de una súper clase. Por lo tanto, estará compartiendo los miembros de varias clases. Este tipo de herencia es útil para representar situaciones que requieren que un cierto elemento se defina a partir de otros dos o más. Por ejemplo, la clase *AlimentosPreparados* podría heredar de las clases *AlimentosVegetales* y *AlimentosAnimales*.

En Java no es posible representar este tipo de relaciones en forma directa. Se puede simular parcialmente por medio de interfaces. Sin embargo, es importante destacar que no es exactamente lo mismo, porque como ya se vio éstas sólo indican comportamiento, mientras que las clases definen características y comportamiento. Por tanto, cuando una clase implementa a más de una interface se está especificando que la misma comparte únicamente comportamiento con todas ellas (dándole su propia forma al implementar los métodos).

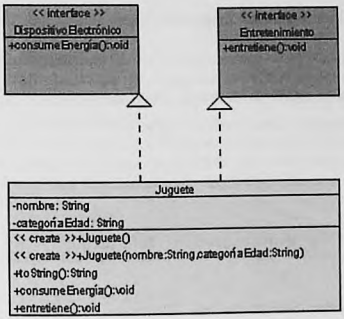


Figura 2.12 Simulación de herencia múltiple

En la figura 2.12 se presenta un esquema de dos interfaces implementadas por una clase. En este caso, la clase *Juguete* debe implementar las operaciones indicadas en las mismas y, de esta manera, simula una especie de herencia múltiple. Un objeto de la clase *Juguete* definida de esta manera es un dispositivo

electrónico y, además, un medio de entretenimiento. Pero al ser interfaces sólo se puede indicar el comportamiento común esperado, por lo que la relación no se puede considerar de herencia, de igual manera que en los casos anteriores.

Muy importante

- La palabra reservada ***extends*** indica relación de herencia entre clases.
- La primera línea del constructor de la clase derivada debe ser la llamada al constructor de la súper clase.
- Si en la clase derivada se incluye un constructor por omisión, Java invoca al constructor por omisión de la clase base. En caso de no tenerlo, se genera un error.
- El constructor de la *súper* clase se invoca por medio de la palabra ***super*** y los parámetros, en caso de llamar al constructor con parámetros.
- La palabra reservada ***super*** hace referencia a atributos y/o métodos de la clase base.
- Las clases derivadas tienen acceso (pueden usar, modificar, etc.) a los miembros protegidos o públicos de la súper clase.
- Si una clase hereda de una súper clase e implementa una interface, entonces en el encabezado de la clase primero se pone el ***extends*** y luego el ***implements***.
- Se puede definir relación de herencia entre interfaces.

2.5.4 Uso del modificador final

El modificador final se puede usar en las clases asociado a métodos o a la misma clase. Al declarar un método como final se impide que el mismo pueda ser sobrescrito en alguna subclase. Por tanto, es muy útil cuando se quiere forzar a que todas las subclases tengan el mismo comportamiento heredado.

En el programa 2.26 se presenta la clase *Vehículo* que tiene un método "final" que no podrá sobrescribir ninguna de sus clases derivadas. El programa 2.27 muestra una clase derivada. Si en ella se intenta sobrescribir el método se genera un error. A partir de la clase *Vehículo* se podrán definir otras, por ejemplo, *Camión* o *Lancha*, y en todas se tendrá la misma versión del método declarado como "final".

Programas 2.26 y 2.27 | **Vehículo.java** | **Auto.java**

```
package cap2;
import java.util.Calendar; // Importa la clase Calendar de Java

/**
 * @author Silvia Guardati
 * Programa 2.26
 * Ejemplo de método "final".
 */
```

```
public class Vehículo {
   private String numSerie, numMotor;
   private double precioFactura;
   private int año;

   public Vehículo( ) {
   }

   public Vehículo(String numSerie, String numMotor, double precioFactura, int año) {
      this.numSerie = numSerie;
      this.numMotor = numMotor;
      this.precioFactura = precioFactura;
      this.año = año;
   }

   @Override
   public String toString( ) {
      return "NumSerie= " + numSerie + "\nNumMotor= " + numMotor + "\nAñoo= " + año;
   }

   /* El método calcula el impuesto anual a pagar por un vehículo. Éste se determina
    * por el precio de compra (factura) y de acuerdo a su antigüedad se aplica un
    * descuento.
    * El método es "final", por lo tanto ninguna subclase podrá sobrescribirlo.
    */
   public final int calcImpuestoAnual( ){
      double impuesto;
      // La clase Calendar se usa como auxiliar para indentificar el año actual.
      Calendar fecha = Calendar.getInstance( );
      int añoActual, totalAños;

      añoActual = fecha.get(fecha.YEAR); // Se obtiene el año actual.
      totalAños = añoActual - año;
      impuesto = precioFactura * 0.035;
      if (totalAños >= 5 && totalAños < 10)
         impuesto = impuesto * 0.97;
      else
         if (totalAños >= 10 && totalAños <= 15)
            impuesto = impuesto * 0.94;
         else
            if (totalAños > 15)
               impuesto = impuesto * 0.90;
      return (int) impuesto;
   }
}
```

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.27
 * Esta clase hereda el método calcImpuestoAnual( ) de la clase Vehiculo. No lo puede
 * sobrescribir por ser "final".
 */
public class Auto extends Vehiculo{
   private String color, marca, modelo;
   private int numPuertas;

   public Auto( ) {
   }

   public Auto(String color, String marca, String modelo, int numPuertas, String numSerie,
   String numMotor, double precioFactura, int año) {
      super(numSerie, numMotor, precioFactura, año);
      this.color = color;
      this.marca = marca;
      this.modelo = modelo;
      this.numPuertas = numPuertas;
   }

   @Override
   public String toString( ) {
      return super.toString( ) + "\nColor= " + color + "\nMarca= " + marca + "\nModelo= " +
      modelo + "\nNúmero de puertas= " + numPuertas;
   }
}
```

Si a una clase se la declara como final, se está indicando que la misma no puede ser derivada. Esta característica es útil cuando se definen clases que queremos que permanezcan inmutables. Ejemplos de este tipo de clases en Java son: *String*, *Integer* y *Double*, entre otras.

Analice el programa 2.28. Se definió la clase *Punto*, de la cual se derivó la clase *PuntoConColor*, programa 2.29. Esta última se declaró como final. Por tanto, si se intentara derivar otra clase a partir de ella, se generaría un error. En este caso, se está especificando que no puede haber una clase más específica de puntos.

Programas 2.28 y 2.29 | **Punto.java** | **PuntoConColor.java**

```
package cap2;

/**
 * @author Silvia Guardati
 * Programa 2.28
 * Clase que define un punto, teniendo sus coordenadas como atributos.
 */
public class Punto {
   private double x, y;

   public Punto() {
   }

   public Punto(double x, double y) {
      this.x = x;
      this.y = y;
   }

   @Override
   public String toString() {
      return "Punto: (" + "x= " + x + ", y= " + y + ')';
   }

   public double getX() {
      return x;
   }

   public double getY() {
      return y;
   }

   public void setX(double x) {
      this.x = x;
   }

   public void setY(double y) {
      this.y = y;
   }
}
```

```
package cap2;

/**
```

```
 * @author Silvia Guardati
 * Programa 2.29
 * Ejemplo de clase "final": no podrán derivarse otras clases a partir de ésta.
 */
public final class PuntoConColor extends Punto{
   private String color;

   public PuntoConColor( ) {
   }

   public PuntoConColor(String color, double x, double y) {
      super(x, y);
      this.color = color;
   }

   @Override
   public String toString( ) {
      return super.toString( ) + "- Color= " + color;
   }

   public String getColor( ) {
      return color;
   }

   public void setColor(String color) {
      this.color = color;
   }
}
```

Otro ejemplo de uso del modificador "final" junto a una clase es el siguiente: en un esquema de jerarquía de clases se definen las figuras geométricas como súper clase o interface. Posteriormente, la clase *Triángulo*, que extiende o implementa a la anterior. A partir de ésta se puede definir la clase *Equilátero* como una clase "final", si se quiere impedir clases de triángulos más específicos que los equiláteros.

2.6 RESUMEN

En este capítulo se presentaron las características fundamentales del paradigma de programación llamado Programación Orientada a Objetos (POO). Se explicaron los conceptos de clase y objetos, así como el de miembros: atributos y métodos.

Asimismo, se estudiaron conceptos avanzados de la POO, como las interfaces y los diversos tipos de herencia, y su uso para modelar la solución de un problema.

Todos los temas se complementaron con ejemplos para ayudar al lector a la comprensión de los mismos.

2.7 EJERCICIOS

2.1 Analice la definición de la clase *Perro* que se da a continuación y diga si los enunciados que aparecen más abajo son verdaderos o falsos.

```
public class Perro{
  private int peso;
  private String nombre;

  public int getPeso(){
    return peso;
  }
  public void setPeso(int nuevoPeso){
    peso = nuevoPeso;
  }
}
```

a. La definición de la clase está correcta.

b. Se requiere incluir, al menos, un constructor para que la clase pueda usarse.

c. El método *getPeso()* podría omitirse sin que afectara el acceso al atributo peso.

d. El atributo *peso* podría modificarse directamente desde alguna clase usuaria, sin necesidad del método *setPeso()*.

2.2 Analice la definición de la clase *Fecha* que se da a continuación y diga si los enunciados que aparecen más abajo son verdaderos o falsos.

```
public class Fecha{
  private int día = 1;
  private int mes = 1;
  private int año = 2013;

  public Fecha(){
  }

  public Fecha(int d, int m, int a){
    día = d;
    mes = m;
    año = a;
  }

  public Fecha(Fecha miFecha){
    día = miFecha.día;
```

```
        mes = miFecha.mes;
        año = miFecha.año;
    }
}
```

a. La definición de la clase está correcta.

b. Es incorrecto tener un constructor por omisión y, además, asignar valores por omisión a los atributos de la clase.

c. Es válido usar un objeto de la clase que se está definiendo como parámetro de uno de los constructores.

d. En el último constructor no se puede hacer referencia a los atributos del objeto *miFecha* porque son privados.

2.3 Retome el problema anterior e incluya en la clase el método *toString ()*, sobrescrito de tal manera que permita desplegar en pantalla los valores de los atributos.

2.4 Retome el problema anterior e incluya en la clase el método *compareTo()* sobrecargado de tal manera que permita comparar dos objetos de la clase *Fecha* y regrese un valor negativo, positivo o cero, dependiendo de si la primera fecha es menor, mayor o igual a la segunda.

2.5 Retome la clase del programa 2.2 y complétela incluyendo los métodos *toString()*, *equals()* y *compareTo()*. Sobrescríbalos o sobrecárguelos, según corresponda.

2.6 Asumiendo que ya resolvió los problemas del 2 al 4, analice las siguientes líneas de código, pertenecientes a una aplicación, y diga si los enunciados que se dan más abajo son verdaderos o no.

```
Fecha miCumple = new Fecha(11, 7, 2013);
Fecha unaFecha = new Fecha();
Fecha fechaFiesta = new Fecha(miCumple);

System.out.println(miCumple);
System.out.println(unaFecha);
System.out.println(fechaFiesta);

if (miCumple.compareTo(fechaFiesta) == 0)
   System.out.println("\n\nLa fiesta será el día de mi cumpleaños.\n");
else
   System.out.println("\n\nLa fiesta será otro día.\n");

unaFecha.día = 10;
```

a. La declaración e instanciación del objeto *miCumple* es correcta.

b. La declaración e instanciación del objeto *unaFecha* es incorrecta.

c. Cuando se imprime el objeto *unaFecha* los valores de los atributos *día*, *mes* y *año* están indefinidos.

d. Como consecuencia del uso del método *compareTo()* se imprime el mensaje: **La fiesta será el día de mi cumpleaños.**

e. La asignación de un nuevo valor al atributo *día*, del objeto *unaFecha*, es incorrecta.

2.7 Retome el programa 2.8 e incluya un método *equals()*, de tal manera que permita comprobar si dos tortugas son iguales (deben tener el mismo dueño, llamarse igual y ser de la misma especie).

2.8 Defina la clase *Persona*. Determine los atributos y el conjunto de métodos –lo más completo posible– que caracterizan al concepto persona.

- Realice el diagrama de clase correspondiente con UML.
- Escriba la clase *Persona* en Java.
- Escriba un programa en Java que utilice la clase previamente definida para declarar e instanciar el objeto *miAmigo*. Además, el programa debe poder, por medio de los métodos incluidos en la clase, realizar las siguientes operaciones:

a. Imprimir el número de teléfono de *miAmigo*.

b. Imprimir todos los datos de *miAmigo*.

c. Imprimir el nombre del deporte que practica *miAmigo*.

d. Cambiar la dirección de *miAmigo*. El usuario dará la nueva dirección.

a. Cambiar el número de teléfono de *miAmigo*. El usuario dará el nuevo número.

2.9 En el programa 2.11 agregue el método equals(), de tal manera que se puedan comparar dos animales y ver si son iguales (tienen el mismo nombre y hábitat). Luego, en la clase *Mamífero* haga lo necesario para que también cuente con un método *equals* para determinar si dos mamíferos son iguales (tienen el mismo nombre y hábitat y el mismo número de mamas).

2.10 Retome los programas 2.16, 2.17 y 2.18. Haga los ajustes necesarios para que las cuentas puedan imprimirse de manera amigable y compararse, tanto para saber si son sólo iguales como para saber si una cuenta es menor, igual o mayor que otra, según su número. La solución debe ser general.

2.11 Defina la clase *Círculo*. Determine los atributos y el conjunto de métodos –lo más completo posible– que caracterizan al concepto círculo.

- Realice el diagrama de clase correspondiente con UML.
- Escriba la clase *Círculo*, en Java.

- Escriba un programa para que, dados los radios correspondientes a manteles circulares, calcule el total de metros cuadrados de tela que se necesitan para su confección. Utilice la clase definida para declarar e instanciar objetos que representan a los manteles.

Datos:

numMant ($1 \leq$ numMant ≤ 100)
radio1
radio2
...
radionumMant

Donde:

- numMant es un valor numérico entero, positivo, que indica el total de manteles que se desea confeccionar.
- *radioi* es un valor numérico entero, positivo, que indica el tamaño del radio en centímetros, del mantel i.

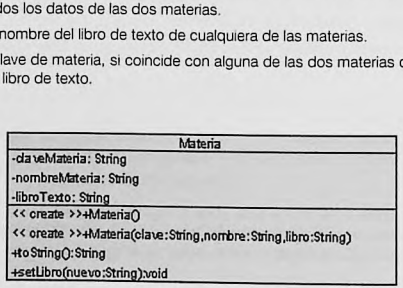
Resultado esperado: total de metros cuadrados de tela requeridos para la confección de los manteles.

2.12 Defina la clase *Materia*, según el diagrama que se muestra más abajo. Posteriormente, desarrolle un programa de aplicación que declare e instancie los objetos *estructurasDeDatos* y *cálculoNumérico* usando la clase previamente definida. El programa debe permitir al usuario, por medio de menús, realizar las operaciones que se listan a continuación. Si requiere métodos adicionales, inclúyalos.

a. Imprimir todos los datos de las dos materias.

b. Cambiar el nombre del libro de texto de cualquiera de las materias.

c. Dada una clave de materia, si coincide con alguna de las dos materias que se tienen, imprimir el nombre del libro de texto.

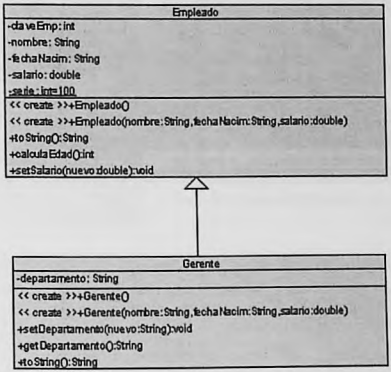


2.13 Retome la clase definida en el ejercicio anterior. ¿Qué método(s) debería agregarle/quitarle para que se pudiera imprimir, desde un programa de aplicación, sólo el nombre de la materia?

2.14 Considere el siguiente diagrama de clases con herencia. Ambas clases tienen dos constructores, uno por omisión y el otro con parámetros, y el método *toString ()* sobrescrito. La clase *Empleado* tiene el método *setSalario(double)* que recibe como parámetro un nuevo salario con el cual actualiza el salario del empleado. La clase *Gerente* tiene el método *setDepar(String)* que recibe como parámetro una cadena, que representa el nombre de un departamento, y actualiza con dicho valor el nombre del departamento que tiene a su cargo.

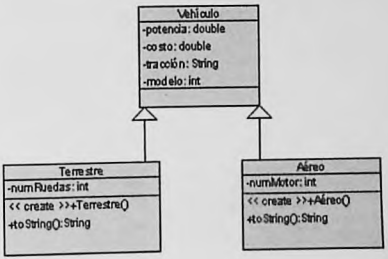
a. Defina las dos clases que aparecen en el diagrama, respetando las especificaciones dadas.

b. Escriba un programa de aplicación que pueda: 1) crear un objeto de tipo *Empleado*, 2) imprimir todos sus datos, 3) actualizar su salario, 4) crear un objeto de tipo *Gerente*, 5) imprimir todos sus datos, 6) actualizar su salario y 7) cambiar el nombre del departamento que tiene a su cargo.

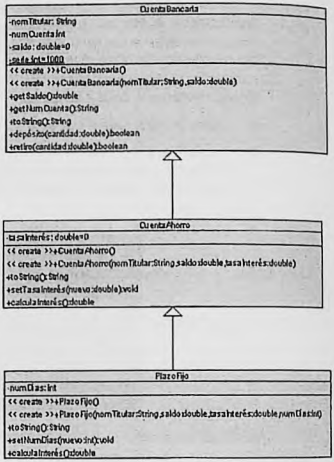


2.15 Considere el siguiente diagrama de herencia simple. En las tres clases se indican algunos atributos y métodos.

a. Defina las clases que aparecen en el diagrama, completando atributos y métodos que crea conveniente para representar de la manera más completa los conceptos: *vehículo en general*, *vehículo terrestre* y *vehículo aéreo*.



b. Escriba un programa de aplicación que declare e instancie objetos usando las clases definidas en el inciso anterior. Utilice dichos objetos para invocar a los métodos incluidos en las clases.



2.16 Considere el siguiente diagrama de herencia de múltiples niveles. Las tres clases tienen dos constructores, uno por omisión y el otro con parámetros, y el método *toString()* sobrescrito. Las dos primeras clases corresponden a los programas 2.16 y 2.18, respectivamente. A la tercera debe definirla completamente, incluyendo la sobrescritura del método *calculaInterés()*, que deberá ajustarse a las características propias de los plazos fijos.

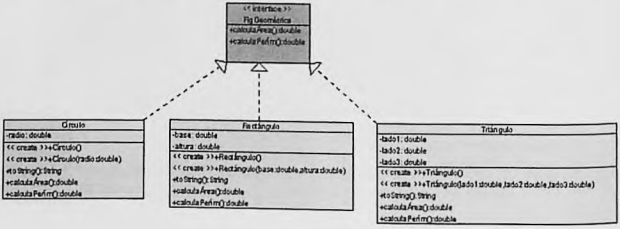
a. Defina las tres clases que aparecen en el diagrama, respetando las especificaciones dadas.

b. Escriba un programa de aplicación que pueda: 1) crear objetos del tipo de cada una de las clases, 2) imprimir todos sus datos, 3) usar los métodos de cada clase –a partir de objetos de la clase base o de objetos de clases derivadas–.

2.17 Analice el siguiente diagrama en el cual se definió una interface: *FiguraGeométrica* y tres clases que la implementan: *Rectángulo*, *Círculo* y *Triángulo*. Estas tres clases deberán implementar los métodos de la interface. Con esta relación se establece que todas las figuras geométricas (sin importar su forma) deben poder calcular su área y su perímetro.

a. Defina todas las clases y la interface que aparecen en el diagrama. Complete con los atributos y métodos que crea conveniente.

b. Escriba un programa de aplicación que pueda: 1) crear objetos del tipo de cada una de las clases, 2) imprimir todos sus datos, 3) usar los métodos de cada clase–.



CLASES ABSTRACTAS, POLIMORFISMO Y CLASES GENÉRICAS

Contenido

Competencias

- Explicar las clases abstractas y señalar su diferencia con las interfaces.
- Explicar el polimorfismo y su uso en la solución de problemas.
- Explicar las clases genéricas, sus características y su uso en la solución de problemas.
- Presentar las pruebas unitarias como una herramienta fundamental en el aseguramiento de la calidad del software.

3.1 INTRODUCCIÓN

En el capítulo anterior se presentaron algunos de los principios más importantes de la POO. En éste se estudian otros principios que ayudan a diseñar e implementar productos de software más generales y, por lo tanto, más flexibles.

Las clases abstractas son usadas para modelar conceptos que no se van a usar para construir objetos, sino que sirven como base para definir otros conceptos, favoreciendo así el reuso de código y la jerarquización de conceptos. Por su parte, el polimorfismo permite que una variable pueda hacer referencia a objetos de distintos tipos, logrando mucha flexibilidad en el manejo de las variables. Por último, las clases genéricas constituyen una especie de plantillas de clases en las cuales al menos un atributo queda sin especificar su tipo. De esta manera, en el momento de ejecutar el programa se asigna dinámicamente un tipo a dicho atributo. Las clases genéricas ofrecen un gran potencial a los desarrolladores de software, tal como se verá en este capítulo y en los subsiguientes.

3.2 CLASES ABSTRACTAS

Las *clases abstractas* definen un concepto general del cual no se requieren instancias para trabajar directamente. Es decir, permiten establecer una jerarquía entre conceptos y favorecen el reuso pero no pueden crearse objetos a partir de ellas. Por lo tanto, una clase abstracta se utiliza para definir otras clases que representan conceptos concretos o más específicos, las cuales sí serán usadas para crear objetos. Este tipo de clases son útiles para el diseño de soluciones flexibles y generales.

NO se pueden crear referencias a objetos de una clase abstracta.

Una clase abstracta puede tener atributos y métodos totalmente definidos y también métodos abstractos. Un *método abstracto* es aquel que no está implementado y por lo tanto su implementación debe hacerse en las clases derivadas de la clase abstracta. De esta manera, en la clase más general se incluyen aquellas características y comportamiento que son comunes y se dejan señalados aquellos elementos que estarán presentes en las clases derivadas pero que tendrán distinta forma. Es otra aplicación de la sobrescritura y/o sobrecarga de métodos. Es importante señalar que si una clase tiene un método abstracto, entonces la clase necesariamente debe ser declarada como abstracta.

En Java, una clase abstracta se declara usando el modificador **abstract** junto a la palabra **class** en el encabezado de la misma. Por su parte, los métodos abstractos se indican por medio del modificador **abstract**, escribiendo sólo la firma del mismo. En UML se señala que una clase es abstracta usando letras itálicas. Tomando como base que no se pueden crear instancias de una clase abstracta, se considera una buena práctica declarar el constructor de la misma como **protected** en lugar de **public**.

En la figura 3.1 se presenta un esquema de una clase abstracta, *Insecto*, a partir de la cual se definen dos clases derivadas concretas. En la clase abstracta hay un método abstracto, *come()*, el cual se implementa en cada una de las clases derivadas de acuerdo con las características propias del insecto que representa. Los constructores se declararon como **protected**. Observe que a los atributos se les dio un valor por omisión en la declaración de los mismos, como una variante de la asignación en el constructor por omisión.

Muy importante

- Una clase abstracta puede tener todos sus métodos definidos. En este caso, se declara abstracta para impedir la creación de objetos a partir de ella.
- Una clase abstracta puede tener uno o más métodos abstractos.
- Si la clase tiene por lo menos un método abstracto, entonces debe ser declarada como abstracta.
- Los métodos abstractos deben definirse en las clases derivadas de la clase abstracta (o en las derivadas de éstas si existe herencia de múltiples niveles).

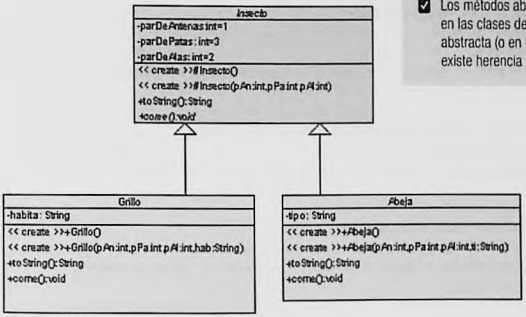


Figura 3.1 Ejemplo de clase abstracta

A continuación se presenta el código de las clases del diagrama anterior. Observe que en la clase *Insecto*, el método abstracto *come()* sólo tiene su firma, mientras que en las clases *Grillo* y *Abeja* aparece implementado. Asimismo, en la clase *Insecto* se dieron valores por omisión a los atributos, considerando que todos los insectos se caracterizan por tener 1 par de antenas, 3 pares de patas y 2 pares de alas. Sin embargo, se incluyó un constructor con todos los atributos para permitir crear un insecto con otros valores, si fuera necesario. Por ejemplo, en el caso de que un insecto en particular por alguna razón tuviera o se hubiera quedado con sólo 2 pares de patas.

Programa 3.1, 3.2 y 3.3 | **Insecto.java** | **Grillo.java** | **Abeja.java**

```
/**
 * @author Silvia Guardati
 * Programa 3.1
 * Clase abstracta con un método abstracto -come( ).
 */
public abstract class Insecto {
   private int parDeAntenas = 1;
   private int parDePatas = 3;
   private int parDeAlas = 2;

   public Insecto( ) {
   }

   public Insecto(int pAn, int pPa, int pAl) {
      parDeAntenas = pAn;
      parDePatas = pPa;
      parDeAlas = pAl;
   }

   // Método abstracto: se debe implementar en las clases derivadas.
   public abstract void come( );

   @Override
   public String toString( ) {
      StringBuilder cad = new StringBuilder( );
      cad.append("tiene " + parDeAntenas + " par de antenas, ");
      cad.append(parDePatas + " pares de patas y " + parDeAlas + " pares de alas\n");
      return cad.toString( );
   }
}
```

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.2
 * La clase Grillo, como derivada de la clase abstracta Insecto, debe implementar
 * el método come( ).
 */
public class Grillo extends Insecto{
   private String habita;
```

```
    public Grillo( ) {
    }

    /* Este constructor permite crear un objeto tipo Grillo tomando los valores
     * por omisión para los atributos heredados de Insecto.
     */
    public Grillo(String habita) {
        super( );
        this.habita = habita;
    }

    public Grillo(int pAn, int pPa, int pAl, String habita) {
        super(pAn, pPa, pAl);
        this.habita = habita;
    }

    @Override
    public String toString( ) {
        return "\nEl grillo " + super.toString( ) + "Habita en " + habita;
    }

    @Override
    public void come( ) {
        System.out.println("\nLos grillos son omnívoros, comen hojas, tallos e insectos.");
    }
}

package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.3
 * La clase Abeja, como derivada de la clase abstracta Insecto, debe implementar
 * el método come( ).
 */
public class Abeja extends Insecto {
    private String tipo;

    public Abeja( ) {
    }

    /* Este constructor permite crear un objeto tipo Abeja tomando los valores
     * por omisión para los atributos heredados de Insecto.
     */
    public Abeja(String tipo) {
```

```
        super( );
        this.tipo = tipo;
    }

    public Abeja(int pAn, int pPa, int pAl, String tipo) {
        super(pAn, pPa, pAl);
        this.tipo = tipo;
    }

    @Override
    public String toString( ) {
        return "\nLa abeja " + super.toString( ) + "Esta abeja es de tipo: " + tipo;
    }

    @Override
    public void come( ) {
        System.out.println("\nLas abejas comen polen y néctar.");
    }
}
```

En el programa 3.4 se presenta una aplicación simple para mostrar cómo se pueden usar las clases definidas a partir de clases abstractas. Como puede observarse, el manejo de estas clases es el mismo que el de clases derivadas a partir de clases concretas.

Programa 3.4

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.4
 * Ejemplo de uso de clases derivadas a partir de una clase abstracta, así como
 * del uso de una clase abstracta para referenciar objetos de clases concretas.
 * Observe que la variable "unInsecto", de tipo Insecto, almacena la referencia
 * a una clase concreta: Abeja. En ningún caso se instancian objetos tipo Insecto.
 */
public class UsaClaseAbstracta {

    public static void main(String[ ] args) {

        /* Se instancian objetos tipo Grillo y tipo Abeja usando las clases
         * respectivas. En ambos casos se asignan los valores por omisión a los
         * atributos correspondientes a Insecto.
         */
```

```
        Grillo unGrillo = new Grillo("campo y casas");
        Abeja unaAbeja = new Abeja("obrera");

        /* Se instancia un objeto tipo Grillo asignando valores a los
         * atributos correspondientes a Insecto. Por alguna razón, este grillo
         * sólo tiene 2 pares de patas.
         */
        Grillo otroGrillo = new Grillo(1, 2, 2, "árboles");

        /* Se instancia un objeto tipo Abeja y se almacena su referencia en una
         * variable del tipo de la clase abstracta (Insecto).
         */
        Insecto unInsecto = new Abeja("reina");

        // Se imprimen todas las variables declaradas.
        System.out.println(unGrillo);
        System.out.println(unaAbeja);
        System.out.println(otroGrillo);
        System.out.println(unInsecto);

        unGrillo.come( ); // Se ejecuta el come( ) de la clase Grillo.
        unaAbeja.come( ); // Se ejecuta el come( ) de la clase Abeja.
        unInsecto.come( ); // Se ejecuta el come( ) de la clase Abeja.
    }
}
```

En los programas 3.5, 3.6 y 3.7 se presenta el código de la clase abstracta *Empleado* y de dos clases derivadas: *Ingeniero* y *Administrador*, respectivamente. La primera se utiliza para representar atributos y comportamiento de cualquier empleado, sin importar su categoría dentro de una organización. Se incluye un método abstracto, *calculaPago()*, el cual se implementará de manera diferente en cada una de las clases derivadas. En este caso, representa una acción que depende de la categoría del empleado y que, por lo tanto, no se puede generalizar.

Programa 3.5, 3.6 y 3.7	Empleado.java	Ingeniero.java	Administrador.java

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.5
```

```
 * Ejemplo de clase abstracta.
 */
public abstract class Empleado{
  private String nombre;
  private String fechaNacimiento;
  private String domicilio;

  public Empleado( ) {
  }

  protected Empleado(String nom, String fNac, String dom){
     nombre = nom;
     fechaNacimiento = fNac;
     domicilio = dom;
  }

  public String toString ( ){
     return "Nombre:\t" + nombre + "\nFecha Nacimiento: " + fechaNacimiento +
            "\nDomicilio: " + domicilio;
  }

  public void setDomicilio(String dom){
     domicilio = dom;
  }

  public String getNombre( ){
     return nombre;
  }

  public String getFechaNacimiento( ){
     return fechaNacimiento;
  }

  public String getDomicilio( ){
     return domicilio;
  }

  /* Método abstracto, se indica sólo la firma. Se deberá implementar en las clases concretas
   * definidas a partir de ésta.
   */
  public abstract double calculaPago(double monto);
}

package cap3;
```

```
/**
 * @author Silvia Guardati
 * Programa 3.6
 * Clase concreta que hereda de la clase Empleado.
 */
public class Ingeniero extends Empleado{
   private int numProyectos;

   public Ingeniero() {
   }

   public Ingeniero(String n, String fn, String d, int np){
      super(n, fn, d);
      numProyectos = np;
   }

   /* Método heredado de Empleado y que debe implementar por ser un método
    * abstracto. Se debe respetar la firma del método, según su declaración en
    * la clase base.
    */
   public double calculaPago(double monto){
      return numProyectos * monto;
   }

   public void setNumProyectos(int numProyectos) {
      this.numProyectos = numProyectos;
   }

   public String toString (){
      return "Ingeniero\n" + super.toString () + "\nNúmero de Proyectos:\t" +
                  numProyectos + "\n";
   }
}

package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.7
 * Clase concreta que hereda de la clase Empleado.
 */
public class Administrador extends Empleado{
   private String departamento;
```

```
    public Administrador( ) {
    }

    public Administrador(String n, String fn, String d, String dep){
      super(n, fn, d);
    departamento = dep;
    }

    /* Método heredado de Empleado y que debe implementar por ser un método
     * abstracto. Se debe respetar la firma del método, según su declaración en
     * la clase base.
     */
    public double calculaPago(double monto){
      double pago;
      if (departamento.equals ("Ventas"))
        pago = monto * 1.5;
      else
        if (departamento.equals ("Recursos Humanos"))
          pago = monto * 1.25;
        else
          if (departamento.equals ("Informática"))
            pago = monto * 1.6;
          else
            pago = monto * 1.1;
      return pago;
    }

    public void setDepartamento(String dep){
      departamento = dep;
    }

    public String toString (){
      return "Administrador\n" + super.toString ( ) + "\nDepartamento:\t" +
              departamento + "\n";
    }
}
```

3.3 POLIMORFISMO

El *polimorfismo* hace referencia a la capacidad de una variable de tener distintas formas. Es importante señalar que cada objeto tiene una única forma, siendo la variable de referencia la que puede tomar diferentes formas al apuntar a distintos objetos. Esta característica se logra por medio de herencia entre clases (concretas o entre clases abstractas y concretas) y del uso de interfaces.

Una variable es polimórfica si puede referenciar a objetos de diferentes formas.

3.3.1 Uso de herencia

Una variable que hace referencia a un objeto de una clase base puede reasignarse con una referencia a un objeto de cualquiera de sus clases derivadas. En consecuencia, se dice que la variable cambia de forma. Retomando el ejemplo de las clases que representan distintas cuentas bancarias, figura 2.10, se pueden hacer las siguientes observaciones:

- Se crea un objeto de la súper clase, usando el constructor con parámetros:

  ```
  CuentaBancaria miCuenta = new CuentaBancaria(numC, nomT, saldo);
  ```

- Se invoca al método *retiro()* de la clase *CuentaBancaria*:

  ```
  boolean resp;
  resp = miCuenta.retiro(monto);
  ```

- Se declara e instancia un objeto de la clase *CuentaCheques*, usando el constructor con parámetros:

  ```
  CuentaCheques tuCuenta = new CuentaCheques(numC, nomT, saldo);
  ```

- Se asigna un objeto de la clase derivada, *CuentaCheques*, a uno de la clase base, *CuentaBancaria*:

  ```
  miCuenta = tuCuenta;
  ```

- Se invoca al método *retiro()*, correspondiente a la clase *CuentaCheques* debido a que ahora la variable *miCuenta* está haciendo referencia a un objeto de dicho tipo:

  ```
  resp = miCuenta.retiro(monto);
  ```

Para saber qué forma tiene actualmente cierto objeto se puede hacer uso del operador instanceof, el cual tiene la siguiente sintaxis:

```
nombreObjeto instanceof nombreClase
```

Muy importante

✔ Bien hecho

✘ Mal hecho

✔ Declaro con la base y asigno derivadas

✘ Declaro con la derivada y asigno base

✘ Declaro derivada y asigno otra derivada

Regresa el valor true si *nombreObjeto* es de tipo *nombreClase* y false en caso contrario. El valor true también se obtiene si se pregunta si el objeto –de una clase derivada– es una instancia de una súper clase. Observe las siguientes instrucciones, basadas en las declaraciones hechas más arriba:

- La evaluación da el valor true, por lo tanto imprime el mensaje.

```
if (tuCuenta instanceof CuentaCheques)
    System.out.println("Sí es una instancia de la clase CuentaCheques");
```

- Esta evaluación también da el valor true –el objeto es una instancia de la súper clase CuentaBancaria–, por lo tanto imprime el mensaje.

```
if (tuCuenta instanceof CuentaBancaria)
    System.out.println("Sí es una instancia de la clase CuentaBancaria");
```

- La evaluación da el valor false –el objeto no es una instancia de la clase CuentaAhorro–, por lo tanto NO imprime el mensaje.

```
if (tuCuenta instanceof CuentaAhorro)
    System.out.println("Sí es una instancia de la clase CuentaAhorro");
```

Para invocar un método específico de una clase derivada en un objeto polimórfico, se debe realizar una conversión explícita a dicha clase. Así, recupera la "forma" de la misma y se le puede asociar el método en cuestión. Por ejemplo, suponga que se quiere actualizar la tasa de interés de una cuenta de ahorro y el objeto que la representa se almacenó en un variable polimórfica. Para que el método *setTasaInterés()* se reconozca se debe convertir explícitamente la variable al tipo *CuentaAhorro*. Para ello se utiliza el nombre de la clase, escrito entre paréntesis. Observe el siguiente código:

```
CuentaBancaria otraCuenta = new CuentaAhorro("Juan Pérez", 1500, 3.4);
((CuentaAhorro)otraCuenta).setTasaInterés(5.2);
```

Se puede usar la clase *Object* en lugar de una súper clase. De esta manera se obtiene mayor generalidad, ya que todas las clases son clases derivadas de la clase *Object*. Por lo tanto, al declarar una variable como una referencia a un *Object* se le podrá asignar referencias a cualquier otro tipo de clases. Observe el siguiente ejemplo:

- Se declara e instancia un objeto de la clase *Object*

```
Object unObjeto = new Object ( );
```

- Se reasigna con la referencia a un objeto de la clase *CuentaAhorro*

```
unObjeto = new CuentaAhorro("Manuela Larios", 10500,3.8);
```

- Se reasigna con la referencia a un objeto de la clase *Perro*

```
unObjeto = new Perro( );
```

- Se convierte explícitamente, usando la clase *Perro*, para poder invocar un método de dicha clase

  ((Perro)unObjeto).setPeso(30);

3.3.2 Uso de interfaces

Otra variante es usar interfaces para declarar variables polimórficas. En este caso, a la variable polimórfica se le podrán asignar referencias a cualquiera de las clases que implementen la interface. Usando el ejemplo que se presenta en la figura 2.7 se pueden hacer las siguientes asignaciones:

- Se declara una variable polimórfica de tipo *Mascota* y se instancia con la clase *Tortuga*

  Mascota otraMascota = new Tortuga("Tranquila", "Julián", "japonesa");

- Se reasigna con la referencia a un objeto de la clase *Perro*

  otraMascota = new Perro("Pocas Pulgas", "Labrador", 6);

- Se invoca a un método de la interface e implementado en la clase *Perro*

  otraMascota.come( );

- Para invocar un método específico de la clase que implementa a la interface, se requiere convertir explícitamente al tipo de la clase correspondiente. Por ejemplo:

  ((Perro)otraMascota).setPeso(30);

3.3.3 Alternativas para determinar el tipo de un objeto

Previamente se vio el uso del operador **instanceof** para determinar el tipo de un objeto guardado en una variable polimórfica. A continuación veremos otras dos alternativas para realizar esta operación.

En el lenguaje Java también se puede usar el **try catch()** con este propósito. En la sección del try se intenta trabajar con el objeto del cual se supone que es de un cierto tipo, y si no corresponde a la clase que se está intentando convertir, se detecta el error y se puede indicar qué hacer en dicho caso en la sección del catch. Observe el siguiente código:

```
CuentaBancaria otraCuenta = new CuentaAhorro("Juan Pérez", 1500, 3.4);

    try{
       double interés = ((CuentaAhorro)otraCuenta).calculaInterés( );
       System.out.println("\nInterés ganado: " + interés);
    }catch (Exception e){
        System.out.println("\nNO es una cuenta de ahorros\n");
}
```

Se trata de convertir explícitamente la variable *otraCuenta* a un objeto de la clase *CuentaAhorro*. Si esta operación se lleva a cabo con éxito (la variable almacena la referencia a un objeto de dicha clase), entonces se procede a invocar el método calculaInterés(). En caso contrario, se captura el error y se imprime un mensaje.

La segunda opción, también haciendo uso de recursos de Java, es comparando con el nombre de la clase que interesa. Todos los objetos heredan de la clase Object el método getClass() que da como resultado el nombre del paquete, punto y el nombre de la clase. Si se quiere usar sólo el nombre de la clase, al resultado de este método se le debe aplicar el método getSimpleName(). Observe el siguiente ejemplo:

```
CuentaBancaria unaCuenta = new CuentaAhorro("Julián Garduño", 3800, 4.5);

   if (unaCuenta.getClass( ).getSimpleName( ).equals("CuentaAhorro")){
      double interés = ((CuentaAhorro)unaCuenta).calculaInterés( );
      System.out.println("\nInterés ganado: " + interés);
   }
   else
      System.out.println("\nNO es una cuenta de ahorro");
```

Estos ejemplos pueden probarse por medio del código almacenado en el archivo PruebasCapítulo3.java, en el paquete cap3.

» Sobrescritura del método equals()

Cuando se presentó el tema de sobrescritura se mencionó el método equals() de la clase Object, y se dijo que puede ser sobrescrito recibiendo un parámetro Object. En este caso, antes de la comparación debe ser convertido explícitamente al tipo de la clase en la cual se esté incluyendo. Sin embargo, como se lo recibe como un Object, el tipo puede o no coincidir con el tipo al cual se lo quiere convertir. Para evitar posibles errores hay que asegurarse de que los tipos son los mismos, y para ello hay que utilizar algunas de las alternativas ya estudiadas: instanceof, getClass().getSimpleName() o try-catch. Los beneficios de la generalidad que se gana podrán apreciarse más claramente en los capítulos donde se estudien estructuras de datos.

A continuación se muestra un ejemplo de equals() para la clase Cliente (programa 3.8), suponiendo que dos clientes son iguales si tienen el mismo nombre. En el primer caso, si c no es una referencia nula y si es una instancia de la clase Cliente, entonces se la convierte a Cliente y se comparan los nombres. El segundo caso es similar, ya que se verifica que el nombre de la clase de c sea Cliente antes de hacer la conversión. Por último, en el tercer caso se hace la conversión dentro del try para manejar el error si no fuera del tipo Cliente.

Ejemplo 3.1

```
// Usando el instanceof
public boolean equals(Object c){
     boolean resp = false;

     if (c != null && c instanceof Cliente)
        resp = ((Cliente)c).nombre.equalsIgnoreCase(nombre);
     return resp;
}
```

```
// Usando el getClass().getSimpleName()
public boolean equals(Object c){
      boolean resp = false;

      if (c != null && c.getClass().getSimpleName().equals("Cliente"))
         resp = ((Cliente)c).nombre.equalsIgnoreCase(nombre);
      return resp;
 }

// Usando try-catch
public boolean equals(Object c){
      boolean resp;

      try{
         resp = ((Cliente)c).nombre.equalsIgnoreCase(nombre);
      }
      catch (Exception e){
         resp = false;
      }
      return resp;
}
```

3.4 CLASES GENÉRICAS

Las clases genéricas son clases en las cuales a alguno de sus miembros no se les asigna un tipo específico de dato. De esta manera, la definición se completa cuando se crean objetos de dicha clase. Esta característica permite declarar clases más generales, que pueden usarse en distintos escenarios.

En Java se pueden definir clases genéricas mediante el uso de la clase *Object* y mediante el uso del tipo genérico T. A continuación se presentan las principales características de ambos, sus similitudes y diferencias.

3.4.1 Clase Object

En Java se tiene la clase **Object**, que es una clase general y es la súper clase de todas las otras clases que se definan durante la solución de un problema. Es decir, cualquier clase que declaremos en Java es automáticamente una clase derivada de *Object*. [1]

Al declarar un miembro de una clase de tipo *Object*, éste podrá posteriormente almacenar datos de distintos tipos, basándose en los principios de herencia y polimorfismo, logrando así generalidad en la definición de la clase. Observe el ejemplo que se presenta en los programas 3.8 y 3.9. Se tiene la clase *Cliente*, la cual describe simplificadamente a un cliente. El atributo domicilio de esta clase se declaró de tipo *Object*. También se definió una clase *Dirección*, la cual define el concepto dirección.

[1] Esta clase cuenta con algunos métodos que pueden sobrescribirse, como el *toString()* y el *equals()*, para adaptarse a las clases derivadas, como ya se vio en secciones anteriores

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.9
 * Se define la clase Dirección.
 */
public class Dirección {
   String calle;
   int numExt, numInt;
   String ciudad;
   int CP;
   String país;

   public Dirección(){
   }

   public Dirección(String ca, int nE, int nI, String ciu, int cp, String pa){
      calle = ca;
      numExt = nE;
      numInt = nI;
      ciudad = ciu;
      CP = cp;
      país = pa;
   }

   public String toString(){
      StringBuilder cad = new StringBuilder();

      cad.append("\nCalle: " + calle + " " + numExt + " - " + numInt);
      cad.append("\nCP: " + CP + " - " + ciudad + "\n" + país);
      return cad.toString();
   }
}
```

A continuación, en el programa 3.10 se presentan algunas posibles instanciaciones de objetos tipo Cliente en los cuales se aprovecha la generalidad de su atributo domicilio. Es importante destacar que esa generalidad también presenta cierta limitación, ya que como se puede observar en la última parte del código, resulta necesario convertir explícitamente al tipo de dato dado, ya que el mismo se pierde al almacenarse en la variable tipo Object.

Programa 3.10 | **UsaClaseGenérica1**

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.10
 * Ejemplo de uso de clases genéricas, basado en los programas 3.8 y 3.9.
 */
public class UsaClaseGenérica1 {

   public static void main(String[] args) {

      /* Se declara e instancia un objeto de tipo Cliente, usando una cadena para darle
       * valor al atributo domicilio.
       */
      Cliente cliente1 = new Cliente("Joaquín Lobos", "México, D.F. - México");

      /* Se declara e instancia un objeto de tipo Cliente, usando un número entero
       * para darle valor al atributo domicilio.
       */
      Cliente cliente2 = new Cliente("Marina Estrada", 25);

      /* Se declara e instancia un objeto de tipo Direccion. */
      Dirección unaDirec = new Dirección("9 de julio", 587, 3, "Santa Fe", 3000, "Argentina");

      /* Se declara e instancia un objeto de tipo Cliente, usando un objeto de tipo Dirección
       * para darle valor al atributo domicilio.
       */
      Cliente cliente3 = new Cliente("Pilar Gómez", unaDirec);

      /* Se imprimen los 3 objetos, cada uno de ellos con distintos tipos de datos
       * asignados al atributo domicilio.
       */
      System.out.println(cliente1); // Tiene una cadena
      System.out.println(cliente2); // Tiene un número entero
      System.out.println(cliente3); // Tiene un objeto tipo Dirección

      /* Se cambia dinámicamente -durante la ejecución- el domicilio del
       * cliente2, asignándole un nuevo objeto de tipo Dirección.
       */
      cliente2.setDomicilio(new Dirección("Reforma", 820, 4, "México DF", 1050, "México"));
      System.out.println(cliente2);
```

```
        /* Como el atributo domicilio es de tipo Object, si se quiere recuperar con la forma
         * del dato que se le asignó, se debe convertir explícitamente.
         * En este caso la conversión es al tipo String.
         */
        String direc = (String) cliente1.getDomicilio( );
    }
}
```

Al ejecutar el programa se tendría la siguiente información. En el caso del cliente Marina Estrada se imprime dos veces; en la primera, su domicilio es un número entero, mientras que en la segunda son los valores de los atributos de un objeto tipo Dirección.

```
Nombre: Joaquín Lobos
Domicilio: México, D.F. - México

Nombre: Marina Estrada
Domicilio: 25

Nombre: Pilar Gómez
Domicilio:
Calle: 9 de julio 587 - 3
CP: 3000 - Santa Fe
Argentina

Nombre: Marina Estrada
Domicilio:
Calle: Reforma 820 - 4
CP: 1050 - México DF
México
```

La relación entre la clase Object y el polimorfismo se presentó en la sección dedicada a la herencia como medio para implementar esta característica de la POO.

3.4.2 Tipo T

Otra manera de definir clases genéricas es usando el tipo genérico **T**. Éste permite declarar un miembro de la clase sin especificar un tipo de dato particular, y el mismo podrá instanciarse de acuerdo a los requerimientos de cada problema. El efecto de usar el tipo T es similar al uso de la clase Object; sin embargo, al momento de instanciar un objeto se puede parametrizar forzando a usar un solo tipo. Esto también ofrece la ventaja de que el dato no pierde su forma al almacenarse en la variable tipo T, lo cual vimos que sucedía con la clase Object. Es decir, al usar una clase que tenga miembros tipo T se puede o no parametrizar (indicar un tipo de dato) al momento de instanciar un objeto. Si se parametriza, entonces todas las ocurrencias de T se reemplazan por el tipo elegido. En caso contrario, el tipo queda sin definir y podrá ser de cualquier tipo y se comporta de manera similar al Object.

En el programa 3.11 se presenta un ejemplo del uso del tipo T para darle generalidad a una clase y, por lo tanto, que un objeto declarado a partir de ella pueda adoptar distintas formas. Observe que el encabezado de la clase está acompañado de <T>, lo cual indica que la clase tiene al menos uno de sus miembros del tipo T.

Programa 3.11 | **Materia.java**

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.11
 * Se define una clase genérica usando el tipo T para uno de los atributos de la misma.
 * En este caso, el atributo calificación queda sin especificar su tipo. Por lo tanto, cada vez que
 * se instancie un objeto de tipo Materia se podrá indicar un tipo de dato para dicho atributo.
 */
public class Materia <T>{
  private int clave;
  private String nombre;
  private T calificación;

  public Materia(){
  }

  public Materia(int cla, String nom, T cal){
     clave = cla;
     nombre = nom;
     calificación = cal;
  }

  public String toString (){
    return "\nClave: " + clave + "\nNombre: " + nombre + "\nCalificación: " +
           calificación + "\n";
  }

  // El método recibe un parámetro de tipo T
  public void setCalificación(T cal){
     calificación = cal;
  }

  // El método da un resultado de tipo T
  public T getCalificación(){
     return calificación;
  }

  // Regresa el nombre de la materia
```

```
    public String getNombre() {
      return nombre;
    }
}
```

Para parametrizar se emplea la siguiente sintaxis:

NombreClase <tipoAsignado> varObj = new NombreClase ( );

El código que aparece más abajo muestra algunos ejemplos del uso de la clase genérica definida en el programa 3.11, así como de la parametrización hecha al declarar e instanciar objetos de tipo Materia.

Programa 3.12 | **UsaClaseGenérica2.java**

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.12
 * Ejemplo de uso de clase genérica, en este caso la clase Materia que tiene un
 * atributo tipo T.
 */
public class UsaClaseGenérica2 {

   public static void main(String[ ] args) {

      /* Se declara e instancia un objeto de tipo Materia con el tipo número entero para
       * asignarse al tipo T. Observe que se usa la clase Integer de Java porque se espera
       * una clase, no un dato simple.
       */
      Materia<Integer> mat1 = new Materia(20002, "Álgebra II", 8);

      /* Se declara e instancia un objeto de tipo Materia con el tipo String para asignarse
       * al tipo T.
       */
      Materia<String> mat2 = new Materia(5001, "Introducción a la Física", "Aprobada");

      /* Se declara e instancia un objeto de tipo Materia con el tipo número de doble precisión
       * para asignarse al tipo T. Observe que se usa la clase Double de Java porque se espera
       * una clase, no un dato simple.
       */
```

```
        Materia<Double> mat3 = new Materia(3010, "Programación I", 9.4);

        // Se imprimen las 3 materias.
        System.out.println("\n" + mat1 + mat2 + mat3);

        /* Se obtiene la calificación de la primer materia. Observe que se asigna directamente
         * a una variable numérica entera, sin necesidad de convertir explícitamente el dato.
         */
        int calif1 = mat1.getCalificación( );
        System.out.println("\nLa calificación obtenida es: " + calif1);


        /* Se le asigna una nueva calificación a la segunda materia, usando una constante de tipo
         * cadena de caracteres.
         */
        mat2.setCalificación("Reprobada");

        /* Se obtiene la calificación de la segunda materia. Observe que se asigna directamente
         * a una variable cadena, sin necesidad de convertir explícitamente el dato.
         */
        String calif2 = mat2.getCalificación( );
        System.out.println("\nLa calificación obtenida es: " + calif2);
    }
}
```

3.4.3 Tipo T y herencia

El tipo T se puede emplear en combinación con la herencia o las interfaces para dar generalidad, pero acotando dentro de un cierto grupo de clases.

Es decir, al especificar el tipo T se puede indicar que éste debe ser de alguna de las clases derivadas de una cierta súper clase o interface. De esta manera, si bien se deja flexibilidad para definir el tipo de uno o varios miembros de la clase, se está restringiendo a que se definan a partir de un conjunto de clases.

En el programa 3.13 se presenta un ejemplo. Se usan las clases de la figura 2.10, donde está la clase base *CuentaBancaria* y las derivadas *CuentaCheques* y *CuentaAhorro*. El código de las mismas puede consultarse en los programas 2.16, 2.17 y 2.18. En el encabezado de la clase se establece que el tipo **T** debe ser alguna de las clases derivadas de *CuentaBancaria*. Así, a cada proveedor se le pagará su mercadería o servicio con depósito en una cuenta de cheques o en una cuenta de ahorros.

Programa 3.13 | **Proveedor.java**

```
package cap3;

import cap2.CuentaBancaria;

/**
 * @author Silvia Guardati
 * Programa 3.13
 * Uso del tipo T restringiendo el tipo a emplear para asignarle valor.
 * En este caso sólo podrá ser alguna clase derivada a partir de CuentaBancaria.
 */

public class Proveedor <T extends CuentaBancaria>{
   String nombre;
   int clave;
   T cuentaDepósito;

   public Proveedor(){
   }

   public Proveedor(String nom, int cla, T cuenta){
      nombre = nom;
      clave = cla;
      cuentaDepósito = cuenta;
   }

   public String toString (){
      StringBuilder cad = new StringBuilder();

      cad.append("\nNombre: " + nombre + "\nClave: " + clave + "\n");
      cad.append("El pago se realiza con depósito en la cuenta: " + cuentaDepósito.getNumCta());
      return cad.toString();
   }

   public T getCuentaDepósito() {
      return cuentaDepósito;
   }

   public void setCuentaDepósito(T cuentaDepósito) {
      this.cuentaDepósito = cuentaDepósito;
   }
}
```

El código que aparece más abajo muestra un ejemplo del uso de la clase definida en el programa 3.13.

Programa 3.14 **UsaTyHerencia.java**

```
package cap3;

/***
 * @author Silvia Guardati
 * Programa 3.14
 * Se crean dos objetos de tipo Proveedor, dándole a T valor de CuentaCheques y
 * CuentaAhorro respectivamente.
 */
public class UsaTyHerencia {

   public static void main(String[ ] args) {
      CuentaCheques cuentaProv1 = new CuentaCheques("Juan Pérez", 5000);
      // T será sólo de tipo CuentaCheques
      Proveedor<CuentaCheques> prov1 = new Proveedor("Juan Pérez", 105, cuentaProv1);

      CuentaAhorro cuentaProv2 = new CuentaAhorro("Francisco López", 1000, 3.2);
      /* T será de tipo CuentaAhorro, pero al no parametrizar la declaración se
       * le podrá asignar posteriormente una cuenta de cheques.
       */
      Proveedor prov2 = new Proveedor("Darío García", 103, cuentaProv2);

      // Se imprimen los dos proveedores
      System.out.println(prov1);
      System.out.println(prov2);

      // Se accede al atributo cuentaDepósito, el cual mantiene su tipo
      prov1.getCuentaDepósito( ).depósito(1000);

      // Se le asigna una cuenta de cheques al segundo proveedor.
      CuentaCheques otraCuenta = new CuentaCheques("Darío García", 0.0);
      prov2.setCuentaDepósito(otraCuenta);
   }
}
```

El tema de clases genéricas se retomará en capítulos subsecuentes, dedicados al estudio de estructuras de datos, donde se podrá apreciar en forma más evidente el potencial del uso de la clase Object y del tipo T.

3.4.4 Tipo T y polimorfismo

El tipo T se puede utilizar para declarar objetos polimórficos (similar a usar la clase Object). Si al instanciar un objeto se parametriza, entonces se pierde la posibilidad de que se maneje como polimórfico, tal como se vio en la sección previa.

Si al instanciar un objeto con uno o más miembros del tipo T no se parametriza, entonces el objeto se podrá manejar como un objeto polimórfico. Retomando la clase Materia del programa 3.11, a continuación se presenta un ejemplo de cómo podría crearse una variable polimórfica de este tipo. Las variables que referencian a objetos tipo Materia pueden cambiar su forma, ya que el atributo "calificación", al ser de tipo T y no haber sido parametrizado, puede adoptar distintas formas. Analice el programa 3.15.

Programa 3.15 **UsaTyPolimorfismo.java**

```
package cap3;

/**
 * @author Silvia Guardati
 * Programa 3.15
 * Ejemplo del uso del tipo T para generar variables polimórficas.
 */
public class UsaTyPolimorfismo {

   public static void main(String[] args) {

      /* Se declaran e instancian objetos de tipo Materia sin indicar un tipo para
       * T, por lo tanto las variables podrán funcionar como polimórficas.
       */
      Materia mat1 = new Materia(20002, "Álgebra II", 8);
      Materia mat2 = new Materia(5001, "Introducción a la Física", "Aprobada");
      Materia mat3 = new Materia(3010, "Programación I", 9.4);

      // Se imprimen las 3 materias.
      System.out.println("\n" + mat1 + mat2 + mat3);

      /* Se obtiene la calificación de la primer materia. Observe que se debe convertir
       * explícitamente el dato a tipo int antes de asignarlo a la variable.
       */
      int calif1 = (int) mat1.getCalificación();
      System.out.println("\nLa calificación obtenida en " + mat1.getNombre() + " es: " + calif1);

      /* Se le asigna una nueva calificación a la primer materia, usando una
       * constante tipo cadena de caracteres (antes un entero).
       */
      mat1.setCalificación("Aprobada");
```

```
        /* Se obtiene la calificación de la primer materia. Observe que se debe convertir
         * explícitamente el dato a tipo String antes de asignarlo a la variable.
         */
        String calif2 = (String)mat1.getCalificación();
        System.out.println("\nLa calificación obtenida en " + mat1.getNombre() + " es: " + calif2);
    }
}
```

Al instanciarse la variable *mat1* se le asigna un entero al atributo calificación. Sin embargo, como no se parametrizó, al recuperar el valor se debe convertir explícitamente al tipo int antes de asignarlo a la variable *calif1*. Por la misma razón, posteriormente se le puede asignar un dato de distinto tipo; en el ejemplo, una cadena. Es importante señalar que al no parametrizar, la recuperación de la información (convirtiéndola explícitamente al tipo esperado) puede ser causa de error. Es decir, dado que no tenemos control sobre lo que se asigna al tipo genérico, tampoco podemos estar seguros del tipo al momento de recuperar la información. Una solución más robusta es intentar la conversión dentro de un try catch, de tal forma que si el tipo no coincide con el esperado se pueda manejar el error en tiempo de ejecución.

3.5 PAQUETES DE CLASES

Cuando en una aplicación se desarrollan varias clases y/o interfaces para la solución de un problema, puede resultar conveniente agruparlas. En Java se pueden formar paquetes de clases, y se recomienda que sea por similitud semántica. Es decir, la relación es por contenido.

Un paquete se define usando la directiva **package** y el nombre asignado al mismo, terminando con punto y coma. Ésta debe ser la primera línea en cada uno de los archivos correspondientes a las clases que forman el paquete. El paquete se guarda en una carpeta que lleva su mismo nombre. Por convención, el nombre debe ser un sustantivo y debe escribirse todo con letras minúsculas. Un ejemplo de paquete es el que agrupa todos los programas del capítulo 3 de este libro (cap3). Como se puede observar, todos los programas tienen como primera línea de código:

```
package cap3;
```

En UML los paquetes se representan con un rectángulo con una pestaña en el extremo superior izquierdo, donde se escribe el nombre del paquete. Observe la figura 3.2.

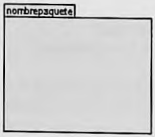


Figura 3.2 Representación UML de un paquete

La figura 3.3 muestra un ejemplo de un paquete que almacena una interface y dos clases, los tres elementos relacionados por los conceptos que representan.

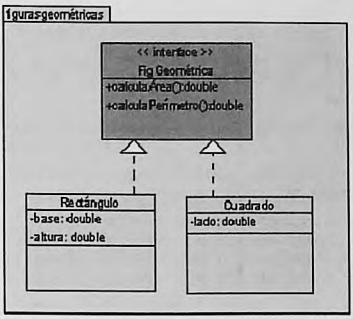


Figura 3.3 Ejemplo de paquete

Cuando en una aplicación se requiere usar una o más clases de un paquete distinto al que se encuentra, se importa todo el paquete o sólo las clases que interesen. Para ello se usa la directiva **import** seguida del nombre del paquete, indicando además si se incluyen todas las clases o los nombres de las clases que se desee.

```
import nombrepaquete.*; // Para incluir todas las clases del paquete

import nombrepaquete.NombreClase; // Para incluir sólo la clase NombreClase
```

Por ejemplo, en el caso de las clases del paquete de la figura 3.3, se haría:

```
import  figurasgeométricas.*; // Para importar la interface y las dos clases

import  figurasgeométricas.Cuadrado; // Para importar sólo la clase Cuadrado
```

Un paquete puede almacenar otros paquetes, formando así una jerarquía de paquetes. Esto es muy útil para organizar los programas/clases que forman un sistema a partir de varios subsistemas. Es una buena práctica que cada subsistema se agrupe en un paquete y, a su vez, las clases relacionadas por contenido pueden formar un nuevo paquete dentro del paquete del subsistema. En el ejemplo de la figura 3.4 se tienen tres subpaquetes: guis, reportes y dominio. A su vez, el paquete dominio está formado por tres clases. En el diagrama también se representa una dependencia entre los subpaquetes guis y reportes con el subpaquete dominio. Entre la clase Empresa y la clase Vehículo hay una asociación (relación que, en este caso, podría indicar que una empresa tiene instancias de Vehículo). Por último, hay una relación de herencia entre Vehículo y Camión.

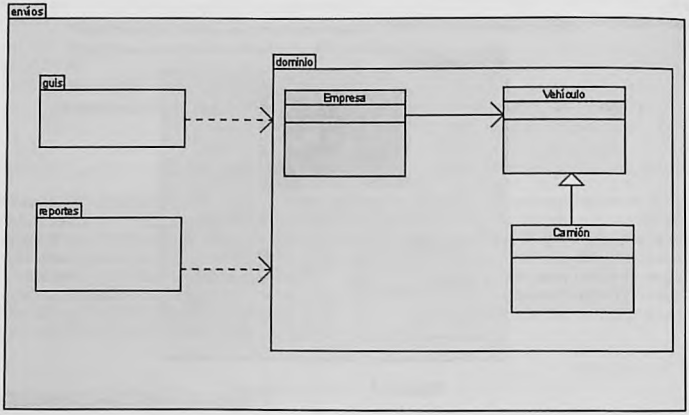


Figura 3.4 Ejemplo de jerarquía de paquetes

Para importar la clase Vehículo se usará la sintaxis paquete.subpaquete.Clase, tal como se muestra a continuación:

```
import envíos.dominio.Vehículo;
```

Buenas prácticas

- Agrupar las clases teniendo en cuenta su similitud semántica.
- Los nombres de los paquetes se escriben con minúsculas.
- Cada subsistema –que forma parte de un sistema– se almacena en un subpaquete.

3.6 PRUEBAS UNITARIAS

En el primer capítulo de esta obra se presentó una introducción a las pruebas de software, dedicando especial interés a las pruebas de escritorio y dejando para este capítulo las pruebas unitarias.

Las pruebas unitarias son las que se aplican sobre los componentes de manera individual, antes de integrarlos para formar una pieza de software más compleja y/o más grande. Por lo tanto, en un ambiente de desarrollo orientado a objetos, las pruebas deben aplicarse a nivel de métodos y luego de clases, antes de integrar las clases para formar un (sub)sistema.

En NetBeans, por medio de JUnit, se pueden generar pruebas unitarias de manera bastante simple, ya que la herramienta crea automáticamente métodos de prueba para cada uno de los métodos de la clase que se quiere probar. Es importante tener en cuenta que sólo genera los encabezados de dichos métodos, quedando a car-

go del programador la implementación de los mismos y la elección del conjunto de datos de prueba. Para llevar a cabo esta última actividad se debe tener conocimiento del problema que se está resolviendo. Algunos criterios para la selección de los datos de prueba son: a) valores dentro del rango esperado, b) valores en los extremos del rango esperado, c) valores fuera del rango esperado, etc. A continuación se presentan los pasos a seguir, usando NetBeans, para generar las pruebas unitarias para los métodos de una cierta clase.

Una vez posicionado el cursor sobre la clase (que se quiere probar), dentro del árbol que muestra la estructura del proyecto (en la parte superior izquierda de la pantalla de NetBeans), bajo los "Source Packages" del proyecto, se debe:

1. Dar click al botón derecho del mouse y elegir Tools → Create (JUnit) Tests (se sugiere aceptar las opciones que la herramienta va a desplegar).
2. NetBeans va a crear una clase cuyo nombre será NombreClaseTest, es decir, el nombre de la clase seguida de la palabra Test. Dicha clase estará en el paquete "Test Packages" del proyecto.
3. En la clase NombreClaseTest se incluyen métodos vacíos llamados "setUpClass", "tearDownClass", "setUp" y "tearDown". Ninguno de estos cuatro métodos es obligatorio, y sólo se tienen que incluir si existe la necesidad. NetBeans permite, al momento de crear la clase NombreClaseTest, indicar que no se desean incluir los últimos dos de los métodos (los otros dos siempre se generan automáticamente, pero se pueden eliminar en forma manual, si así se desea).
4. Además de los métodos mencionados en el punto anterior, también se genera un método constructor para la clase NombreClaseTest y un método de prueba para cada uno de los métodos de la clase NombreClase, con excepción de los métodos constructores. El método constructor de NombreClaseTest está vacío, mientras que los otros métodos de prueba tienen algunas líneas de código generado automáticamente por NetBeans. Si un método de la clase NombreClase se llama *haceAlgo*, el método de prueba correspondiente se denominará *testHaceAlgo*. Si el método *haceAlgo* de la clase NombreClase regresa un valor real (*double*), el contenido del método de prueba *testHaceAlgo* se verá como se muestra a continuación:

```
public void testHaceAlgo() {
    System.out.println("haceAlgo");
    NombreClase instance = new NombreClase();
    double expResult = 0.0;
    double result = instance.haceAlgo();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
```

En la variable *instance* se asigna una instancia de la clase NombreClase y se declaran algunas variables auxiliares. Posteriormente, cada programador deberá hacer los ajustes necesarios para llevar a cabo la prueba. Por ejemplo, en la variable *expResult* se deberá asignar el valor esperado como consecuencia de ejecutar el método *haceAlgo*. A su vez, en la variable *result* se asigna el resultado del método invocado a través de la instancia creada. Por último, se compara el valor obtenido por la ejecución con el valor esperado (utilizando *assertEquals*). Este método, si se utiliza para comparar valores reales como en este caso, necesita un tercer argumento (no mostrado) que representa una épsilon (factor de tolerancia en la comparación). La llamada al método *fail* va a hacer que la prueba falle automáticamente, por lo tanto hay que quitarla por completo antes de ejecutar la prueba (o ponerla como comentario).

5. Después de adaptar todos los métodos de prueba, se deben ejecutar. Para ello, con el mouse sobre la clase NombreClaseTest se elige la opción Run File (se ejecuta aunque no existe un método main en la clase NombreClaseTest). En la ventana de salida (generalmente en la parte inferior derecha de la ventana de NetBeans) aparecerá el resultado de ejecutar las pruebas; se despliega un resultado por cada uno de los métodos de prueba y un resultado global, que es la conjunción de los resultados individuales. Existen algunas variantes del método assert, por ejemplo: assertFalse/True, assertNotNull, assertSame, assertFail, entre otras.

Retomando la clase Rectángulo del programa 2.2, usando JUnit se generó la clase RectánguloTest, cuyo código se muestra en el programa 3.16.

Programa 3.16 **RectánguloTest.java**

```
package cap3;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Silvia Guardati
 * Programa 3.16
 * Clase de prueba correspondiente a la clase Rectángulo, programa 2.2
 * El código -excepto este comentario- está como lo generó JUnit.
 */
public class RectánguloTest {

    public RectánguloTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }
```

```
    @After
    public void tearDown( ) {
    }

    /**
     * Test of calculaArea method, of class Rectángulo.
     */
    @Test
    public void testCalculaArea( ) {
        System.out.println("calculaArea");
        Rectángulo instance = new Rectángulo( );
        double expResult = 0.0;
        double result = instance.calculaArea( );
        assertEquals(expResult, result, 0.0);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }

    /**
     * Test of calculaPerímetro method, of class Rectángulo.
     */
    @Test
    public void testCalculaPerímetro( ) {
        System.out.println("calculaPerímetro");
        Rectángulo instance = new Rectángulo( );
        double expResult = 0.0;
        double result = instance.calculaPerímetro( );
        assertEquals(expResult, result, 0.0);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }
}
```

El código del programa 3.16 se ajusta para probar el cálculo del área y del perímetro de un rectángulo, que son los dos métodos programados en la clase Rectángulo. El código ya modificado se presenta en el programa 3.16 (modificado). Observe que se quitaron todos los otros métodos generados automáticamente. Además, se eliminó la instrucción fail de los métodos. Este programa se puede consultar a la carpeta "test" del proyecto que complementa este libro.

```
package cap3;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
```

```
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 * @author Silvia Guardati
 * Programa 3.16 (modificado)
 * Clase de prueba correspondiente a la clase Rectángulo, programa 2.2
 * El código generado por JUnit se modificó para probar los métodos calculaArea( ) y
 * calculaPerímetro( ).
 */
public class RectánguloTest {

    public RectánguloTest( ) {
    }

    /**
     * Test of calculaArea method, of class Rectángulo.
     */
    @Test
    public void testCalculaArea( ) {
        System.out.println("calculaArea");
        Rectángulo instance = new Rectángulo(4.0, 2.0);
        double expResult = 8.0;  // Resultado esperado para un rectángulo de 4 x 2
        double result = instance.calculaArea( );
        assertEquals(expResult, result, 0.0);
    }

    /**
     * Test of calculaPerímetro method, of class Rectángulo.
     */
    @Test
    public void testCalculaPerímetro( ) {
        System.out.println("calculaPerímetro");
        Rectángulo instance = new Rectángulo(4.0, 2.0);
        double expResult = 12.0; // Resultado esperado para un rectángulo de 4 x 2
        double result = instance.calculaPerímetro( );
        assertEquals(expResult, result, 0.0);
    }
}
```

Por la importancia de las pruebas en el aseguramiento de la calidad de los productos de software, se insiste en que todos los componentes generados deben ser probados. En aquellos que son críticos, pueden aplicarse diversos tipos de pruebas.

3.7 RESUMEN

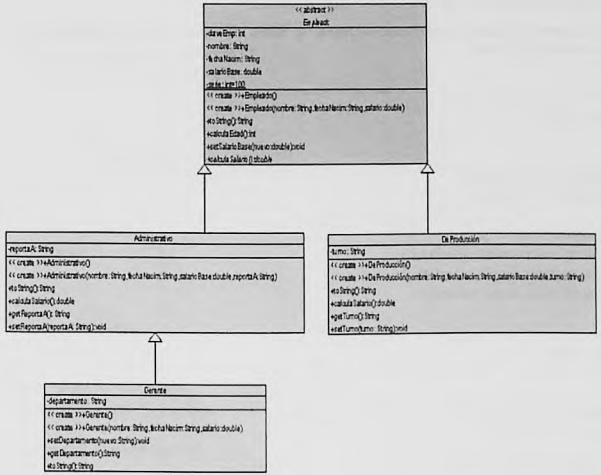
En este capítulo se presentaron conceptos avanzados de la POO como las clases abstractas, las clases genéricas y el polimorfismo. Estos temas, junto con los del capítulo 2, completan el estudio de la POO en esta obra.

Además se incluyeron otros temas, como los paquetes de clases, los cuales se usan para agrupar clases dando origen a subsistemas dentro de un producto de software de cierto tamaño. Otro tema tratado es el de las pruebas unitarias, las cuales son fundamentales en el desarrollo de software de calidad.

Todos los temas se complementaron con ejemplos para ayudar al lector a la comprensión de los mismos.

3.8 EJERCICIOS

3.1 Considere el siguiente diagrama de herencia entre clases, donde la súper clase es una clase abstracta con un método abstracto. Este método tendrá que implementarse en cada clase derivada de acuerdo con la forma en que se calcule el salario según el empleado sea un administrativo o uno de producción.



a. Defina todas las clases que aparecen en el diagrama. Complete con los atributos y métodos que crea conveniente.

b. Escriba un programa de aplicación que pueda: 1) crear objetos del tipo de cada una de las clases derivadas, 2) crear objetos polimórficos, usando la clase abstracta, 3) imprimir todos sus datos, 4) cambiar el departamento que tiene a su cargo un gerente, 5) cambiar el turno de un empleado de producción, 6) calcular e imprimir el sueldo de un gerente y de un empleado de producción, 7) cambiar el domicilio de un gerente (¿Qué atributo requiere agregar? ¿A qué clase?), 8) cambiar el nombre de la persona a quien reporta un empleado administrativo. Del 3 al 8 deberán resolverse usando variables polimórficas e instancias de las clases concretas.

3.2 Dibuje el diagrama de clases correspondiente al siguiente código. Posteriormente, en dicho diagrama agregue tres subclases de la clase Triángulo para definir los triángulos equiláteros, isósceles y escalenos, respectivamente. Considere que la herencia es un mecanismo que promueve el reuso de recursos, así como la representación de jerarquías entre conceptos.

- ¿Se requieren atributos/métodos adicionales?
- ¿Se requiere sobrecargar o sobreescribir algún método?
- ¿Qué ventaja/desventaja ofrece esta representación?
- Programe las clases incorporadas al diagrama. Estas tres nuevas clases deberán ser declaradas con el modificador "final".

```
package cap3;

/**
 * @author Silvia Guardati
 * Ejercicio 2 - Capítulo 3
 */
public abstract class Figura {
   public abstract double calculaÁrea( );
   public abstract double calculaPerímetro( );
   public abstract String toString( );
}

package cap3;

/**
 * @author Silvia Guardati
 * Ejercicio 2 - Capítulo 3
 */
public class Triángulo extends Figura{
```

```
    private double lado1, lado2, lado3;

    public Triángulo() {
    }

    public Triángulo(double lado1, double lado2, double lado3) {
       this.lado1 = lado1;
       this.lado2 = lado2;
       this.lado3 = lado3;
    }

    @Override
    public double calculaÁrea() {
       double s, área;

       s = 0.5 * (lado1 + lado2 + lado3);
       área = Math.sqrt(s * (s - lado1) * (s - lado2) * (s - lado3));
       return área;
    }

    @Override
    public double calculaPerímetro() {
       return lado1 + lado2 + lado3;
    }

    @Override
    public String toString() {
       return "\nLados del triángulo: " + lado1 + " - " + lado2 + " - " + lado3;
    }
}

package cap3;


/**
 * @author Silvia Guardati
 * Ejercicio 2 - Capítulo 3
 */
public abstract class Cuadrilátero extends Figura{
   protected double lado1, lado2, lado3, lado4;

   protected Cuadrilátero() {
   }
```

```
    protected Cuadrilátero(double lado1, double lado2, double lado3, double lado4) {
        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
        this.lado4 = lado4;
    }
}
```

```
package cap3;

/**
 * @author Silvia Guardati
 * Ejercicio 2 - Capítulo 3
 */
public class Cuadrado extends Cuadrilátero{

    public Cuadrado( ) {
    }

    public Cuadrado(double lado) {
        super(lado, lado, lado, lado);
    }

    @Override
    public double calculaÁrea( ) {
        return lado1 * lado2;
    }

    @Override
    public double calculaPerímetro( ) {
        return lado1 * 4;
    }

    @Override
    public String toString( ) {
        return "\nLado del cuadrado: " + lado1;
    }
}
```

```
package cap3;

/**
 * @author Silvia Guardati
 * Ejercicio 2 - Capítulo 3
```

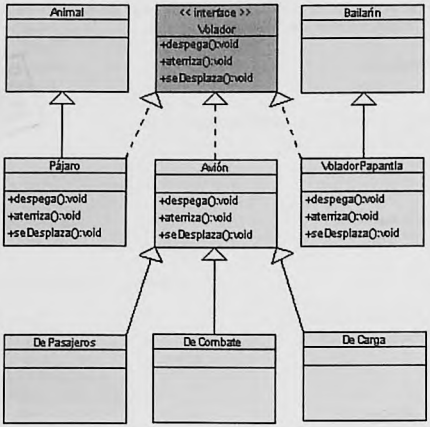
```
 */
public class Rectangulo extends Cuadrilátero{

  public Rectangulo() {
  }

  public Rectangulo(double lado1, double lado2) {
    super(lado1, lado2, lado1, lado2);
  }

  @Override
  public double calculaÁrea() {
    return lado1 * lado2;
  }

  @Override
  public double calculaPerímetro() {
    return 2 * (lado1 + lado2);
  }

  @Override
  public String toString() {
    return "\nLados del rectángulo: " + lado1 + " - " + lado2;
  }
}
```


3.3 Retome las clases definidas en el problema 2. Escriba un método main en el cual:

- **a.** Declare 3 objetos tipo Triángulo y 2 de tipo Cuadrilátero. Luego asigne a cada uno de los triángulos un objeto diferente, según los 3 tipos de triángulo definidos. Haga lo correspondiente con los cuadriláteros.
- **b.** Calcule e imprima el área y el perímetro de cada una de las figuras geométricas definidas.
- **c.** Imprima las dimensiones de cada una de las figuras.

3.4 Considere el siguiente esquema donde se representa una interface y relación de herencia entre varias clases.

- **a.** Defina todas las clases, subclases y la interface que aparece en el esquema. Complete con los atributos y métodos que crea conveniente.
- **b.** Escriba un programa de aplicación que pueda: 1) crear objetos del tipo de cada una de las clases, 2) crear objetos polimórficos, aprovechando las relaciones de herencia y la interface, 3) imprimir todos sus datos, 4) usar los métodos de cada clase –a partir de objetos de la clase base o interface (polimorfismo) o de objetos de clases derivadas.



3.5 Considerando las clases e interface programadas en el problema 4, indique si los siguientes enunciados son verdaderos o falsos.

a. En la clase Pájaro no se pueden incluir otros métodos si los mismos no fueron especificados en la interface Volador.

b. Un objeto de la clase Pájaro tendrá todos los atributos de la clase Animal y todos los métodos de la interface Volador.

c. A un objeto declarado de tipo Avión se le podrá asignar un objeto de tipo DeCombate.

d. A un objeto declarado de tipo Volador se le podrá asignar un objeto de tipo DeCarga.

e. A un objeto declarado de tipo Volador se le podrá asignar un objeto de tipo Pájaro.

f. Objetos de las clases DePasajeros, Pájaro y VoladorPapantla tienen cierto comportamiento en común.

3.6 Retome la clase Materia (programa 3.11) e indique si las siguientes líneas de código son correctas o no. Justifique su respuesta.

a. Materia <int> mat1 = new Materia(5004, "Historia del Arte", 8);

b. Materia mat2 = new Materia(5010, "Historia de Egipto", 9);

c. Materia <Integer> mat3 = new Materia(5015, "Historia de Grecia", 9);

d. Materia mat4 = new Materia(5012, "Historia de Europa Central", "Excelente");

e. Materia <Calificac> mat5 = new Materia(5000, "Historia I", new Calificac());

3.7 Retome el ejercicio 16 del capítulo 2 y declare variables polimórficas usando la clase CuentaBancaria. A estas variables asígneles objetos construidos usando las clases CuentaAhorro y PlazoFijo y use los métodos de dichas clases para conocer los datos de las cuentas, así como para realizar depósitos y retiros.

3.8 Retome el ejercicio 17 del capítulo 2 y declare tres variables polimórficas a partir de la interface Fig Geométrica. A cada una de esas variables asígnele un objeto de tipo Círculo, Rectángulo y Triángulo respectivamente. Utilice los métodos definidos para conocer el área y el perímetro de cada una de las figuras.

3.9 Señale las principales diferencias entre *clases abstractas e interfaces*.

3.10 Por medio de JUnit diseñe e implemente pruebas unitarias para las clases Círculo (ejercicio 11, del capítulo 2), Cuadrado y Equilátero (ejercicio 2 de este capítulo), de tal manera que pruebe si los métodos para calcular el área y el perímetro están obteniendo resultados correctos.

ARREGLOS 4

Contenido

Competencias

- Explicar el concepto de estructuras de datos y, en particular, el de arreglos.
- Presentar las principales operaciones en arreglos.
- Presentar los arreglos paralelos.
- Mostrar el uso de los arreglos en la solución de problemas algorítmicos.

4.1 INTRODUCCIÓN

En los tres primeros capítulos se introdujeron los datos simples, como números y caracteres, y las clases para representar conceptos. En ambos casos cada variable es capaz de almacenar un solo dato. Es decir, con una variable numérica se guarda en memoria un único número (que podrá variar durante la ejecución del programa), mientras que en una variable cuyo tipo sea una clase se guarda la referencia a un objeto, instancia de dicha clase. Sin embargo, hay problemas en los cuales se requiere manejar mucha información relacionada, y si se pretendiera almacenarla en variables independientes entre sí, sería muy difícil y, en algunos casos, imposible procesarla. La solución es usar **estructuras de datos** para el almacenamiento con los correspondientes algoritmos para la manipulación de la información.

Una estructura de datos es una colección de elementos que pueden ser de tipo simple o no. Cada estructura determina la manera en que se almacenan los datos y, en consecuencia, la manera en que se tendrá acceso a ellos. En este capítulo se presentan los **arreglos**, que son una estructura de datos muy útil por ser fáciles de entender y de usar.

Muy importante

- Colección finita de datos: se debe especificar el número máximo de elementos a guardar.
- Colección estática de datos: durante la ejecución no se puede alterar su tamaño.
- Colección homogénea de datos: todos los elementos son del mismo tipo.
- Colección lineal: cada elemento tiene un único predecesor y un único sucesor, con excepción del primero y del último, respectivamente.
- Colección ordenada de datos: se puede identificar cuál es el primer elemento, cuál es el segundo y así hasta el final.

Un arreglo es una **estructura lineal** y **ordenada**; lineal, porque cada elemento tiene un único predecesor y un único sucesor, con excepción del primero, que no tiene predecesor, y del último, que no tiene sucesor. Ordenada, porque sus elementos ocupan posiciones dentro del arreglo que permiten establecer un orden entre ellos; es decir, se puede saber cuál es el primero, cuál el segundo y así sucesivamente. También se caracteriza por ser una **estructura homogénea**, porque todos sus elementos son del mismo tipo. Por último, se dice que es una **estructura estática**, ya que durante la ejecución del programa no puede cambiar de tamaño.

4.2 COMPONENTES DE UN ARREGLO

Todo arreglo cuenta con un **nombre** que identifica a toda la estructura, los **elementos** que también reciben el nombre de componentes o casillas del arreglo, y los **índices**, que son los que permiten el acceso a cada uno de los elementos. Gráficamente, un arreglo puede verse como muestra la figura 4.1. El *nombreArreglo* es el nombre de todo el arreglo y los números que acompañan a cada casilla son los índices.

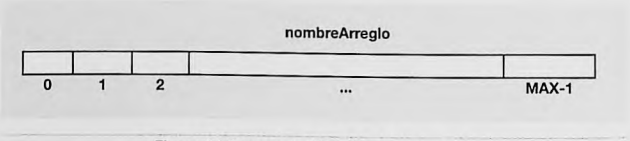


Figura 4.1 Representación gráfica de un arreglo

Cada una de las casillas se identifica por medio del índice, que debe tomar un valor único dentro de un rango de posibles valores. En Java, los índices son números enteros que van desde 0 hasta el máximo dado al arreglo, menos uno. Por ejemplo, si el arreglo fue definido con un máximo de 20 elementos, entonces sus casillas se enumeran del 0 al 19.

Con el nombre se referencia a toda la estructura, mientras que el nombre acompañado del índice escrito entre *[]* se utiliza para señalar sólo al elemento almacenado en la posición dada por el índice. Es el índice el que determina la posición (el orden) que ocupan los elementos; es decir, el elemento almacenado en la posición 0 es el primer elemento del arreglo, el que ocupa la posición 1 es el segundo, y así hasta el elemento del último índice.

- *nombreArreglo* → hace referencia a toda la estructura de datos, todo el arreglo
- *nombreArreglo[0]* → hace referencia sólo al valor almacenado en la primera casilla
- *nombreArreglo[n]* → hace referencia al valor almacenado en la enésima + 1 casilla (recuerde que empiezan a enumerarse a partir del 0)

4.3 DECLARACIÓN DE ARREGLOS EN JAVA

Al declarar un arreglo en Java se crea una variable que tiene la capacidad de almacenar varios datos bajo un mismo nombre. A la vez, con el nombre y el índice se tiene acceso a cada uno de sus componentes. Internamente, un arreglo en Java es un objeto; por tanto, según lo visto en el capítulo anterior, el arreglo debe declararse e instanciarse. Como cualquier otro objeto, puede hacerse en una sola línea de código o en dos.

```
tipo nombreArreglo[ ] = new tipo [MAX];  // Declaración e instanciación del arreglo
o
tipo nombreArreglo[ ];  // Declaración del arreglo
nombreArreglo = new tipo [MAX];  // Instanciación del arreglo
```

donde:

- *tipo* es algún tipo de dato válido en Java
- *nombreArreglo* es el nombre del arreglo. Se sugiere que sea representativo de los datos que almacena
- *MAX* es el total de elementos o casillas que tendrá el arreglo
- Los *[]* en el lado izquierdo de la expresión pueden ir a la izquierda o a la derecha de *nombreArreglo*

Una vez declarado el arreglo, se reserva en memoria tanto espacio como se haya indicado con MAX y dependiendo del tipo de dato usado. Sin embargo, esto no quiere decir que necesariamente todas las casillas deban estar ocupadas. El problema determina el total de elementos con los que se trabajará. Por tanto, es importante destacar que, en general, cuando se tiene un arreglo, también se requiere una variable extra que indique el total de elementos que almacena.

Si se desea conocer el máximo espacio reservado en un arreglo (por ejemplo, en un subprograma en que se recibe al arreglo como parámetro), se puede usar:

```
nombreArreglo.length
```

Ejemplo 4.1

El código de este ejemplo puede consultarse en el programa PruebaArreglos.java del paquete cap4, del proyecto de NetBeans que complementa este capítulo. Se sugiere al lector que lo analice y pruebe para reafirmar los conceptos estudiados en esta sección.

```
final int MAX = 10; // Se declara una constante.
// Arreglo que puede almacenar máximo 10 números enteros.
int edades[ ] = new int[MAX];
// Arreglo que puede almacenar máximo 10 números de doble precisión.
double[ ] números = new double[MAX];
// Arreglo que puede almacenar máximo 10 cadenas de caracteres.
String[ ] nombres;
nombres = new String[MAX];
```

Los arreglos pueden declararse e inicializarse en una sola línea. Observe el siguiente código:

```
String nombres[ ] = {"Julián", "Marcos", "Paolo", "Daniel", "Germán"};
int enteros[ ] = {15, 21, 8, 14};
```

Es importante tener en cuenta que, en estos casos, los arreglos creados tienen un máximo de elementos igual a la cantidad de elementos asignados (5 en el primer ejemplo y 4 en el segundo). Asimismo, los elementos se asignan de la casilla 0 en adelante, según el orden en el cual se dan entre las llaves. Por tanto, el primer ejemplo es equivalente a hacer:

```
String nombres[ ] = new String[5];
nombres[0] = "Julián";
nombres[1] = "Marcos";
nombres[2] = "Paolo";
nombres[3] = "Daniel";
nombres[4] = "Germán";
```

Como se mencionó más arriba, para conocer el total de casillas reservadas para un cierto arreglo se puede hacer uso del atributo *length*, que es propio de todos los arreglos en Java.

```
// Imprime el total de casillas reservadas para el arreglo edades.
System.out.println("Total de casillas reservadas para edades: " + edades.length);
```

Los arreglos también pueden ser declarados a partir de alguna clase previamente definida. Más adelante, en este capítulo, se volverá sobre este tema.

Muy importante

- Cuando se trabaja con un arreglo, salvo algunas excepciones, SIEMPRE se debe tener una variable que almacene cuantos elementos contiene el arreglo.
- Un arreglo se puede declarar e inicializar al mismo tiempo, con lo cual el total de valores asignados determina la capacidad máxima del arreglo.
- En Java la capacidad máxima de almacenamiento de un arreglo se obtiene con **varArreglo.length**.

4.4 OPERACIONES CON ARREGLOS

Las estructuras de datos condicionan la manera en que se almacenan los datos, así como la forma en la cual, posteriormente, dichos datos se recuperan. En el caso de los arreglos, al ser una estructura lineal, los datos se almacenan linealmente, es decir, cada dato sólo tendrá "un vecino" de cada lado, con excepción del primero, que no tiene un predecesor, y del último, que no tiene un sucesor. En cuanto a la recuperación, se puede hacer de manera individual por medio del índice o de la estructura como un todo, por medio de su nombre. Es importante destacar que son muy pocas las operaciones que pueden hacerse sobre toda la estructura. A continuación se presentan las principales operaciones con sus características.

4.4.1 Lectura, impresión y asignación

Las dos primeras deben realizarse elemento por elemento. Es decir, no es posible leer o imprimir todos los componentes de un arreglo en una sola instrucción, sino que se requiere hacerlo individualmente utilizando el índice. Suponiendo que al leer valores se guardan en posiciones consecutivas del arreglo, y conociendo el total de elementos a leer, entonces resulta natural usar un ciclo *for* para ir leyendo y guardando en las casillas del arreglo los valores leídos. Observe el siguiente código en el cual se leen valores enteros y se almacenan en el arreglo *edades*, declarado más arriba. Si bien el arreglo está formado por 10 casillas, no es necesario leer o trabajar con todas ellas. En este ejemplo sólo se usan las primeras *totalEdades* casillas, siendo *totalEdades* <= 10. Es conveniente que si el valor de dicha variable se lee previamente, se valide de tal manera que se asegure que sea un valor mayor o igual a 1 (por lo menos hay un dato) y menor o igual a 10, que es el máximo del espacio reservado. Todo el código que se muestra en esta sección se encuentra en el programa PruebaArreglos.java.

```
for (i = 0; i < totalEdades; i++){
    System.out.print("\nIngrese edad: ");
    edades[i] = lee.nextInt();
}
```

Para imprimir el contenido de un arreglo, y suponiendo que tiene información previamente asignada/leída ocupando posiciones consecutivas, se utiliza un ciclo *for* –similar al anterior– para ir recorriendo cada casilla e imprimiendo su contenido.

```
System.out.print("\nEdades: ");
for (i = 0; i < totalEdades; i++)
    System.out.print(edades[i] + " ");
```

En el ejemplo anterior, luego de imprimir el título fuera del ciclo, se imprime el contenido de la casilla 0 seguido de un espacio; luego, el contenido de la casilla 1 seguido de un espacio, y así hasta el último elemento del arreglo.

En el caso de la asignación, se puede hacer referencia a: 1) asignar un valor a una casilla en particular, 2) asignar un cierto valor a las primeras *n* casillas, 3) asignar todo un arreglo a otra variable de tipo arreglo y 4) asignar valores a ciertas posiciones arbitrarias, aplicando algún criterio como, por ejemplo, posiciones pares o impares. Veamos cada caso trabajando sobre arreglos declarados más arriba.

```
// Se asigna una cadena sólo a la casilla de la posición 0
nombres[0] = "Juan Martínez";
```

En el ejemplo anterior la casilla correspondiente al índice 0 almacena la cadena asignada, mientras que las demás permanecen sin cambios.

```
// Se asigna la cadena vacía a todas las casillas del arreglo.
for (i = 0; i < MAX; i++)
    nombres[i] = "";
```

En este ejemplo, al usar la constante *MAX* como límite del ciclo *for*, se asigna una cadena vacía a cada una de las casillas del arreglo.

A continuación se presenta un ejemplo de asignación de un arreglo a otra variable de tipo arreglo. Observe que los dos arreglos son del mismo tipo, ambos de cadenas de caracteres. También es importante señalar que la variable a la cual se le asignará el arreglo, ya existente, no requiere ser instanciada, sólo declarada.

```
// La variable otrosNombres sólo se declara, no se instancia.
String otrosNombres[ ];
/* Se asigna el arreglo nombres a la variable otrosNombres. Ahora hay dos variables
 * que hacen referencia al mismo arreglo.
```

```
*/
otrosNombres = nombres;
// Se imprime el contenido del arreglo, el cual almacena cadenas vacías.
System.out.println("\n\nContenido del arreglo de cadenas de caracteres:");
for (i = 0; i < MAX; i++)
    System.out.print(otrosNombres[i] + " - ");
```

Por último, presentamos un ejemplo donde se asigna el resultado de la expresión *2i*, con i = 0, 2, 4 ..., hasta MAX-1 en las casillas correspondientes a los índices pares.

```
// Se asigna el resultado de la expresión sólo en las casillas con índice par.
for (i = 0; i < MAX; i = i + 2)
    números[i] = Math.pow(2, i);
```

Una vez ejecutado el ciclo anterior (ver archivo PruebaArreglos.java), el arreglo queda como se muestra en la figura 4.2.

1.0	0.0	4.0	0.0	16.0	0.0	64.0	0.0	256.0	00
0	1	2	3	4	5	6	7	8	9

Figura 4.2 Estado del arreglo luego de la asignación de *2i* a los índices pares

En este ejemplo los demás componentes del arreglo quedan sin cambios, es decir, con el valor 0.0, que es el asignado por Java como valor por omisión a las variables tipo *double*.

4.4.2 Búsqueda de un elemento en un arreglo

La operación de **búsqueda** es muy importante porque permite recuperar un dato previamente almacenado. Para llevar a cabo esta operación se supone que los elementos ocupan posiciones consecutivas, sin dejar espacios vacíos en posiciones intermedias. En caso contrario, los métodos que se presentan a continuación deberán adaptarse.

El método más simple para buscar un elemento consiste en empezar desde la primera (o la última) casilla e ir comparando hasta que se encuentra el valor buscado o hasta que se llega a la última (o a la primera). Esta búsqueda se llama **secuencial** y puede aplicarse haya o no orden entre los valores almacenados en el arreglo. Si se conoce el contenido y se sabe que los elementos están ordenados (creciente o decrecientemente), se puede modificar el algoritmo para ganar cierta eficiencia.

Considerando que la búsqueda es una operación que frecuentemente se usa en la solución de distintos problemas, conviene programar el algoritmo encapsulándolo en un subprograma que pueda ser reusado las veces que se necesite. A continuación se presenta el subprograma que implementa el algoritmo de búsqueda secuencial en un arreglo desordenado de números enteros.

```
/* Método que implementa el algoritmo de búsqueda secuencial en arreglo desordenado.
 * arre[ ] es el arreglo en el cual se busca
 * n es el total de elementos almacenados en el arreglo
 * aBuscar es el elemento a buscar
 * Regresa la posición en la que está aBuscar o -1 si no lo encuentra.
 */
public static int buscaSecuencialDesordenado(int arre[ ], int n, int aBuscar){
      int i, resp;

      i = 0; // Para buscar desde la posición 0.
      resp = -1;
      /* Se busca mientras haya elementos para comparar y mientras el elemento visitado
       * sea distinto del elemento buscado.
       */
      while (i < n && arre[i] != aBuscar)
            i++;  // Se avanza al siguiente elemento.

      /* Si esta condición es verdadera, entonces es la segunda condición del while la que
       * no se cumple, por lo tanto el elemento que está en la posición i es igual al
       * elemento buscado. Se guarda la posición en la variable resp.
       */
      if (i < n)
            resp = i;
      return resp;
}
```

El método recibe como parámetros el arreglo la cantidad de elementos que almacena y el elemento a buscar. Regresa como resultado la posición en la que lo encontró o un -1 en caso contrario. Es importante notar que el método también pudo declararse booleano, por lo que regresaría *true* si lo encuentra o *false* en otro caso. ¿Qué ventajas/desventajas tiene esta propuesta sobre la anterior? En ambas soluciones el usuario de la misma puede determinar si el elemento está o no en el arreglo. Sin embargo, con la primera también puede tener acceso a dicho elemento, ya que el método proporciona el índice donde está almacenado. Con la segunda esto no es posible. Por tanto, si se quisiera –una vez encontrado– modificar ese dato, habría que volver a buscarlo para conocer su posición.

Es importante destacar el orden de las condiciones de *while*. Siempre debe ir primero ($i < n$), ya que es la que garantiza que el índice no supere la del arreglo. Si no lo hiciera, se podría tener un error en tiempo de ejecución.

Si los datos guardados en el arreglo tuvieran un orden entre sí, por ejemplo, del más pequeño al más grande, se puede modificar el algoritmo para evitar hacer comparaciones innecesarias. Es decir, si al comparar se encuentra un valor mayor al elemento buscado, quiere decir que el dato no está en el arreglo (como están en orden creciente, a partir de esa posición sólo habrá elementos más grandes). La solución que aparece más abajo corresponde a un arreglo de enteros ordenados crecientemente. Si los mismos estuvieran ordenados decrecientemente (de mayor a menor) se requiere hacer pequeños cambios en las condiciones del ciclo *while* y de *if/else*. Quedan a cargo del lector dichos cambios.

```
/* Método que implementa el algoritmo de búsqueda secuencial en un arreglo ordenado crecientemente.
 * arre[ ] es el arreglo en el cual se busca
 * n es el total de elementos almacenados en el arreglo
 * aBuscar es el elemento a buscar
 * Regresa la posición en la que lo encuentra o el negativo de la posición en la que
 * debería estar, más 1, en caso contrario.
 */
public static int buscaSecuencialOrdenado(int arre[ ], int n, int aBuscar){
   int i, resp;

   i = 0; // Para buscar desde la posición 0.
   /* Se busca mientras haya elementos para comparar y mientras el elemento visitado
    * sea menor –el arreglo está ordenado crecientemente– que el elemento buscado.
    */
   while (i < n && arre[i] < aBuscar)
      i++;  // Se avanza al siguiente elemento.

   /* Si no se llegó al límite del arreglo y se encontró el elemento buscado, se asigna
    * su posición como resultado. En caso contrario, el resultado es la posición en la
    * que "debería" estar, sin alterar el orden existente.
    */
   if (i < n && arre[i] == aBuscar)
      resp = i;
   else
      resp = -(i + 1);
   return resp;
}
```

Observe que al salir del ciclo *while* se evalúan dos condiciones. La primera de ellas es para asegurar que la *i* todavía hace referencia a una posición dentro del arreglo. Este caso se presenta cuando se encuentra el

elemento buscado o cuando se llega a un valor mayor que el elemento buscado. Por tanto, se requiere la segunda condición para comprobar si se encontró el elemento. La instrucción correspondiente a la claúsula *else* se ejecuta cuando *i* llega al límite —quiere decir que el elemento buscado es más grande que el último elemento del arreglo– o si se llega a un valor mayor —el elemento no está y no estará a partir de esa posición—.

En el caso de éxito de la búsqueda se regresa la posición del elemento como resultado. Si no se encuentra, regresa como resultado el negativo de la posición en la que debería estar, más 1. Si sólo se busca al elemento bastaría con regresar un -1 (como en el caso de la búsqueda en arreglos desordenados); sin embargo, si la búsqueda se hace como una operación auxiliar previa a agregar un elemento al arreglo, para asegurar que dicho elemento no se repita, entonces es mucho más útil regresar la posición que le corresponde para posteriormente hacer la inserción. Para distinguir este caso del arreglo de éxito, se usa el negativo y, como la posición podría ser 0, se le suma 1. El usuario de este método deberá pasarlo a positivo y restarle 1 para obtener la posición exacta a la que debe asignar el valor. Sobre la operación de inserción se volverá más adelante en este capítulo.

Los métodos definidos anteriormente funcionan para arreglos de enteros. Si los mismos almacenaran otro tipo de datos, se requerirá hacer algunos cambios para adaptarlos. Si fueran datos tipo *double* o *char*, en lugar de *int*, sólo bastará cambiar el tipo en la firma del método. En cambio, si los datos fueran de alguna clase (por ejemplo, la clase *String* o alguna definida por el usuario), entonces, además de los tipos en la firma, se debe cambiar los operadores == y < por los métodos *equals()* y *compareTo()*, respectivamente, para realizar las comparaciones. El lector podrá cuestionar la inconveniencia de tener tantas versiones de un mismo método. La respuesta a esto se verá más adelante en el próximo capítulo cuando se estudien los arreglos de objetos y, en particular, los arreglos genéricos.

En las dos versiones del método presentado se encuentra la primera ocurrencia del elemento buscado. Es decir, si el arreglo –ordenado o no– tuviera elementos repetidos, la posición es la del primero de ellos. Ambos métodos deberían modificarse para encontrar la ocurrencia *n* del dato buscado. Estas posibles variantes quedan a cargo del lector.

4.4.3 Inserción de elementos en un arreglo

Esta operación permite agregar nuevos elementos a un arreglo existente. Es importante señalar que el arreglo no puede aumentar su tamaño, por tanto sólo es posible agregar un nuevo valor si aún no está lleno.

Otro aspecto a tener en cuenta es si el problema que se está resolviendo con el uso de arreglos permite o no almacenar elementos repetidos. Existen aplicaciones en que es muy probable (casi seguro) que se tendrán valores repetidos, por ejemplo, las calificaciones de un grupo de alumnos o las edades de un grupo de pacientes. Por otra parte, existen otras en las que seguramente no se permitirán datos repetidos, por ejemplo, las claves de empleados de cierta empresa o los números de cuentas bancarias de un grupo de clientes. Por tanto, cuando se diseña un algoritmo para insertar elementos en un arreglo se debe disponer de información sobre este punto.

Por último, es necesario conocer si el arreglo está o no ordenado. Si está ordenado, al insertar un nuevo dato no se deberá alterar el orden existente. En cambio, si está desordenado, se puede aprovechar esa situación para facilitar la solución. A continuación se presentan algoritmos correspondientes a las alternativas planteadas. En estos métodos se usan arreglos de enteros; la generalización del código se verá más adelante en este mismo capítulo.

```
/* Método que inserta un dato en la primera casilla disponible de un arreglo desordenado
 * n representa la cantidad de elementos almacenados en el arreglo.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio.
 */
public static int insertaDesordenadoConRepetidos(int arre[ ], int n, int aInsertar){
  if (n < arre.length){ // Hay espacio en el arreglo
    arre[n] = aInsertar;  // n es la primera posición disponible.
    n = n + 1; // El arreglo tiene un elemento más.
  }
  return n; // El resultado es el nuevo tamaño.
}
```

El código presentado arriba permite insertar un nuevo elemento en un arreglo desordenado, en el cual puede haber elementos repetidos. La única condición que se impone para llevar a cabo la operación es que el arreglo tenga espacio disponible. Esta condición se evalúa comparando *n* con el *length* del arreglo. Si hay espacio, se asigna el dato en la primera casilla disponible, que es *n* (recuerde que los valores se guardan desde la posición 0 hasta la n-1), y se incrementa *n* (el total de elementos aumentó en 1).

En Java, cuando se pasa un arreglo como parámetro de un método, se pasa su referencia. Por lo tanto, cualquier cambio que se le haga perdura en el arreglo. Por el contrario, los cambios que se hacen a datos primitivos no se mantienen una vez ejecutado el subprograma. Es por esta razón que la *n*, si se quiere mantener su nuevo valor, se debe regresar como resultado del método.

El algoritmo anterior debe modificarse para adaptarse a inserciones en arreglos desordenados, en los cuales no se puede almacenar elementos repetidos. Observe el siguiente código:

```
/* Método que inserta un dato en la primera casilla disponible de un arreglo desordenado
 * n representa la cantidad de elementos almacenados en el arreglo.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio y si dicho
 * valor no se encuentra en el arreglo.
 */
public static int insertaDesordenadoSinRepetidos(int arre[ ], int n, int aInsertar){
  if (n < arre.length) // Hay espacio en el arreglo
    if (buscaSecuencialDesordenado(arre, n, aInsertar) == -1){
    // No está en el arreglo
      arre[n] = aInsertar;  // n es la primera posición disponible.
      n = n + 1; // El arreglo tiene un elemento más.
    }
  return n; // El resultado es el nuevo tamaño.
}
```

En el caso de los arreglos ordenados, igual que en los desordenados, lo primero que debe verificarse es que haya espacio suficiente. Una vez comprobado esto, independientemente de si se permite o no elementos repetidos, es necesario encontrar la posición en que se debe insertar el nuevo dato para no alterar el orden existente. Para ello se usa el método de búsqueda secuencial en arreglos ordenados.

Posteriormente, si procede la inserción, lo cual sí depende de si puede o no haber elementos repetidos, se debe recorrer una posición hacia la derecha a todos los valores, desde el último hasta el que actualmente ocupa la posición en que se debe hacer la inserción. Así se define un método auxiliar que lleva a cabo el desplazamiento para modularizar la solución y simplificarla por medio del reuso. A continuación se presenta el código:

```
/* Método auxiliar que desplaza todos los elementos del arreglo una posición a la
 * derecha, sin alterar el contenido de las casillas.
 */
public static void desplazaHaciaDerecha (int arre[ ], int n, int pos){
   int i;
   for (i = n; i > pos; i--)
      arre[i] = arre[i-1];
}
```

En la figura 4.3 se muestra el efecto del movimiento descrito. El elemento a insertar es 48, por lo cual la posición que le corresponde es la 2, en el arreglo dado (*a*). Siendo $n = 5$, el ciclo se ejecuta desde 5 hasta 3, asignando a la casilla 5 el contenido de la casilla 4, a la 4 el contenido de la 3 y, por último, a la 3 el de la 2, quedando como se muestra en (*b*).

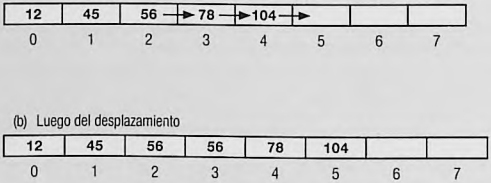


Figura 4.3 Desplazamiento de los elementos del arreglo hacia la derecha

Una vez hecho el desplazamiento se asigna el nuevo dato en la posición correspondiente, según el orden, y se incrementa el total de componentes del arreglo. El siguiente código implementa el algoritmo de inserción en arreglos ordenados, en los cuales se pueden repetir elementos.

```
/* Método que inserta un dato en un arreglo ordenado
 * n representa la cantidad de elementos almacenados en el arreglo.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio.
 */
public static int insertaOrdenadoConRepetidos(int arre[ ], int n, int aInsertar){
   if (n < arre.length){ // Hay espacio en el arreglo
      int pos = buscaSecuencialOrdenado(arre, n, aInsertar);
      if (pos < 0)  // Si no está, obtiene la posición en la que debe asignarlo.
         pos = -pos -1;
      desplazaHaciaDerecha (arre, n, pos);
      arre[pos] = aInsertar;
      n = n + 1; // El arreglo tiene un elemento más.
   }
   return n; // El resultado es el nuevo tamaño.
}
```

Observe que si el elemento no está en el arreglo (*pos < 0*), antes de hacer el desplazamiento se debe operar la variable *pos* para generar la posición en que se debe insertar el nuevo valor para no alterar el orden. Primero se la convierte a positiva y luego se le resta 1 (se sugiere volver a revisar el método de búsqueda secuencial en arreglos ordenados).

Retomando el ejemplo anterior, luego de desplazar los elementos hacia la derecha, se asigna en la casilla correspondiente a *pos* el nuevo dato, en este caso el 48. El arreglo queda como se muestra en la figura 4.4.

12	45	48	56	78	104		
0	1	2	3	4	5	6	7

Figura 4.4 Arreglo luego de insertar el 48

Si la naturaleza del problema no permite que se repitan elementos, hay que hacer un pequeño ajuste al método anterior, quedando como se muestra a continuación:

```
/* Método que inserta un dato en un arreglo ordenado
 * n representa la cantidad de elementos almacenados en el arreglo.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio y si dicho
 * valor no se encuentra en el arreglo.
```

```
/* Método que inserta un dato en un arreglo ordenado
 * n representa la cantidad de elementos almacenados en el arreglo.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio.
 */
public static int insertaOrdenadoConRepetidos(int arre[ ], int n, int aInsertar){
   if (n < arre.length){ // Hay espacio en el arreglo
      int pos = buscaSecuencialOrdenado(arre, n, aInsertar);
      if (pos < 0)  // Si no está, obtiene la posición en la que debe asignarlo.
         pos = -pos -1;
      desplazaHaciaDerecha (arre, n, pos);
      arre[pos] = aInsertar;
      n = n + 1; // El arreglo tiene un elemento más.
   }
   return n; // El resultado es el nuevo tamaño.
}
```

Observe que si el elemento no está en el arreglo (*pos* < 0), antes de hacer el desplazamiento se debe operar la variable *pos* para generar la posición en que se debe insertar el nuevo valor para no alterar el orden. Primero se la convierte a positiva y luego se le resta 1 (se sugiere volver a revisar el método de búsqueda secuencial en arreglos ordenados).

Retomando el ejemplo anterior, luego de desplazar los elementos hacia la derecha, se asigna en la casilla correspondiente a *pos* el nuevo dato, en este caso el 48. El arreglo queda como se muestra en la figura 4.4.

12	45	48	56	78	104		
0	1	2	3	4	5	6	7

Figura 4.4 Arreglo luego de insertar el 48

Si la naturaleza del problema no permite que se repitan elementos, hay que hacer un pequeño ajuste al método anterior, quedando como se muestra a continuación:

```
/* Método que inserta un dato en un arreglo ordenado
 * n representa la cantidad de elementos almacenados en el arreglo.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio y si dicho
 * valor no se encuentra en el arreglo.
```

```
*/
public static int insertaOrdenadoSinRepetidos(int arre[ ], int n, int aInsertar){
  if (n < arre.length){ // Hay espacio en el arreglo
    int pos = buscaSecuencialOrdenado(arre, n, aInsertar);
    if (pos < 0){  // Si no está, obtiene la posición en la que debe asignarlo.
      pos = -pos -1;
      desplazaHaciaDerecha (arre, n, pos);
      arre[pos] = aInsertar;
      n = n + 1; // El arreglo tiene un elemento más.
  }
}
  return n; // El resultado es el nuevo tamaño.
}
```

La diferencia con respecto al ejemplo anterior es que en este último se hace la inserción (recorrimiento, asignación e incremento) sólo si el elemento no fue encontrado (*pos < 0*).

Muy importante

- ☑ Verificar si hay espacio.
- ☑ Si el arreglo no está ordenado, se asigna en la primera casilla disponible.
- ☑ Si el arreglo está ordenado:
  1. Encontrar la posición en la que debe insertarse (búsqueda).
  2. Desplazar hacia la derecha.
  3. Asignar en la casilla que le corresponde según el orden.
- ☑ Incrementar el total de elementos.
- ☑ Los cambios hechos al arreglo en el método permanecen, ya que se recibe su referencia.
- ☑ Si no se pueden repetir valores, SIEMPRE se deben buscar (estén o no ordenados).

4.4.4 Eliminación de elementos en el arreglo

Eliminar un elemento de un arreglo es muy útil, ya que en muchos problemas uno (o más) de sus datos puede dejar de ser parte del problema y, por tanto, ya no debe almacenarse junto con los demás. El manejo de datos generalmente es dinámico, es decir, hay que agregar nuevos valores, eliminar o modificar alguno o varios de los existentes.

Cuando se quiere eliminar un elemento de un arreglo, lo primero que debe verificarse es que dicho elemento esté guardado en el arreglo. Posteriormente se debe "quitar", para lo cual se lo sustituye con otro elemento del arreglo, decrementando el tamaño del mismo. En realidad, no hay manera de eliminar físicamente el dato; por

tanto, lo que se hace es una eliminación lógica. Es decir, se hacen los cambios necesarios para que ese valor ya no esté en el arreglo y se reduce el total de elementos en 1.

Si el arreglo está desordenado, en la posición del elemento a eliminar se asigna el valor almacenado en la última casilla y se reduce el total de elementos. A pesar de que el dato (de la última casilla) sigue estando ahí, al decrementar el total se ignora. Es decir, en cualquier operación que se realice posteriormente sobre el arreglo, el nuevo valor del total de elementos (*n*) es el que determina cuántos y cuáles datos se procesarán.

```
/* Método que elimina un elemento de un arreglo desordenado.
 * arre es el arreglo del cual se quitará un elemento.
 * n es el total de elementos.
 * aEliminar es el dato que se quiere eliminar.
 * Regresa como resultado el total de elementos modificado (o no).
 */
public static int eliminaDesordenado(int arre[ ], int n, int aEliminar){
   int pos;

   pos = buscaSecuencialDesordenado(arre, n, aEliminar);
   if (pos >= 0){ // El dato está en el arreglo
      arre[pos] = arre[n-1]; // Se reemplaza por el contenido de la última casilla
      n = n - 1; // Disminuye en 1 el total de elementos
   }
   return n;
}
```

En la figura 4.5 se presenta un arreglo desordenado de 6 elementos en el cual se quiere eliminar el 1 en (*a*). Luego de que se ha aplicado el método anterior, el arreglo queda como se muestra en (*b*).

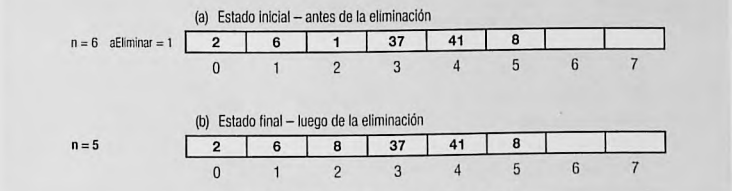


Figura 4.5 Eliminación en arreglos desordenados

Para arreglos ordenados el algoritmo anterior no funciona, ya que se alteraría el orden de los elementos. Por tanto, para que esto no suceda, se requiere desplazar todos los elementos una posición hacia la izquierda, desde el que está a la derecha del elemento a eliminar hasta el último. Observe el siguiente ejemplo en el cual se tiene un arreglo ordenado de 6 elementos (*n* = 6), el cual quiere eliminar el valor 78, figura 4.6 (*a*).

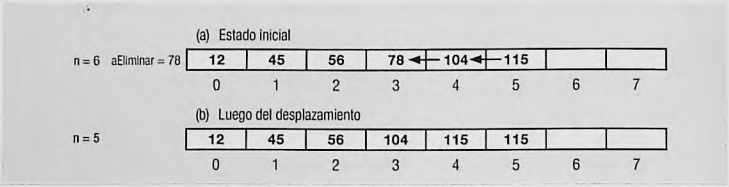


Figura 4.6 Desplazamiento hacia la izquierda

El código correspondiente al desplazamiento hacia la izquierda se muestra a continuación. El resultado de aplicar el método al arreglo de la figura 4.6 se puede observar en (*b*).

```
/* Método auxiliar que desplaza todos los elementos del arreglo una posición a la
 * izquierda, sin alterar el contenido de las casillas.
 */
public static void desplazaHaciaIzquierda (int arre[], int n, int pos){
    int i;
    for (i = pos; i < n - 1; i++)
        arre[i] = arre[i+1];
}
```

Una vez verificado que el elemento esté en el arreglo y hecho el desplazamiento, se debe decrementar el total, como en el caso anterior. Analice el siguiente método:

```
/* Método que elimina un elemento de un arreglo ordenado.
 * arre es el arreglo del cual se quitará un elemento.
 * n es el total de elementos.
 * aEliminar es el dato que se quiere eliminar.
 * Regresa como resultado el total de elementos modificado (o no).
 */
```

```
public static int eliminaOrdenado(int arre[ ], int n, int aEliminar){
    int pos;

    pos = buscaSecuencialOrdenado(arre, n, aEliminar);
    if (pos >= 0){ // El dato está en el arreglo
       desplazaHaciaIzquierda(arre, n, pos);
       n = n - 1; // Disminuye en 1 el total de elementos
    }
  return n;
}
```


Los dos métodos de eliminación se basan en el resultado obtenido por los métodos de búsqueda, por lo que en ambos casos se elimina la primera ocurrencia del elemento. Es decir, si el arreglo –ordenado o no– tuviera elementos repetidos, la posición obtenida por la búsqueda es la del primero de ellos, por lo cual será el que se elimine del arreglo. Ambos métodos deberían modificarse para eliminar la ocurrencia *n*-ésima del dato. Estas posibles variantes quedan a cargo del lector.

Muy importante

- ☑ Verificar que el elemento esté en el arreglo (búsqueda).
- ☑ Si el arreglo no está ordenado, se reemplaza por el dato que ocupa la última casilla.
- ☑ Si el arreglo está ordenado:
  - Desplazar hacia la izquierda.
- ☑ Decrementar el total de elementos.
- ☑ Los cambios hechos al arreglo en el método permanecen, ya que se recibe su referencia.

En el programa 4.1, OAE.java, se define una clase que agrupa todos los métodos estudiados previamente. Asimismo, el programa 4.2 (UsaOAE.java) prueba los métodos de la clase OAE con un arreglo desordenado y con uno ordenado. Se sugiere al lector que revise el material y haga pruebas para reforzar los conocimientos adquiridos. En todos los casos, los métodos se aplican a arreglos de enteros. Para generalizar, se puede sobrecargar los métodos de tal manera que se tenga un método de inserción para cada uno de los tipos que se quieran manejar; lo mismo para cada uno de los métodos estudiados. Otra alternativa de solución es generar clases diferentes para cada uno de los tipos de datos. Queda a cargo del lector analizar e implementar estas variantes. En la siguiente sección se presenta una manera más general para manejar arreglos de distintos tipos.

Programas 4.1 y 4.2 | **OAE.java (Operaciones para Arreglos de Enteros) y UsaOAE.java**

```
package cap4;
/**
 * @Silvia Guardati
 * Programa 4.1
 * Definición de una clase que concentra todos los métodos estudiados para el manejo
 * de arreglos. En este caso es para arreglos de enteros.
 * OAE: Operaciones para Arreglos de Enteros.
 */
public class OAE {

/* Método que implementa el algoritmo de búsqueda secuencial en arreglo desordenado.
 * arre[ ] es el arreglo en el cual se busca
 * n es el total de elementos almacenados en el arreglo
 * aBuscar es el elemento a buscar
 * Regresa la posición en la que está aBuscar o -1 si no lo encuentra.
 */
public static int buscaSecuencialDesordenado(int arre[ ], int n, int aBuscar){
    int i, resp;

    i = 0; // Para buscar desde la posición 0.
    resp = -1;
    /* Se busca mientras haya elementos para comparar y mientras el elemento
     * visitado sea distinto del elemento buscado.
     */
    while (i < n && arre[i] != aBuscar)
        i++;  // Se avanza al siguiente elemento.

    /* Si esta condición es verdadera, entonces es la segunda condición del
     * while la que no se cumple, por lo tanto el elemento que está en la
     * posición i es igual al elemento buscado. Se guarda la posición en la
     * variable resp.
     */
    if (i < n)
        resp = i;
```

```
        return resp;
    }

    /* Método que implementa el algoritmo de búsqueda secuencial en un arreglo ordenado.
     * arre[ ] es el arreglo en el cual se busca
     * n es el total de elementos almacenados en el arreglo
     * aBuscar es el elemento a buscar
     * Regresa la posición en la que lo encuentra o el negativo de la posición en la
     * que debería estar, más 1, en caso contrario.
     */
        public static int buscaSecuencialOrdenado(int arre[ ], int n, int aBuscar){
        int i, resp;

        i = 0; // Para buscar desde la posición 0.
        /* Se busca mientras haya elementos para comparar y mientras el elemento
         * visitado sea menor -el arreglo está ordenado crecientemente- que el
         * elemento buscado.
         */
        while (i < n && arre[i] < aBuscar)
            i++;  // Se avanza al siguiente elemento.

        /* Si no se llegó al límite del arreglo y se encontró el elemento buscado,
         * se asigna su posición como resultado. En caso contrario, el resultado es
         * la posición en la que "debería" estar, sin alterar el orden existente.
         */
        if (i < n && arre[i] == aBuscar)
            resp = i;
        else
            resp = -(i + 1);
        return resp;
    }

    /* Método que inserta un dato en la primera casilla disponible de un arreglo desordenado.
     * n representa la cantidad de elementos almacenados en el arreglo.
     * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio.
     */
```

```
    public static int insertaDesordenadoConRepetidos(int arre[ ], int n, int aInsertar){
    if (n < arre.length){ // Hay espacio en el arreglo
      arre[n] = aInsertar;  // n es la primera posición disponible.
      n = n + 1; // El arreglo tiene un elemento más.
    }
    return n; // El resultado es el nuevo tamaño.
}

/* Método que inserta un dato en la primera casilla disponible de un arreglo desordenado
 * n representa la cantidad de elementos almacenados en el arreglo.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio y si dicho
 * valor no se encuentra en el arreglo.
 */
public static int insertaDesordenadoSinRepetidos(int arre[ ], int n, int aInsertar){
    if (n < arre.length) // Hay espacio en el arreglo
      if (buscaSecuencialDesordenado(arre, n, aInsertar) == -1){
        // No está en el arreglo
        arre[n] = aInsertar;  // n es la primera posición disponible.
        n = n + 1; // El arreglo tiene un elemento más.
}
    return n; // El resultado es el nuevo tamaño.
}

/* Método auxiliar que desplaza todos los elementos del arreglo una posición a la
 * derecha, sin alterar el contenido de las casillas.
 */
public static void desplazaHaciaDerecha (int arre[ ], int n, int pos){
    nt i;
    for (i = n; i > pos; i--)
      arre[i] = arre[i-1];
}

/* Método que inserta un dato en un arreglo ordenado
* n representa la cantidad de elementos almacenados en el arreglo.
* aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio.
*/
```

```
    public static int insertaOrdenadoConRepetidos(int arre[ ], int n, int aInsertar){
      if (n < arre.length){ // Hay espacio en el arreglo
        int pos = buscaSecuencialOrdenado(arre, n, aInsertar);
        if (pos < 0)  // Si no está obtiene la posición en la que debe asignarlo.
           pos = -pos -1;
        desplazaHaciaDerecha (arre, n, pos);
        arre[pos] = aInsertar;
        n = n + 1; // El arreglo tiene un elemento más.
        }
      return n; // El resultado es el nuevo tamaño.
    }

  /* Método que inserta un dato en un arreglo ordenado
   * n representa la cantidad de elementos almacenados en el arreglo.
   * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio y si dicho
   * valor no se encuentra en el arreglo.
   */
   public static int insertaOrdenadoSinRepetidos(int arre[ ], int n, int aInsertar){
     if (n < arre.length){ // Hay espacio en el arreglo
        int pos = buscaSecuencialOrdenado(arre, n, aInsertar);
        if (pos < 0){  // Si no está obtiene la posición en la que debe asignarlo.
           pos = -pos -1;
           desplazaHaciaDerecha (arre, n, pos);
           arre[pos] = aInsertar;
           n = n + 1; // El arreglo tiene un elemento más.
        }
      }
      return n; // El resultado es el nuevo tamaño.
      }

  /* Método que elimina un elemento de un arreglo desordenado.
   * arre es el arreglo del cual se quitará un elemento.
   * n es el total de elementos.
   * aEliminar es el dato que se quiere eliminar.
   * Regresa como resultado el total de elementos modificado (o no).
   */
```

```
public static int eliminaDesordenado(int arre[ ], int n, int aEliminar){
  int pos;

  pos = buscaSecuencialDesordenado(arre, n, aEliminar);
  if (pos >= 0){ // El dato está en el arreglo
    arre[pos] = arre[n-1]; // Se reemplaza por el contenido de la última casilla
    n = n - 1; // Disminuye en 1 el total de elementos
  }
  return n;
}

/* Método auxiliar que desplaza todos los elementos del arreglo una posición a la
 * izquierda, sin alterar el contenido de las casillas.
 */
public static void desplazaHaciaIzquierda (int arre[ ], int n, int pos){
  int i;
  for (i = pos; i < n - 1; i++)
    arre[i] = arre[i+1];
}

/* Método que elimina un elemento de un arreglo ordenado.
 * arre es el arreglo del cual se quitará un elemento.
 * n es el total de elementos.
 * aEliminar es el dato que se quiere eliminar.
 * Regresa como resultado el total de elementos modificado (o no).
 */
public static int eliminaOrdenado(int arre[ ], int n, int aEliminar){
  int pos;

  pos = buscaSecuencialOrdenado(arre, n, aEliminar);
  if (pos >= 0){ // El dato está en el arreglo
    desplazaHaciaIzquierda(arre, n, pos);
    n = n - 1; // Disminuye en 1 el total de elementos
  }
  return n;
}
```

```
/* Método que regresa una cadena formada con el contenido del arreglo que recibe
 * como parámetro. Es muy útil para conocer el contenido del arreglo.
 */
public static String imprime(int arre[ ], int n){
    int i;
    StringBuilder cad = new StringBuilder( );

    for (i = 0; i < n; i++)
      cad.append(arre[i] + " ");
    return cad.toString( );
}
}

package cap4;

/**
 * @author Silvia Guardati
 * Programa 4.2
 * Ejemplo de aplicación de los métodos de la clase OAE.
 */
public class UsaOAE {

  public static void main(String[ ] args) {
    int aOrdenado[ ] = new int [10];
    int aDesordenado[ ] = new int [10];

    aDesordenado[0] = 1;
    aDesordenado[1] = 8;
    aDesordenado[2] = 4;
    aDesordenado[3] = 2;

    aOrdenado[0] = 2;
    aOrdenado[1] = 6;
    aOrdenado[2] = 11;
    aOrdenado[3] = 23;
```

```
// Imprime estado inicial del arreglo desordenado.
System.out.println("\nArreglo desordenado:\n" + OAE.imprime(aDesordenado, 4));

// Imprime estado inicial del arreglo ordenado.
System.out.println("\nArreglo ordenado:\n" + OAE.imprime(aOrdenado, 4));

// Búsqueda secuencial en arreglos desordenados.
int índice;
índice = OAE.buscaSecuencialDesordenado(aDesordenado, 4, 8);
System.out.println("Debe imprimir 1: " + índice);
índice = OAE.buscaSecuencialDesordenado(aDesordenado, 4, 1);
System.out.println("Debe imprimir 0: " + índice);
índice = OAE.buscaSecuencialDesordenado(aDesordenado, 4, 2);
System.out.println("Debe imprimir 3: " + índice);
índice = OAE.buscaSecuencialDesordenado(aDesordenado, 4, 9);
System.out.println("Debe imprimir -1: " + índice);

// Búsqueda secuencial en arreglos ordenados.
índice = OAE.buscaSecuencialOrdenado(aOrdenado, 4, 2);
System.out.println("Debe imprimir 0: " + índice);
índice = OAE.buscaSecuencialOrdenado(aOrdenado, 4, 11);
System.out.println("Debe imprimir 2: " + índice);
índice = OAE.buscaSecuencialOrdenado(aOrdenado, 4, 15);
System.out.println("Debe imprimir -4: " + índice);
índice = OAE.buscaSecuencialOrdenado(aOrdenado, 4, 4);
System.out.println("Debe imprimir -2: " + índice);
índice = OAE.buscaSecuencialOrdenado(aOrdenado, 4, 0);
System.out.println("Debe imprimir -1: " + índice);

// Inserción en arreglos desordenados, permitiendo elementos repetidos.
int n = OAE.insertaDesordenadoConRepetidos(aDesordenado, 4, 4);
String resultado = OAE.imprime(aDesordenado, n);
System.out.println("\nInserta 4 en desordenado con repetidos: " + resultado);

// Inserción en arreglos desordenados, NO permitiendo elementos repetidos.
n = OAE.insertaDesordenadoSinRepetidos(aDesordenado, n, 1);
```

```
resultado = OAE.imprime(aDesordenado, n);
System.out.println("\nInserta 1 en desordenado sin repetidos (no pudo): " + resultado);

n = OAE.insertaDesordenadoSinRepetidos(aDesordenado, n, 5);
resultado = OAE.imprime(aDesordenado, n);
System.out.println("\nInserta 5 en desordenado sin repetidos: " + resultado);

// Inserción en arreglos ordenados, permitiendo elementos repetidos.
int m = OAE.insertaOrdenadoConRepetidos(aOrdenado, 4, 2);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nInserta 2 en ordenado con repetidos: " + resultado);

 // Inserción en arreglos ordenados, NO permitiendo elementos repetidos.
m = OAE.insertaOrdenadoSinRepetidos(aOrdenado, m, 6);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nInserta 6 en ordenado sin repetidos -no puede: " + resultado);

m = OAE.insertaOrdenadoSinRepetidos(aOrdenado, m, 8);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nInserta 8 en ordenado sin repetidos: " + resultado);

m = OAE.insertaOrdenadoSinRepetidos(aOrdenado, m, 1);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nInserta 1 en ordenado sin repetidos: " + resultado);

m = OAE.insertaOrdenadoSinRepetidos(aOrdenado, m, 30);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nInserta 30 en ordenado sin repetidos: " + resultado);

// Eliminación en arreglos desordenados.
n = OAE.eliminaDesordenado(aDesordenado, n, 1);
resultado = OAE.imprime(aDesordenado, n);
System.out.println("\nElimina 1 en desordenado: " + resultado);

n = OAE.eliminaDesordenado(aDesordenado, n, 1);
resultado = OAE.imprime(aDesordenado, n);
System.out.println("\nElimina 1 -no está- en desordenado: " + resultado);
```

```
// Eliminación en arreglos ordenados.
m = OAE.eliminaOrdenado(aOrdenado, m, 2);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nElimina primer ocurrencia de 2 en ordenado: " + resultado);

m = OAE.eliminaOrdenado(aOrdenado, m, 2);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nElimina segunda ocurrencia de 2 en ordenado: " + resultado);

m = OAE.eliminaOrdenado(aOrdenado, m, 7);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nElimina 7 -no está- en ordenado: " + resultado);

m = OAE.eliminaOrdenado(aOrdenado, m, 23);
resultado = OAE.imprime(aOrdenado, m);
System.out.println("\nElimina 23 en ordenado: " + resultado);
}
}
```

En la sección 4.6 de este capítulo se presentan algunos ejemplos de aplicación de arreglos, en los cuales se retoma el OAE.

4.4.5 Otras operaciones

Además de las operaciones ya estudiadas, existen otras que por su utilidad merecen una atención especial. Son operaciones muy frecuentes cuando se trabaja con datos almacenados en arreglos y, por lo tanto, es conveniente implementarlas como métodos auxiliares que puedan reusarse todas las veces que se necesite.

Las operaciones propuestas son: sumar los valores almacenados en el arreglo, calcular su promedio, encontrar la posición del valor más grande, la del más pequeño, cuántos valores son más grandes que un cierto dato dado y otras similares. Estas operaciones pueden tener muchas aplicaciones en la solución de problemas. Por ejemplo, el promedio puede servir para calcular el promedio de edades de un grupo de personas o el promedio de hijos que tienen las familias de una cierta población. A su vez, la posición del mayor puede determinar el mes en el cual se tuvo el mayor ingreso de turistas (el arreglo almacena el total de turistas que ingresan por mes) o el día de la semana en que más niños faltaron a cierto colegio (el arreglo almacena inasistencias de niños a la escuela, por día). Los métodos se incluyeron en la clase OAE, programa 4.1. A su vez, en la clase UsaOAE, programa 4.2, se muestra el uso de los mismos. La versión completa de estos dos programas está en el paquete cap4, del proyecto EstructurasDatosBásicas de Netbeans que complementa a este libro.

4.5 OPERACIONES CON ARREGLOS GENÉRICOS

En la sección previa se estudiaron las principales operaciones con arreglos, y todas ellas se aplicaron sobre arreglos de enteros. Si el problema requiere de arreglos de cadenas, de doble precisión o de cualquier otro tipo, se debería repetir el código reemplazando el tipo por aquel que fuera necesario. Para evitar reescribir el código, creando distintas versiones según el tipo de dato que se almacene, es conveniente definir métodos que manejen datos genéricos. De esta forma, los métodos se generalizan a cualquier tipo de información. A continuación se presenta y explica el código correspondiente a algunos de los métodos; la versión completa se encuentra en el programa 4.3, OAG.java. Observe que en la firma de cada método, junto a *static* se agrega *<T>* para indicar que dicho método maneja datos tipo T. A su vez, si se requiere usar el método *compareTo()* asociado a los elementos del arreglo (porque son del tipo T, por lo tanto se comparan con el *equals()* o el *compareTo*()), se debe poner *<T extends Comparable<T>>*.

A continuación se presenta el método de búsqueda secuencial en arreglos desordenados. Note que se modificó la firma del método, agregando *<T>* y cambiando el tipo del arreglo y del elemento a buscar, y también la compración, ya que ahora se usa el método *equals()*.

```
/* Método que implementa el algoritmo de búsqueda secuencial en un arreglo genérico
 * desordenado. Regresa la posición en la que está aBuscar o -1 si no lo encuentra.
 */
 public static <T> int buscaSecuencialDesordenado(T arre[ ], int n, T aBuscar){
    int i, resp;

    i = 0; // Para buscar desde la posición 0.
    resp = -1;
      /* Se busca mientras haya elementos para comparar y mientras el elemento visitado
       * sea distinto del elemento buscado.
       */
    while (i < n && !arre[i].equals(aBuscar)) // Se compara con el equals( )
      i++;  // Se avanza al siguiente elemento.

    /* Si esta condición es verdadera, entonces es la segunda condición del while la que
     * no se cumple, por lo tanto el elemento que está en la posición i es igual al
     * elemento buscado. Se guarda la posición en la variable resp.
     */
    if (i < n)
      resp = i;
    return resp;
 }
```

El siguiente método corresponde a la búsqueda secuencial en arreglos ordenados. Como en el caso anterior, se modificó la firma y la manera de comparar los elementos.

```
/* Método que implementa el algoritmo de búsqueda secuencial en un arreglo
 * genérico ordenado.
 * Regresa la posición en la que lo encuentra o el negativo de la posición en la que
 * debería estar más 1, en caso contrario.
 */
 public static <T extends Comparable<T>> int buscaSecuencialOrdenado(T arre[], int n,
 T aBuscar){
   int i, resp;
   i = 0; // Para buscar desde la posición 0.
   resp = -1;
   /* Se busca mientras haya elementos para comparar y mientras el elemento visitado
    * sea menor -el arreglo está ordenado crecientemente- que el elemento buscado.
    */
    while (i < n && arre[i].compareTo(aBuscar) < 0)
      i++; // Se avanza al siguiente elemento.

   /* Si no se llegó al límite del arreglo y se encontró el elemento buscado, se asigna
    * su posición como resultado. En caso contrario, el resultado es la posición en la
    * que "debería" estar, sin alterar el orden existente.
    */
    if (i < n && arre[i].equals(aBuscar))
      resp = i;
    else
      resp = -(i + 1);
    return resp;

}
```

En el programa 4.3, OAG.java, se incluyeron todos los métodos adaptados para trabajar con arreglos genéricos. El programa 4.4, UsaOAG.java tiene un método *main* para probar el funcionamiento de los mismos. Se declararon dos arreglos: uno, que almacena cadenas de caracteres, y otro, números de doble precisión. Con ambos se usan los mismos métodos para buscar, insertar o eliminar valores, lo que destaca el potencial de los arreglos genéricos. Se sugiere al lector que analice el código y haga pruebas para reforzar los temas estudiados.

Programa 4.3 **OAG.java (Operaciones para Arreglos Genéricos)**

```
package cap4;

/**
 * @Silvia Guardati
 * Programa 4.3
 * Esta clase agrupa los principales métodos para arreglos genéricos.
 * OAG: Operaciones para Arreglos Genéricos.
 */
public class OAG {

 /* Método que implementa el algoritmo de búsqueda secuencial en un arreglo genérico
  * desordenado. Regresa la posición en la que está aBuscar o -1 si no lo encuentra.
  */
 public static <T> int buscaSecuencialDesordenado(T arre[ ], int n, T aBuscar){
     int i, resp;

     i = 0; // Para buscar desde la posición 0.
     resp = -1;
     /* Se busca mientras haya elementos para comparar y mientras el elemento
      * visitado sea distinto del elemento buscado.
      */
     while (i < n && !arre[i].equals(aBuscar)) // Se compara con el equals( )
        i++;  // Se avanza al siguiente elemento.

     /* Si esta condición es verdadera, entonces es la segunda condición del
      * while la que no se cumple, por lo tanto el elemento que está en la
      * posición es igual al elemento buscado. Se guarda la posición i en la
      * variable resp.
      */
     if (i < n)
        resp = i;
     return resp;
 }
```

```
/* Método que implementa el algoritmo de búsqueda secuencial en un arreglo
 * genérico ordenado.
 * Regresa la posición en la que lo encuentra o el negativo de la posición en la
 * que debería estar, más 1, en caso contrario.
 */
 public static <T extends Comparable<T>> int buscaSecuencialOrdenado(T arre[ ], int n,
 T aBuscar){
   int i, resp;

   i = 0; // Para buscar desde la posición 0.
   resp = -1;
   /* Se busca mientras haya elementos para comparar y mientras el elemento visitado
    * sea menor -el arreglo está ordenado crecientemente- que el elemento buscado.
    */
   while (i < n && arre[i].compareTo(aBuscar) < 0)
     i++;  // Se avanza al siguiente elemento.

   /* Si no se llegó al límite del arreglo y se encontró el elemento buscado, se
    * asigna su posición como resultado. En caso contrario, el resultado es la
    * posición en la que "debería" estar, sin alterar el orden existente.
    */
   if (i < n && arre[i].equals(aBuscar))
     resp = i;
   else
     resp = -(i + 1);
   return resp;
}

/* Método que inserta un dato en la primera casilla disponible de un arreglo genérico
 * desordenado, permitiendo repetir datos.
 */
   public static <T> int insertaDesordenadoConRepetidos(T arre[ ], int n, T aInsertar){
      if (n < arre.length){ // Hay espacio en el arreglo
        arre[n] = aInsertar;  // n es la primera posición disponible.
        n = n + 1; // El arreglo tiene un elemento más.
      }
```

```
    return n; // El resultado es el nuevo tamaño.
  }

 /* Método que inserta un dato en la primera casilla disponible de un arreglo genérico
  * desordenado, sin permitir repeticiones.
  */
 public static <T> int insertaDesordenadoSinRepetidos(T arre[ ], int n, T aInsertar){
    if (n < arre.length) // Hay espacio en el arreglo
      if (buscaSecuencialDesordenado(arre, n, aInsertar) == -1){// No está en el arreglo
          arre[n] = aInsertar;  // n es la primera posición disponible.
          n = n + 1; // El arreglo tiene un elemento más.
  }
    return n; // El resultado es el nuevo tamaño.
  }

 /* Método auxiliar que desplaza todos los elementos del arreglo una posición a la
  * derecha, sin alterar el contenido de las casillas.
  */
  public static <T> void desplazaHaciaDerecha (T arre[ ], int n, int pos){
     int i;
     for (i = n; i > pos; i--)
        arre[i] = arre[i-1];
  }

 /* Método que inserta un dato en un arreglo genérico ordenado, permitiendo repetir
  * datos.
  */
   public static <T extends Comparable<T>> int insertaOrdenadoConRepetidos(T arre[ ],
   int n, T aInsertar){
     if (n < arre.length){ // Hay espacio en el arreglo
        int pos = buscaSecuencialOrdenado(arre, n, aInsertar);
        if (pos < 0)  // Si no está obtiene la posición en la que debe asignarlo.
           pos = -pos -1;
        desplazaHaciaDerecha (arre, n, pos);
        arre[pos] = aInsertar;
```

```
            n = n + 1; // El arreglo tiene un elemento más.
        }
        return n; // El resultado es el nuevo tamaño.
    }

    // Método que inserta un dato en un arreglo genérico ordenado, sin permitir repeticiones.
    public static <T extends Comparable<T>> int insertaOrdenadoSinRepetidos(T arre[ ], int n, T
    aInsertar){
        if (n < arre.length){ // Hay espacio en el arreglo
            int pos = buscaSecuencialOrdenado(arre, n, aInsertar);
            if (pos < 0){  // Si no está obtiene la posición en la que debe asignarlo.
                pos = -pos -1;
                desplazaHaciaDerecha (arre, n, pos);
                arre[pos] = aInsertar;
                n = n + 1; // El arreglo tiene un elemento más.
            }
        }
        return n; // El resultado es el nuevo tamaño.
    }

    // Método que elimina un elemento de un arreglo genérico desordenado.
    public static <T> int eliminaDesordenado(T arre[ ], int n, T aEliminar){
        int pos;

        pos = buscaSecuencialDesordenado(arre, n, aEliminar);
        if (pos >= 0){// El dato está en el arreglo
            arre[pos] = arre[n-1]; // Se reemplaza por el contenido de la última casilla
            arre[n-1] = null;
            n = n - 1; // Disminuye en 1 el total de elementos
        }
        return n;
    }

    /* Método auxiliar que desplaza todos los elementos del arreglo una posición a la
     * izquierda, sin alterar el contenido de las casillas.
     */
```

```
public static <T> void desplazaHaciaIzquierda (T arre[ ], int n, int pos){
   int i;
   for (i = pos; i < n - 1; i++)
      arre[i] = arre[i+1];
}

// Método que elimina un elemento de un arreglo genérico ordenado.
  public static <T extends Comparable <T>> int eliminaOrdenado(T arre[ ], int n, T aEliminar){
    int pos;

    pos = buscaSecuencialOrdenado(arre, n, aEliminar);
    if (pos >= 0){ // El dato está en el arreglo
       desplazaHaciaIzquierda(arre, n, pos);
       arre[n-1] = null; // Se pierde la referencia al objeto de esta posición
       n = n - 1; // Disminuye en 1 el total de elementos
    }
    return n;
  }

/* Método que regresa una cadena formada con el contenido del arreglo que recibe
 * como parámetro. Requiere que los tipos con los cuales se dé valor a T tengan un
 * toString( ). Es muy útil para conocer el contenido del arreglo.
 */
 public static <T> String imprime(T arre[ ], int n){
      int i;
      StringBuilder cad = new StringBuilder( );

      for (i = 0; i < n; i++)
         cad.append(arre[i] + " ");
      return cad.toString( );
    }
```

Programa 4.4 **UsaOAG.java**

```
package cap4;

/**
 *
 * @author Silvia Guardati
 * Programa 4.4
 * Ejemplo de uso de los métodos de la clase OAG en arreglos de tipo Double y en
 * arreglos de tipo String.
 */
public class UsaOAG {

   public static void main(String[ ] args) {
      Double califDes[ ] = new Double[10];
      String díasOrd[ ] = new String[7];

      califDes[0] = 8.5;
      califDes[1] = 9.3;
      califDes[2] = 5.5;

      díasOrd[0] = "Jueves";
      díasOrd[1] = "Martes";
      díasOrd[2] = "Sábado";

      // Impresión del arreglo de datos de tipo Double (está desordenado).
      System.out.println("\nArreglo de tipo Double: " + OAG.imprime(califDes, 3));

      // Impresión del arreglo de cadenas de caracteres (está ordenado).
      System.out.println("\nArreglo de tipo String: " + OAG.imprime(díasOrd, 3));

      // Búsqueda secuencial en arreglos desordenados.
      int índice = OAG.buscaSecuencialDesordenado(califDes, 3, 8.5);
      System.out.println("\nDebe imprimir 0: " + índice);
      índice = OAG.buscaSecuencialDesordenado(califDes, 3, 5.5);
      System.out.println("Debe imprimir 2: " + índice);
      índice = OAG.buscaSecuencialDesordenado(califDes, 3, 10.0);
      System.out.println("Debe imprimir -1: " + índice);
```

```
// Búsqueda secuencial en arreglos ordenados.
índice = OAG.buscaSecuencialOrdenado(díasOrd, 3, "Jueves");
System.out.println("\nDebe imprimir 0: " + índice);
índice = OAG.buscaSecuencialOrdenado(díasOrd, 3, "Sábado");
System.out.println("Debe imprimir 2: " + índice);
índice = OAG.buscaSecuencialOrdenado(díasOrd, 3, "Viernes");
System.out.println("Debe imprimir -4: " + índice);
índice = OAG.buscaSecuencialOrdenado(díasOrd, 3, "Lunes");
System.out.println("Debe imprimir -2: " + índice);
índice = OAG.buscaSecuencialOrdenado(díasOrd, 3, "Domingo");
System.out.println("Debe imprimir -1: " + índice);

// Inserción en arreglos desordenados, permitiendo elementos repetidos.
int n = OAG.insertaDesordenadoConRepetidos(califDes, 3, 8.5);
String res = OAG.imprime(califDes, n);
System.out.println("\n\nInserta 8.5 en desordenado con repetidos: " + res);

// Inserción en arreglos desordenados, NO permitiendo elementos repetidos.
n = OAG.insertaDesordenadoSinRepetidos(califDes, n, 5.5);
res = OAG.imprime(califDes, n);
System.out.println("\nInserta 5.5 en desordenado sin repetidos -no puede: " + res);

n = OAG.insertaDesordenadoSinRepetidos(califDes, n, 10.0);
res = OAG.imprime(califDes, n);
System.out.println("\nInserta 10.0 en desordenado sin repetidos: " + res);

// Inserción en arreglos ordenados, permitiendo elementos repetidos.
int m = OAG.insertaOrdenadoConRepetidos(díasOrd, 3, "Jueves");
res = OAG.imprime(díasOrd, m);
System.out.println("\n\nInserta Jueves en ordenado con repetidos: " + res);

// Inserción en arreglos ordenados, NO permitiendo elementos repetidos.
m = OAG.insertaOrdenadoSinRepetidos(díasOrd, m, "Martes");
res = OAG.imprime(díasOrd, m);
System.out.println("\nInserta Martes en ordenado sin repetidos -no puede: " + res);
```

```
m = OAG.insertaOrdenadoSinRepetidos(díasOrd, m, "Lunes");
res = OAG.imprime(díasOrd, m);
System.out.println("\nInserta Lunes en ordenado sin repetidos: " + res);

m = OAG.insertaOrdenadoSinRepetidos(díasOrd, m, "Domingo");
res = OAG.imprime(díasOrd, m);
System.out.println("\nInserta Domingo en ordenado sin repetidos: " + res);

m = OAG.insertaOrdenadoSinRepetidos(díasOrd, m, "Viernes");
res = OAG.imprime(díasOrd, m);
System.out.println("\nInserta Viernes en ordenado sin repetidos: " + res);

// Eliminación en arreglos desordenados.
n = OAG.eliminaDesordenado(califDes, n, 8.5);
res = OAG.imprime(califDes, n);
System.out.println("\n\nElimina la primer ocurrencia de 8.5 en desordenado: " + res);

n = OAG.eliminaDesordenado(califDes, n, 5.5);
System.out.println("\nElimina 5.5 en desordenado: " + OAG.imprime(califDes, n));

n = OAG.eliminaDesordenado(califDes, n, 7.8);
res = OAG.imprime(califDes, n);
System.out.println("\nElimina 7.8 -no está- en desordenado: " + res);

// Eliminación en arreglos ordenados.
m = OAG.eliminaOrdenado(díasOrd, m, "Jueves");
res = OAG.imprime(díasOrd, m);
System.out.println("\n\nElimina primer ocurrencia de Jueves en ordenado: " + res);

m = OAG.eliminaOrdenado(díasOrd, m, "Jueves");
res = OAG.imprime(díasOrd, m);
System.out.println("\nElimina segunda ocurrencia de Jueves en ordenado: " + res);

m = OAG.eliminaOrdenado(díasOrd, m, "Miércoles");
res = OAG.imprime(díasOrd, m);
```

```
        System.out.println("\nElimina Miércoles -no está- en ordenado: " + res);
        m = OAG.eliminaOrdenado(díasOrd, m, "Martes");
        System.out.println("\nElimina Martes en ordenado: " + OAG.imprime(díasOrd, m));
    }
}
```

4.6 APLICACIÓN DE ARREGLOS

Antes de continuar estudiando otras operaciones y diferentes tipos de arreglos, es necesario presentar por qué son importantes las estructuras de datos y, en particular, los arreglos para la solución de cierta clase de problemas.

» Ejemplo 1

Considere que dispone del total de goles anotados por cada uno de los equipos participantes del último mundial de futbol. Con esa información le piden que calcule el promedio de goles anotados, el equipo que más goles anotó y cuántos equipos anotaron menos goles que el promedio. Para resolver este problema requiere poder almacenar 32 valores numéricos, uno por cada uno de los equipos, y conservarlos en memoria para calcular el promedio y luego comparar cada uno de los valores con el promedio para contar cuántos anotaron menos goles que el promedio. Si sólo se dispusiera de datos numéricos simples sería bastante ineficiente resolver este problema, ya que se debería hacer la lectura de los datos dos veces (una antes y otra después de calcular el promedio), o bien, usar 32 variables para almacenar cada uno de los datos y así disponer de ellos cuantas veces se requiera.

Por medio de los arreglos, el problema anterior tiene fácil solución, ya que los datos se leen una sola vez en una única variable que tiene la capacidad de almacenar 32 valores y, como permanecen en memoria, se pueden usar las veces necesarias para generar toda la información solicitada. En el programa 4.5 se presenta la solución de este problema. Se definió la clase *Mundial* que tiene como atributo el año, la sede y los goles anotados por los 32 equipos. Se incluyó un *main* muy simple para probar la solución.

Programa 4.5 | **Mundial.java**

```
package cap4;

import java.io.File;
import java.util.Scanner;
```

```
/**
 * @author Silvia Guardati
 * Programa 4.5
 * Ejemplo de aplicación de arreglos para la solución de problemas.
 * Programa que calcula: a) el promedio de goles anotados por todos los equipos durante
 * el último mundial de futbol, b) cuántos equipos anotaron menos goles que el promedio,
 * y c) total de equipos que anotaron 0 goles durante el mundial.
 */
public class Mundial {

  private String sede;
  private int año;
  private final int TOTAL_PAÍSES = 32;
  private int goles[ ];

  // Se instancia el arreglo declarado como atributo
  public Mundial( ) {
    goles = new int[TOTAL_PAÍSES];
  }

  // Además de instanciar el arreglo, se le asignan valores a sede y año
  public Mundial(String sede, int año) {
    this( );
    this.sede = sede;
    this.año = año;
  }

  /* Se leen de un archivo los goles anotados por cada uno de los 32 equipos
   * participantes. Si el archivo no puede abrirse, regresa false y al atributo
   * goles se le asigna null.
   */
  public boolean leeGoles(String nomArch){
    int i;
    boolean resp = true;
```

```
        try{
          Scanner lee = new Scanner(new File(nomArch));
          i = 0;
          while (i < TOTAL_PAÍSES && lee.hasNext()){
             goles[i] = lee.nextInt();
             i = i + 1;
          }
          if (i< TOTAL_PAÍSES)
             resp = false;
          lee.close();
        }catch (Exception e){
          goles = null;
          resp = false;
        }
        return resp;
    }

    /* Se utiliza el método correspondiente de la clase OAE para calcular el
     * promedio de goles anotados por los 32 equipos.
     */
    public double promedioDeGoles( ){
        return OAE.calculaPromedio(goles, TOTAL_PAÍSES);
    }

    /* Se utiliza el método de OAE para obtener el total de equipos que anotaron menos
     * goles que el promedio.
     */
    public int cuentaMenosPromedio(double promGoles){
        return OAE.cuentaMenores(goles, TOTAL_PAÍSES, (int)promGoles);
    }

    // Se utiliza el método de OAE para contar cuántos equipos anotaron 0 goles.
    public int cuentaCeros( ){
        return OAE.cuentaIguales(goles, TOTAL_PAÍSES, 0);
    }
```

```
    public static void main(String[ ] args) {
      Mundial último = new Mundial("Brasil", 2014);
      double promGoles;
      int totalEq;

      /* El archivo "Goles" se encuentra en el proyecto EstructurasDatosBásicas.
       * Si no se pudo leer el archivo sólo se imprime un mensaje adecuado.
       */
      if (último.leeGoles("Goles")) {
        promGoles = último.promedioDeGoles( );
        System.out.println("\nPromedio de goles anotados en el último mundial: " + promGoles);

        totalEq = último.cuentaMenosPromedio(promGoles);
        System.out.println("Total de equipos que anotaron menos goles que el promedio: " +
        totalEq);

        totalEq = último.cuentaCeros( );
        System.out.println("Total de equipos que anotaron 0 goles: " + totalEq);
      }
      else
        System.out.println("\nError al leer el archivo.");
    }
}
```

» **Ejemplo 2**

De un alumno se conoce su nombre, el nombre de la carrera que está estudiando y las calificaciones de las materias cursadas hasta el momento (valores comprendidos entre 0 y 10). Entre la funcionalidad esperada está: 1) agregar nuevas calificaciones a medida que el alumno cursa materias, 2) saber si sacó al menos un 10 y 3) obtener toda la información del alumno. Observe el programa 4.6. En los métodos de la clase *Alumno* se hace uso de los métodos de la clase OAE (programa 4.1) para implementar las operaciones sobre el arreglo de calificaciones. Se incluyó un método *main* para probar la funcionalidad.

Programa 4.6 | **Alumno.java**

```
package cap4;

/**
 * @author Silvia Guardati
```

```
* Programa 4.6
* Ejemplo de aplicación de la clase OAE para la solución de problemas en los cuales
* se requiere el uso de arreglos de enteros.
*/
public class Alumno{
  private String nombreAlum, nombreCar;
  private int califs[ ]; // Almacena las calificaciones obtenidas.
  private int matCur; // Almacena el total de materias cursadas.
  private final int MAX_MAT = 60; // Máximo de materias a cursar.

  /* Se instancia el arreglo declarado como atributo y se indica que el total
   * de elementos guardados en el arreglo es 0.
   */
  public Alumno( ) {
    califs = new int[MAX_MAT];
    matCur = 0;
  }

  // Además de instanciar el arreglo, se asignan valores a algunos atributos
  public Alumno(String nombreAlum, String nombreCar) {
    this( );
    this.nombreAlum = nombreAlum;
    this.nombreCar = nombreCar;
  }

  /* El alumno termina de cursar una materia. Se guarda la calificación obtenida
   * en el arreglo —si hay espacio— y se actualiza el total.
   */
  public boolean altaCalif(int cal){
    int n;
    boolean resp;

    resp = false;
    n = OAE.insertaDesordenadoConRepetidos(califs, matCur, cal);
    if (matCur < n){
      resp = true; // Asigna true si se pudo insertar
```

```
        matCur = n; // Actualiza el total de materias cursadas
      }
      return resp;
   }

   // Método que regresa true si el alumno obtuvo al menos un 10.
   public boolean sacóDiez( ){
      return OAE.buscaSecuencialDesordenado(califs, matCur, 10) >= 0;
   }

   // Método que imprime los datos del alumno, incluyendo las calificaciones.
   public String toString( ) {
      StringBuilder cad = new StringBuilder( );
      cad.append("Alumno: " + nombreAlum);
      cad.append("\nCursa la carrera: "+ nombreCar);
      cad.append("\nCalificaciones: " + OAE.imprime(califs, matCur));
      return cad.toString( );
   }

   public static void main(String[ ] args) {
      Alumno unAlumno = new Alumno ("Joaquín Gutiérrez", "Ingeniería Industrial");

      // Captura las calificaciones obtenidas por el alumno.
      unAlumno.altaCalif(6);
      unAlumno.altaCalif(9);
      unAlumno.altaCalif(10);
      unAlumno.altaCalif(7);
      unAlumno.altaCalif(8);

      // Informa si el alumno sacó al menos un 10.
      if (unAlumno.sacóDiez( ))
         System.out.println("¡Sí obtuvo al menos un diez!");
      else
         System.out.println("Hasta ahora no obtuvo ningún diez.");
```

```
        // Imprime todos los datos del alumno.
        System.out.println(unAlumno);
    }
}
```

» Ejemplo 3

Se debe representar el concepto de una tienda que tiene como atributos el nombre, la dirección y los nombres de los productos que vende. La funcionalidad que se pide es: 1) consultar si un producto es vendido en la tienda, 2) agregar un nuevo producto, 3) eliminar un producto y 4) generar un listado de todos los productos. Se quiere que los nombres se almacenen de manera ordenada. Para resolver el problema se usarán los métodos de la clase OAG. La solución se presenta en el programa 4.7, en el cual, además de la clase *Tienda*, se incluye un método *main* para probar el funcionamiento de los métodos.

Programa 4.7 Tienda.java

```
package cap4;

/**
 * @Silvia Guardati
 * Programa 4.7
 * Clase Tienda que usa los métodos de la clase OAG para realizar las operaciones
 * sobre el arreglo de nombres de productos.
 */
public class Tienda {
    private String nombre, dirección;
    private String nomProductos[ ];
    private int totalProd;
    private final int MAX_PROD = 50;

    /* Se instancia el arreglo declarado como atributo y se indica que el total
     * de elementos guardados en el arreglo es 0.
     */
    public Tienda( ) {
        nomProductos = new String[MAX_PROD];
        totalProd = 0;
    }
```

```
// Además de instanciar el arreglo, se asignan valores a algunos atributos
public Tienda(String nombre, String dirección) {
   this();
   this.nombre = nombre;
   this.dirección = dirección;
}

public boolean consultaProducto(String nombre){
   int pos;

   pos = OAG.buscaSecuencialOrdenado(nomProductos, totalProd, nombre);
   /* Si el valor de pos es mayor o igual a cero, quiere decir que se encontró
    * el producto, y por lo tanto regresará true. En caso contrario, regresará
    * false indicando que la tienda no vende ese producto.
    */
   return pos >= 0;
}

// Agrega un nuevo producto (no debe repetirse).
public void altaProducto(String nombre){
   totalProd = OAG.insertaOrdenadoSinRepetidos(nomProductos, totalProd, nombre);
}

// Elimina uno de los productos que vende la tienda.
public void bajaProducto(String nombre){
   totalProd = OAG.eliminaOrdenado(nomProductos, totalProd, nombre);
}

// Genera una lista con los nombres de todos los productos.
public String listaProductos(){
   return OAG.imprime(nomProductos, totalProd);
}

public static void main(String[] args) {
   // Se construye un objeto tipo Tienda
   Tienda unaTienda = new Tienda("La última opción", "Callejón 4, número 123");
```

```
        // Se dan de alta algunos productos.
        unaTienda.altaProducto("Harina");
        unaTienda.altaProducto("Azúcar");
        unaTienda.altaProducto("Mermelada");
        unaTienda.altaProducto("Aceite");
        unaTienda.altaProducto("Sal");
        System.out.println("\nLista de productos: " + unaTienda.listaProductos());

        // Se elimina uno de los productos.
        unaTienda.bajaProducto("Mermelada");
        System.out.println("\nLista de productos: " + unaTienda.listaProductos());

        // Se consulta si la tienda vende harina. Debe dar una respuesta afirmativa.
        if (unaTienda.consultaProducto("Harina"))
            System.out.println("\nLa tienda sí vende harina.");
        else
            System.out.println("\nLa tienda no vende harina.");

        // Se consulta si la tienda vende tequila. Debe dar una respuesta negativa.
        if (unaTienda.consultaProducto("Tequila"))
            System.out.println("\nLa tienda sí vende tequila.");
        else
            System.out.println("\nLa tienda no vende tequila.");
    }
}
```

4.7 ARREGLOS PARALELOS

Se llama arreglos paralelos a aquellos que almacenan información relacionada y en los cuales la posición de cada elemento ocupa un papel fundamental, ya que el contenido de la casilla 0 en uno de los arreglos es un dato que complementa al dato almacenado en la casilla 0 del otro arreglo. Esta relación se mantiene para cada una de las casillas de los arreglos. En general, los arreglos paralelos son muy útiles para guardar diferentes datos de un mismo concepto.

En la figura 4.7 se presenta un ejemplo de arreglos paralelos. En el primero de ellos se almacenan los nombres de un grupo de alumnos, mientras que en el segundo se almacenan las calificaciones obtenidas por dichos alumnos en un examen. En este ejemplo, el alumno Raúl obtuvo 8.5 de calificación, la alumna María sacó 9, la alumna Claudia un 9.3, y así sucesivamente. Dada la relación entre los datos, se puede acceder a partir de un valor de uno de los arreglos a su correspondiente en el otro arreglo. Siguiendo con el ejemplo, dado el nombre de un alumno se puede encontrar qué calificación sacó en el examen. Para ello, se busca en el arreglo de nombres y, si está, se usa su posición para tener acceso a su calificación. O bien, se podrían obtener todos los alumnos que sacaron calificaciones mayores a 9. Para esto se debería recorrer el arreglo de calificaciones y si la calificación es mayor a 9, entonces se imprime la casilla correspondiente del arreglo de nombres.

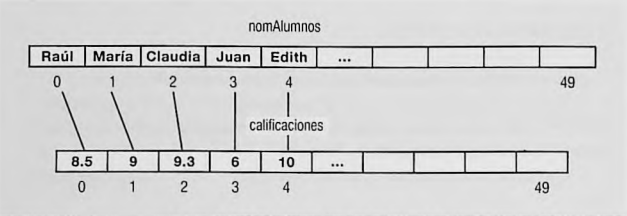


Figura 4.7 Ejemplo de arreglos paralelos

En el programa 4.8 se presenta el código correspondiente a este ejemplo. Es un problema muy simple que sirve para ilustrar cómo se trabaja con los arreglos paralelos.

Programa 4.8 **EjemploArreglosParalelos.java**

```
package cap4;

import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Programa 4.8
 * Ejemplo de uso de arreglos paralelos en la solución de problemas.
 * Funcionalidad de la clase: a) dado el nombre de un alumno, encuentra e imprime su
 * calificación y b) imprime los nombres de todos los alumnos que obtuvieron una
 * calificación mayor a 9.
 */
```

```
public class EjemploArreglosParalelos {
   public static void main(String[ ] args) {
      final int MAX = 50;
      String nomAlumnos [ ] = new String[MAX];
      double calificaciones [ ] = new double[MAX];
      int i, totalAl, pos;
      String nombre;
      Scanner lee = new Scanner(System.in);

      do{
         System.out.println("\nIngresa total de alumnos: ");
         totalAl = lee.nextInt( );
      } while (totalAl < 1 || totalAl > MAX);

      for (i = 0; i < totalAl; i++){
         System.out.println("\nIngresa nombre del alumno " + (i + 1) + ": ");
         nomAlumnos[i] = lee.next( );
         System.out.println("\nIngresa calificación de " + nomAlumnos[i] + ": ");
         calificaciones[i] = lee.nextDouble( );
      }

      /* Dado el nombre de un alumno, si está en el arreglo de nombres, entonces
       * se usa su posición para obtener la calificación en el arreglo de
       * calificaciones.
       */
      System.out.println("\nDe quién quieres conocer la calificación: ");
      nombre = lee.next( );
      pos = OAG.buscaSecuencialDesordenado(nomAlumnos, totalAl, nombre);
      if (pos >= 0) // Verifica que lo haya encontrado.
         System.out.println("\n" + nombre + " obtuvo: " + calificaciones[pos]);
      else
         System.out.println("\nEse alumno no está registrado.");

      /* Se revisa todo el arreglo de calificaciones y, cada vez que se encuentra
       * una mayor a 9, se imprime el alumno que la obtuvo: está en la misma
       * posición pero en el arreglo de nombres.
       */
```

```
        System.out.println("\nAlumnos con calificación > 9: ");
        for (i = 0; i < totalAl; i++)
          if (calificaciones[i] > 9)
            System.out.println(nomAlumnos[i]);
    }
}
```

Retomamos el problema del mundial de futbol en el cual se almacenan los goles anotados por cada uno de los equipos participantes. Supongamos ahora que, además de los goles, se deben almacenar los nombres de los países. La información se vería como se muestra en la figura 4.8.

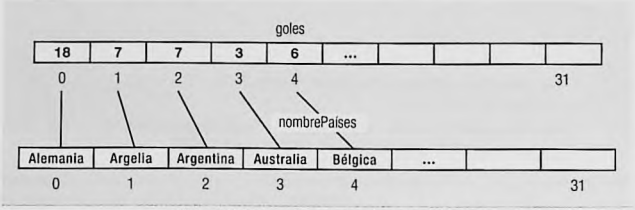


Figura 4.8 Ejemplo de arreglos paralelos

En el ejemplo los datos deben interpretarse como sigue: Alemania anotó 18 goles, Argelia anotó 7, Argentina anotó 7 goles, Australia 3, y así sucesivamente. De manera general, se está expresando que el país cuyo nombre se guarda en la casilla 0 del arreglo *nombrePaises* anotó los goles guardados en la casilla 0 del arreglo *goles*; el país cuyo nombre se guarda en la casilla 1 del arreglo *nombrePaises* anotó los *goles* guardados en la casilla 1 del arreglo *goles*, y así hasta el final. En el programa 4.9 se presenta una aplicación de arreglos paralelos para procesar la información del mundial.

Programa 4.9 **MundialConPaíses**

```
package cap4;
import java.io.File;
import java.util.Scanner;

/**
 * @author Silvia Guardati
```

```
* Programa 4.9
* Ejemplo de aplicación de arreglos paralelos para la solución de problemas.
* Esta clase es una ampliación de la clase Mundial (programa 4.5).
* Funcionalidad de la clase: a) calcula el promedio de goles anotados por todos
* los equipos durante el último mundial de futbol, b) obtiene cuántos equipos
* anotaron menos goles que el promedio, c) proporciona los nombres de los equipos
* que anotaron 0 goles durante el mundial y d) da el nombre del equipo que anotó
* más goles. En algunos métodos se hace uso de la clase OAE.
*/
public class MundialConPaíses {

   private String sede;
   private int año;
   private final int TOTAL_PAÍSES = 32;
   private int goles[ ];
   private String nomPaíses[ ];

  // Se instancian los arreglos declarados en la sección de atributos
   public MundialConPaíses( ) {
      goles = new int[TOTAL_PAÍSES];
      nomPaíses = new String[TOTAL_PAÍSES];
   }
   public MundialConPaíses(String sede, int año) {
      this( );
      this.sede = sede;
      this.año = año;
   }

   /* Se leen de un archivo los goles anotados por cada uno de los 32 equipos
    * participantes. Si el archivo no puede abrirse, regresa false.
    */
   public boolean leeGoles(String nomArch){
      int i;
      boolean resp = true;

      try{
         Scanner lee = new Scanner(new File(nomArch));
```

```
        i = 0;
        while (i < TOTAL_PAÍSES && lee.hasNext()){
          goles[i] = lee.nextInt();
          i = i + 1;
        }
        if (i < TOTAL_PAÍSES)
          res=false;
        lee.close();
      }catch (Exception e){
        goles = null;
        resp = false;
      }
      return resp;
    }

    /* Se leen de un archivo los nombres de los 32 países participantes.
     * Si el archivo no puede abrirse, regresa false.
     */
    public boolean leeNomPaíses(String nomArch){
      int i;
      boolean resp = true;

      try{
        Scanner lee = new Scanner(new File(nomArch));
        i = 0;
        while (i < TOTAL_PAÍSES && lee.hasNextLine()){
          nomPaíses[i] = lee.nextLine();
          i = i + 1;
        }
        if (i < TOTAL_PAÍSES)
          res=false;
        lee.close();
      }catch (Exception e){
        nomPaíses = null;
        resp = false;
      }
```

```
    return resp;
  }

  // Se utiliza la clase OAE para obtener el promedio.
  public double promedioDeGoles( ){
    return OAE.calculaPromedio(goles,TOTAL_PAÍSES);
  }

  /* Se utiliza la clase OAE para determinar cuántos equipos anotaron menos
   * goles que el promedio.
   */
  public int cuentaMenosPromedio(double promGoles){
    return OAE.cuentaMenores(goles, TOTAL_PAÍSES, (int) promGoles);
  }

  // Obtiene los nombres de los países cuyos equipos no anotaron goles.
  public String obtieneEqConCeros( ){
    int i;
    StringBuilder cad = new StringBuilder("\nNo anotaron goles los equipos de los países: ");

    for (i = 0; i < TOTAL_PAÍSES; i++)
      if (goles[i] == 0)
        cad.append(nomPaíses[i] + " - ");

    return cad.toString( );
  }

  // Obtiene nombre del equipo que anotó más goles.
  public String obtieneNomMásGoles( ){
    int pos;

    pos = OAE.buscaPosMayor(goles, TOTAL_PAÍSES);
    return nomPaíses[pos];
  }

  public static void main(String[ ] args) {
    MundialConPaíses último = new MundialConPaíses("Brasil", 2014);
    double promGoles;
```

```
        int totalEq;
        String nomPaíses, paísMásGoles;

        // Los archivos "Países" y "Goles" se encuentran en el proyecto EstructurasDatosBásicas.
        if (último.leeNomPaíses("Países") && último.leeGoles("Goles")) {
          // Calcula e imprime el promedio de goles.
          promGoles = último.promedioDeGoles( );
          System.out.println("\nPromedio de goles anotados en el mundial: " + promGoles);

          // Obtiene e imprime el total de equipos que anotaron menos goles que el promedio.
          totalEq = último.cuentaMenosPromedio(promGoles);
          System.out.println("\nTotal de equipos que anotaron menos goles que el promedio: " +
          totalEq);

          // Obtiene e imprime los nombres de los países que no anotaron goles.
          nomPaíses = último.obtieneEqConCeros( );
          System.out.println(nomPaíses);

          // Obtiene e imprime el nombre del país que más goles anotó.
          paísMásGoles = último.obtieneNomMásGoles( );
          System.out.println("\nPaís que más goles anotó: " + paísMásGoles);
        }
        else
          System.out.println("\nError en la lectura de los archivos.\n");
      }
    }
```

4.8 RESUMEN

En este capítulo se presentaron los arreglos unidimensionales, siendo la primera estructura de datos estudiada en este libro. Asimismo, se explicaron las principales ventajas y desventajas que tienen estas estructuras.

También se mostró como almacenar y recuperar datos por medio de esta estructura, así como su uso en la solución de problemas.

Todos los temas se complementaron con ejemplos para ayudar al lector a la comprensión de los mismos.

4.9 EJERCICIOS

4.1 Defina la clase *CompetenciaNatación* que tiene entre sus atributos un arreglo con las alturas de un grupo de nadadores y el total de nadadores, máximo 50. Puede incluir otros atributos. Agregue los métodos necesarios para tener la siguiente funcionalidad:

b. Agregar la altura de un nuevo nadador que se incorporó a la competencia.

c. Calcular e imprimir el promedio de altura del grupo.

d. Calcular la desviación estándar del grupo:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

e. Dada una altura (dato de entrada), contar e imprimir el total de nadadores que tienen una altura mayor a la misma.

4.2 Considere que se ha almacenado las claves de un grupo de productos, ordenadas crecientemente. Durante la captura de las claves, por error, se ingresaron algunas claves repetidas, ocupando posiciones consecutivas. Escriba una aplicación que elimine todos los elementos repetidos del arreglo. Una vez eliminados, se deberá imprimir el arreglo. Pruebe su solución.

4.3 Escriba el método *estáOrdenadoCreciente()* en la clase OAG que reciba un arreglo genérico y el total de elementos. Regrese *true* si el arreglo está ordenado crecientemente y *false* en caso contrario. Debe procurar una solución eficiente.

4.4 Escriba el método *equals()* en la clase OAG que reciba como parámetro dos arreglos genéricos y el total de elementos que almacena cada uno de ellos. Regrese *true* si los arreglos son iguales y *false* en caso contrario. Dos arreglos son iguales si tienen la misma cantidad de elementos, el mismo tipo de elementos y elementos iguales entre sí, en contenido y en orden.

4.5 Retome el método de búsqueda secuencial para arreglos ordenados crecientemente. Haga los cambios necesarios para que pueda utilizarse con arreglos ordenados decrecientemente.

4.6 Escriba el método *existeOcurrencia()* en la clase OAG que reciba un arreglo genérico, el total de elementos, un dato genérico y un entero *n*. El método debe regresar *true* si en el arreglo hay, por lo menos, *n* ocurrencias del dato. En caso contrario, debe regresar *false*. Por ejemplo, si el arreglo contiene los valores: 3, 6, 9, 3, 5, 10, 1, 3 y 7, el dato = 3 y n = 2, entonces el método debe regresar *true* porque el arreglo tiene al menos 2 ocurrencias del dato (3). Debe procurar una solución eficiente. Pruebe su solución.

4.7 Una agencia automotriz tiene como atributos: nombre, dirección, teléfono y el total de unidades vendidas por mes. Para guardar las ventas se usa un arreglo de enteros de 12 posiciones. Las posiciones correspondientes a los meses posteriores a la fecha actual almacenan 0. En la clase se deben agregar los métodos que permitan la funcionalidad que se describe más abajo. Si lo cree conveniente, puede apoyarse en la clase OAE.

a. Se hace una venta. Se dan como datos el mes actual y el total de unidades vendidas. Actualice la información que corresponda.

b. Se quiere imprimir toda la información de la agencia.

c. Al terminar el año se quiere conocer el promedio de unidades vendidas.

d. Al terminar el año se quiere calcular en cuántos meses se vendieron más de 50 unidades.

e. Al terminar el año se quiere calcular en cuántos meses no se vendieron automóviles.

4.8 En las últimas elecciones en el país Pueblo Unido se presentaron 6 candidatos. Se tiene la información de los votos y con ella se quiere generar información útil para el electorado. La información se proporciona de la siguiente manera:

candidato

candidato

candidato

...

-1

donde: candidato es un entero ($-1 \leq$ candidato ≤ 7)

Si es 1, 2, ..., 6, indica el número del candidato al que votó.

Si es 0, indica que fue un voto en blanco.

Si es 7, indica que el voto se anuló.

Si es -1, indica que ya no hay datos (bandera de fin de datos).

Organice la clase como lo crea conveniente para que tenga la siguiente funcionalidad:

a. Almacenar el total de votos recibidos por cada uno de los 6 candidatos, así como el total de votos anulados y el total de votos en blanco.

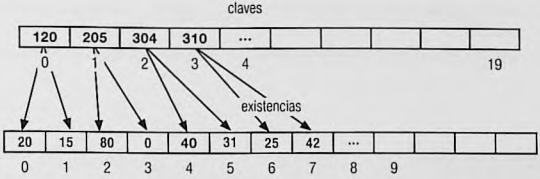
b. Obtener el candidato que ganó las elecciones (es el que recibió más votos).

c. Calcular el porcentaje de votos anulados con respecto al total de votos emitidos.

d. Calcular el porcentaje de votos en blanco con respecto al total de votos emitidos.

e. Determinar si el número de votos en blanco es mayor que el número de votos recibidos por alguno de los candidatos. Si es así, se debe generar un mensaje de advertencia.

4.9 Una librería tiene, de cada libro, su clave y el total de unidades con encuadernación estándar y el total de unidades con encuadernación de lujo. Para las claves se usa un arreglo; para las existencias, otro. También se conoce el total de libros que vende la librería ($1 \leq n \leq 20$). Observe la figura que aparece página siguiente.



Del libro con clave 120 se tienen 20 unidades con encuadernación estándar y 15 unidades de lujo; del libro con clave 205 se tienen 80 unidades con encuadernación estándar y 0 unidades de lujo, etcétera.

a. Defina la clase Librería que tiene los siguientes atributos: nombre, dirección, teléfono, claves de los libros y existencias en sus dos versiones de encuadernación (estándar y de lujo). La librería debe tener la funcionalidad que se describe en los siguientes puntos.

b. Método que imprima la clave y el total de libros disponibles para la venta, de todos los libros que tiene la librería. Por cada libro se debe imprimir su clave y el total de libros (considerando los 2 tipos de encuadernación).

c. Método que regresa la clave del libro del cual se tiene más del doble de unidades estándar que de lujo. Suponga que, si existe, sólo un libro cumple con esta condición.

d. Método que regresa de cuántos libros no se tienen ejemplares con encuadernación de lujo. Es decir, sólo se tienen ejemplares con encuadernación estándar.

e. Escriba un método *main* para probar su solución.

4.10 Escriba una clase aplicación (puede revisar la clase *MundialConPaíses*) que tenga, entre sus atributos, los nombres de los países participantes en el último mundial, así como los goles que anotó y que recibió cada equipo. La funcionalidad esperada se describe más abajo. Posteriormente, escriba un método main para probar su solución.

a. Leer los nombres de los países participantes en el último mundial y guardarlos en un arreglo de cadenas.

b. Leer los goles anotados por cada uno de los equipos participantes del mundial y guardarlos en un arreglo de enteros.

c. Leer los goles recibidos por cada uno de los equipos participantes del mundial y guardarlos en un arreglo de enteros.

d. Ordenar el arreglo de nombres alfabéticamente. Tenga en cuenta que el arreglo es paralelo con los que guardan los goles a favor y los goles en contra.

e. Encontrar e imprimir el nombre del equipo que más goles anotó.

f. Encontrar e imprimir el nombre del equipo que más goles recibió.

g. Imprimir, si es posible, el nombre de todos los equipos que no recibieron goles a lo largo de su participación en el mundial.

h. Imprimir, si es posible, el nombre de todos los equipos que anotaron más de 10 goles durante el mundial.

i. Imprimir el nombre de todos los equipos que recibieron más goles que los que anotaron.

Arreglos y POO

5

Contenido

Competencias

- Presentar el diseño y la implementación de arreglos usando el paradigma de programación orientada a objetos.
- Explicar los arreglos polimórficos y mostrar que constituyen una herramienta poderosa para almacenar y recuperar datos.
- Presentar los arreglos bidimensionales y su implementación por medio del paradigma de programación orientada a objetos.
- Mostrar el uso de los arreglos bidimensionales en la solución de problemas.
- Explicar las clases ArrayList y Vector de Java.

5.1 INTRODUCCIÓN

En el capítulo anterior se presentaron los arreglos, como declarar y construir variables tipo arreglos, las principales operaciones y su uso en la solución de problemas. En todos los casos se manejó a los arreglos (datos) y a las operaciones que se pueden realizar sobre ellos de manera separada.

En este capítulo se retoman los arreglos pero ahora usando el paradigma de la programación orientada a objetos para su representación y manejo. Por lo tanto, se definirá una clase cuyos miembros serán el arreglo (datos) y las operaciones a realizar sobre el arreglo (métodos).

También en este capítulo se estudian los arreglos bidimensionales y algunas operaciones que pueden hacerse utilizando los mismos.

Por último, se presentan las clases *ArrayList* y *Vector* de Java, las cuales utilizan arreglos unidimensionales para almacenar datos. Ambas son genéricas, es decir, permiten manejar diferentes tipos de datos.

5.2 LA CLASE ARREGLO

Para la implementación de los arreglos se puede usar la orientación estructurada, en la cual se ve por un lado la colección de elementos y de manera independiente el conjunto de operaciones válidas para dicha colección. Es decir, datos y operaciones están separados, como se puede apreciar en lo estudiado hasta el momento.

El otro enfoque, y es el que presentamos en esta sección, es el de la orientación a objetos. En este caso, el arreglo se define como una clase formada por atributos y métodos. Los atributos son la colección de elementos y el total de elementos guardados. Por su parte, los métodos son todas las operaciones que pueden realizarse con los elementos del arreglo. De esta manera, se tiene como un todo a los datos y a las operaciones. En la figura 5.1 se presenta el diagrama UML de la clase *Arreglo*.

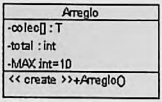


Figura 5.1 La clase *Arreglo*

Aprovechando las características de la programación orientada a objetos, las cuales se analizaron en los capítulos 2 y 3, y las operaciones sobre arreglos genéricos estudiadas en el capítulo anterior, definiremos una clase para arreglos genéricos. Por lo tanto, se usará la misma clase para almacenar y posteriormente operar diversos tipos de datos. La figura 5.2 presenta el diagrama de clase UML de la clase *ArregloGenéricoDesordenado*, utilizada para representar el concepto correspondiente a un arreglo capaz de almacenar datos de tipo T, desordenado. Después se hará lo correspondiente con los arreglos ordenados.

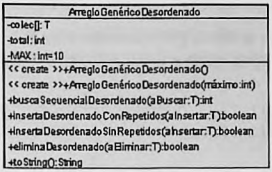


Figura 5.2 Diagrama de clase de *ArregloGenéricoDesordenado*

El programa 5.1 contiene el código requerido para definir una clase como la de la figura 5.2. Para facilitar su uso se renombra la clase como AGD (*ArregloGenéricoDesordenado*). Es importante recordar que las clases usadas para darle valor a T deberán tener sobrescrito el método *equals()* de la clase *Object*, como se vio en el capítulo 2.

Programa 5.1 | **AGD.java (Arreglo Genérico Desordenado)**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.1
 * Clase que define un arreglo genérico cuyos elementos están desordenados.
 * Se implementan las principales operaciones: búsqueda, inserción y eliminación.
 * AGD: Arreglo Genérico Desordenado.
 */
public class AGD <T> {
   private T colec[];
   private int total;
   private final int MAX = 10;

   /* Se instancia un arreglo de objetos y se lo convierte a tipo T.
    * Se utiliza la constante para definir el tamaño máximo del arreglo.
    */
   public AGD( ) {
      colec = (T[ ]) new Object[MAX];
      total = 0;
   }
```

```
/* Se instancia un arreglo de objetos y se lo convierte a tipo T.
 * Se utiliza el parámetro para definir el tamaño máximo del arreglo.
 */
public AGD(int máximo) {
   colec = (T[]) new Object[máximo];
   total = 0;
}

  // Recibe un arreglo ya creado y copia sus elementos al atributo correspondiente.
public AGD(T[] arreglo, int total) {
   colec = (T[]) new Object[arreglo.length];
   for (int i = 0 ; i < total; i++)
      colec[i] = arreglo[i];
   this.total = total;
}

// Regresa el arreglo genérico
public T[] getColec( ) {
   return colec;
}

// Regresa el total de elementos del arreglo
public int getTotal( ) {
   return total;
}

/* Regresa el valor almacenado en la casilla índice. Si el índice está fuera de
 * rango, entonces lanza una excepción.
 */
public T getElemento(int índice){
   if (índice >= 0 && índice < total)
      return colec[índice];
   else
      throw new IndexOutOfBoundsException( );
}
```

```
/* Búsqueda secuencial en un arreglo genérico desordenado.
 * Regresa la posición en la que está aBuscar o -1 si no lo encuentra.
 */
public int buscaSecuencialDesordenado(T aBuscar){
   int i;

   i = 0; // Para buscar desde la posición 0.
   while (i < total && !colec[i].equals(aBuscar))
      i++;  // Se avanza al siguiente elemento.
   if (i == total)
      i = -1;
   return i;
}

/* Método que inserta un dato en un arreglo desordenado.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio.
 */
public boolean insertaDesordenadoConRepetidos(T aInsertar){
   boolean resp = false;
   if (total < colec.length){ // Hay espacio en el arreglo
      colec[total] = aInsertar;  // total es la primera posición disponible.
      total = total + 1; // El arreglo tiene un elemento más.
      resp = true;
   }
   return resp; // Se pudo o no insertar el nuevo dato.
}

/* Método que inserta un dato en un arreglo desordenado.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio
 * y si dicho valor no se encuentra en el arreglo.
 */
public boolean insertaDesordenadoSinRepetidos(T aInsertar){
   boolean resp = false;
   if (total < colec.length) // Hay espacio en el arreglo
      if (buscaSecuencialDesordenado(aInsertar) == -1){ // No está en el arreglo
         colec[total] = aInsertar;  // total es la primera posición disponible.
         total = total + 1; // El arreglo tiene un elemento más.
```

```
            resp = true;
        }
      return resp; // Se pudo o no insertar el dato.
    }

    /* Método que elimina un elemento de un arreglo genérico desordenado.
     * aEliminar es el dato que se quiere eliminar.
     * Regresa como resultado true si pudo eliminar, false en caso contrario.
     */
    public boolean eliminaDesordenado(T aEliminar){
      int pos;
      boolean resp = false;

      pos = buscaSecuencialDesordenado(aEliminar);
      if (pos >= 0){ // El dato está en el arreglo
        // Se reemplaza por el contenido de la última casilla
        colec[pos] = colec[total-1];
        colec[total-1] = null;
        total = total - 1; // Disminuye en 1 el total de elementos
        resp = true;
      }
      return resp; // Se pudo o no eliminar el dato.
    }

    /* Regresa una cadena con el contenido del arreglo.
     * Para que la información represente el objeto que almacena, se requiere que la
     * clase usada para parametrizar tenga el método toString( ) sobrescrito.
     */
    public String toString( ){
      int i;
      StringBuilder cad = new StringBuilder( );

      for (i = 0; i < total; i++)
        cad.append(colec[i] + " ");
      return cad.toString( );
    }
}
```

En los métodos que insertan o eliminan no se regresa el total de elementos actualizado ya que, a diferencia de los métodos en la clase OAG, son métodos que pertenecen a la clase, y por lo tanto, modifican directamente el atributo correspondiente.

En el programa 5.2 se presenta un ejemplo de uso de la clase AGD. En este caso se crea un objeto de dicha clase parametrizando con la clase *Rectángulo* (programa 2.2, que se puede consultar en el paquete cap2 del proyecto EstructurasDatosBásicas que complementa este libro). Posteriormente se insertan algunos rectángulos, se hacen búsquedas, eliminaciones y se imprime todo el arreglo. Es importante destacar que para que los métodos de la clase *AGD* puedan funcionar correctamente, la clase *Rectángulo* debe tener sobrescrito los métodos *equals()* y *toString()*. El primero de ellos para las búsquedas y el segundo para el *toString()* -de *AGD*. Se sugiere al lector revisar el código y probarlo.

Programa 5.2 **EjemploAGD.java**


```
package cap5;
import cap2.Rectángulo;
/**
 * @author Silvia Guardati
 * Programa 5.2
* Ejemplo de uso de la clase AGD.
 */
public class EjemploAGD {
   public static void main(String[] args) {
      /* Se construye un objeto de tipo AGD, parametrizando
       * con la clase Rectángulo. Es decir, cada casilla del arreglo puede almacenar
       * la referencia a un objeto tipo Rectángulo.
       */
      AGD<Rectángulo> arre = new AGD( );
      Rectángulo r1 = new Rectángulo(4.5, 3.8);
      Rectángulo r2 = new Rectángulo(3.8, 4.2);
      Rectángulo r3 = new Rectángulo(5.0, 4.0);
      Rectángulo r4 = new Rectángulo(2.5, 3.1);

      // Se insertan 3 rectángulos en el arreglo.
      arre.insertaDesordenadoConRepetidos(r1);
      arre.insertaDesordenadoConRepetidos(r2);
      arre.insertaDesordenadoConRepetidos(r3);

      /* Se imprime el contenido del arreglo por medio del método toString( ) de la
       * clase AGD.
```

```
        */
        System.out.println("\nDatos de los rectángulos: " + arre);

        /* Se busca al rectángulo r2 en el arreglo, imprimiendo un mensaje según el
         * resultado. En  este caso, el rectángulo sí está en el arreglo.
         */
        if (arre.buscaSecuencialDesordenado(r2) >= 0)
           System.out.println("\nEl rectángulo: " + r2 + " está en el arreglo.");
        else
           System.out.println("\nEl rectángulo: " + r2 + " no está en el arreglo.");

        /* Se busca al rectángulo r4 en el arreglo, imprimiendo un mensaje según el
         * resultado. En este caso, el rectángulo no está en el arreglo.
         */
        if (arre.buscaSecuencialDesordenado(r4) >= 0)
           System.out.println("\nEl rectángulo: " + r4 + " está en el arreglo.");
        else
           System.out.println("\nEl rectángulo; " + r4 + " no está en el arreglo.");

        /* Se intenta eliminar el rectángulo r1, imprimiendo un mensaje de acuerdo
         * al resultado del método. En este caso, sí se elimina.
         */
        if (arre.eliminaDesordenado(r1))
           System.out.println("\nEl rectángulo: " + r1 + " se eliminó del arreglo.");
        else
           System.out.println("\nEl rectángulo: " + r1 + " no se eliminó del arreglo.");

        // Se imprime el contenido del arreglo, luego de la eliminación de r1.
        System.out.println("\nRectángulos luego de eliminar a r1: " + arre);
    }
}
```

Como ya se vio en secciones previas, ciertas operaciones varían según los elementos tengan o no orden entre sí. Por lo tanto, se considera conveniente definir otra clase para representar a arreglos genéricos cuyos elementos están ordenados crecientemente. La figura 5.3 presenta el diagrama de clase UML de la clase *ArregloGenéricoOrdenado*.

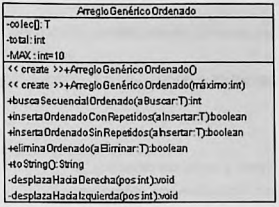


Figura 5.3 Diagrama de la clase *ArregloGenéricoOrdenado*

El programa 5.3 corresponde a la clase anterior y, para facilitar su uso, se renombra la clase como *AGO*. Observe que en el encabezado de la clase se establece que T debe extender a la interface Comparable de Java. De esta manera se obliga a que el tipo que se use al parametrizar T debe contar con un método *compareTo()*. Asimismo, como ya se mencionó anteriormente, deberá contar con el método *equals()*.

Programa 5.3 **AGO.java (Arreglo Genérico Ordenado)**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.3
 * Clase que define un arreglo genérico cuyos elementos están ordenados.
 * Se implementan las principales operaciones: búsqueda, inserción y eliminación.
 * AGO: Arreglo Genérico Ordenado
 */
public class AGO <T extends Comparable<T>> {
   private T colec[];
   private int total;
   private final int MAX = 10;

   /* Se instancia un arreglo de objetos que pueden ser comparados entre sí
    * y se convierte explícitamente a tipo T.
    */
   public AGO() {
      colec = (T[]) new Comparable[MAX];
      total = 0;
   }
```

```
// Se usa el parámetro para definir el tamaño máximo del arreglo
public AGO(int máximo) {
   colec = (T[]) new Comparable[máximo];
   total = 0;
}

/* Recibe un arreglo ya creado y copia sus elementos al
 * atributo correspondiente. Posteriormente ordena los
 * elementos del arreglo.
 */
public AGO(T[] arreglo, int total) {
   colec = (T[]) new Comparable[arreglo.length];
   for (int i = 0 ; i < total; i++)
      colec[i] = arreglo[i];
   this.total = total;
   ordenaSelecciónDirecta();
}

// Regresa el arreglo genérico
public T[] getColec( ) {
   return colec;
}

// Regresa el total de elementos del arreglo
public int getTotal( ) {
   return total;
}

/* Regresa el valor almacenado en la casilla índice. Si el índice está fuera de
 * rango, entonces lanza una excepción.
 */
public T getElemento(int índice){
   if (índice >= 0 && índice < total)
      return colec[índice];
   else
      throw new IndexOutOfBoundsException( );
}
```

```
/* Búsqueda secuencial en un arreglo genérico ordenado.
 * aBuscar es el elemento a buscar
 * Regresa la posición en la que lo encuentra o el negativo de la posición en la quemás 1 en la
 * que debería estar, en caso contrario.
 */
public int buscaSecuencialOrdenado(T aBuscar){
   int i;

   i = 0; // Para buscar desde la posición 0.

   while (i < total && colec[i].compareTo(aBuscar) < 0)
      i++;  // Se avanza al siguiente elemento.
   if (i == total || colec[i].compareTo(aBuscar) > 0)
      i = -(i + 1);
   return i;
}

/* Inserta un dato en un arreglo ordenado, sin permitir elementos repetidos.
 * aInsertar es el valor que quiere agregarse en el arreglo.
 * Regresa true si se pudo insertar (hay espacio y no se repite) y false en
 * caso contrario.
 */
public boolean insertaOrdenadoSinRepetidos(T aInsertar){
   boolean resp = false;
   if (total < colec.length){ // Hay espacio en el arreglo
      int pos = buscaSecuencialOrdenado(aInsertar);
      if (pos < 0){  // Si no está obtiene la posición en la que debe asignarlo.
         pos = -pos -1;
         desplazaHaciaDerecha(pos);
         colec[pos] = aInsertar;
         total = total + 1; // El arreglo tiene un elemento más.
         resp = true;
      }
   }
   return resp;
}
```

```
/* Método que inserta un dato en un arreglo genérico ordenado, permitiendo
 * elementos repetidos.
 * aInsertar es el valor que quiere agregarse en el arreglo, si hay espacio.
 * Regresa true si se puede insertar y false en caso contrario.
 */
public boolean insertaOrdenadoConRepetidos(T aInsertar){
   boolean resp = false;
   if (total < colec.length){ // Hay espacio en el arreglo
      int pos = buscaSecuencialOrdenado(aInsertar);
      if (pos < 0)  // Si no está obtiene la posición en la que debe asignarlo.
         pos = -pos -1;
      desplazaHaciaDerecha(pos);
      colec[pos] = aInsertar;
      total = total + 1; // El arreglo tiene un elemento más.
      resp = true;
   }
   return resp;
}

/* Método auxiliar que recorre todos los elementos del arreglo una posición a la
 * derecha, sin alterar el contenido de las casillas.
 */
private  void desplazaHaciaDerecha(int pos){
   int i;
   for (i = total; i > pos; i--)
      colec[i] = colec[i-1];
}

/* Elimina un elemento de un arreglo genérico ordenado.
 * aEliminar es el dato que se quiere eliminar.
 * Regresa como resultado true si el dato pudo eliminarse y false en caso contrario.
 */
public boolean eliminaOrdenado(T aEliminar){
   int pos;
   boolean resp = false;
```

```
        pos = buscaSecuencialOrdenado(aEliminar);
        if (pos >= 0){ // El dato está en el arreglo
            desplazaHaciaIzquierda(pos);
            colec[total-1] = null;
            total = total - 1; // Disminuye en 1 el total de elementos
            resp = true;
        }
        return resp;
    }

    /* Método auxiliar que recorre todos los elementos del arreglo una posición a la
     * izquierda, sin alterar el contenido de las casillas.
     */
    private void desplazaHaciaIzquierda(int pos){
        int i;
        for (i = pos; i < total - 1; i++)
            colec[i] = colec[i+1];
    }

    // Regresa una cadena formada con el contenido del arreglo genérico.
    public String toString(){
        int i;
        StringBuilder cad = new StringBuilder();

        for (i = 0; i < total; i++)
            cad.append(colec[i] + " ");
        return cad.toString();
    }
}
```

En el programa 5.5 se presenta un ejemplo de uso de arreglos genéricos ordenados. Se utiliza la clase anterior parametrizada con la clase *Cliente* (programa 5.4). Observe que esta última clase implementa la interface Comparable de Java y, por lo tanto, tiene un método *compareTo()* que permite comparar dos clientes y determinar cuál es mayor, menor o igual, según sus nombres.

Programa 5.4 **Cliente.java**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.4
 * Clase que define el concepto "Cliente". Se usará para probar un arreglo genérico
 * ordenado. Por lo tanto, esta clase debe implementar la interface Comparable de
 * Java, la cual obliga a implementar el método compareTo( ).
 */
public  class Cliente implements Comparable <Cliente>{
   String nombre;
   String domicilio;
   double saldo;

   public  Cliente( ){
   }

   public  Cliente(String  nom, String dom, double sal){
      nombre = nom;
      domicilio = dom;
      saldo = sal;
   }

   /* Sólo se recibe el nombre. Es muy útil para crear objetos auxiliares para
    * realizar comparaciones, tanto con el equals( ) como con el compareTo( ).
    */
   public  Cliente(String  nom){
      nombre = nom;
   }

   public  String  toString ( ){
      return "\nNombre: "+ nombre + "\nDomicilio: " + domicilio + "\nSaldo: " + saldo;
   }

   public String getNombre( ){
```

```
        return nombre;
    }

    public String getDomicilio() {
        return domicilio;
    }

    public void setDomicilio(String domicilio) {
        this.domicilio = domicilio;
    }

    public double getSaldo(){
        return saldo;
    }

    public void setSaldo(double sal){
        saldo = sal;
    }

    /* Sobrecarga del método para determinar si el cliente que invocó al método es
     * mayor, menor o igual que el "otro" cliente. El orden será alfabético, dado
     * por el nombre de los clientes.
     */
    public int compareTo(Cliente otro) {
        return nombre.compareTo(otro.nombre);
    }

    // Dos clientes son iguales si tienen el mismo nombre. El método se sobrescribe.
    public boolean equals(Object otro){
        boolean resp = false;
        if (otro != null && otro instanceof Cliente)
            resp = nombre.equalsIgnoreCase(((Cliente)otro).nombre);
        return resp;
    }
}
```

Programa 5.5 **EjemploAGO.java**

```
package cap5;

import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Programa 5.5
 * Ejemplo de uso de un objeto tipo AGO (ArregloGenéricoOrdenado). Se utiliza la
 * clase Cliente (programa 5.4) para darle valor a T.
 */
public class EjemploAGO {

   public static void main(String[ ] args) {
      // Se construye un objeto tipo AGO parametrizando con Cliente.
      AGO <Cliente> arre = new AGO( );

      Cliente c1 = new Cliente("Martínez, Lucía", "Reforma 128", 0.0);
      Cliente c2 = new Cliente("Alvarez, Marcos", "Calle 10 - 432", 3728.45);
      Cliente c3 = new Cliente("De Luca, Alicia", "Av. Paz 350", 1500.0);
      Cliente c4 = new Cliente("Suárez, Dolores", "Encinos 350", 0.0);

      // Se insertan tres clientes en el arreglo, de manera ordenada, por nombres.
      arre.insertaOrdenadoSinRepetidos(c1);
      arre.insertaOrdenadoSinRepetidos(c2);
      arre.insertaOrdenadoSinRepetidos(c3);

      // Se imprime el contenido del arreglo.
      System.out.println("\nDatos de los clientes: " + arre);

      /* Se busca un cliente, indicando si está o no registrado. En este caso
       * sí está registrado.
       */
      if ( arre.buscaSecuencialOrdenado(c1) >= 0)
         System.out.println("\n" + c1.getNombre( ) + " sí está registrado ");
      else
```

```
            System.out.println("\n" + c1.getNombre( ) + " no está registrado.");

        /* Se busca un cliente, indicando si está o no registrado. En este caso
         * no está registrado.
         */
        if (arre.buscaSecuencialOrdenado(c4) >= 0)
            System.out.println("\n" + c4.getNombre( ) + " sí está registrado");
        else
            System.out.println("\n" + c4.getNombre( ) + " no está registrado.");

        /* Se intenta eliminar un cliente, dando sólo su nombre. Imprimiendo un
         * mensaje de acuerdo al resultado del método. En este caso, sí se elimina.
         */
        if (arre.eliminaOrdenado(new Cliente("Martínez, Lucía")))
            System.out.println("\nCliente dado de baja.");
        else
            System.out.println("\nEse cliente no está registrado.");
    }
}
```

Se sugiere al lector revisar el código y probarlo. También es importante revisar el código de la clase *Cliente* (programa 5.4) en la cual se implementaron los métodos *compareTo()* y *equals()*.

5.3 ARREGLOS POLIMÓRFICOS

Con las dos clases definidas para manejar arreglos genéricos se pueden también crear arreglos polimórficos. Como se vio en el capítulo 3, si al instanciar una variable usando un clase genérica no se parametriza, entonces dicha variable puede funcionar como una variable polimórfica, ya que podrá cambiar de forma (referenciar a datos de distintos tipos) durante la ejecución. En el caso de los arreglos, estos podrán almacenar datos de distintos tipos en cada una de sus casillas.

Asimismo, se pueden definir arreglos polimórficos por medio de interfaces o de herencia, tema que se estudió en el capítulo 3. Con el uso de estos recursos, el tipo a asignar en el arreglo se acota dentro de lo permitido por la interface o por la herencia respectivamente, ya que, en el primer caso, sólo se podrán asignar objetos de clases que implementen a la interface; y, en el segundo, objetos de las clases derivadas de la clase con la que se haya parametrizado.

En el programa 5.6 se presenta un ejemplo del uso de arreglos polimórficos implementados por medio de interfaces. El arreglo es del tipo de la interface Figura, por lo tanto cada una de las casillas puede almacenar objetos de cualquiera de las clases que implementan dicha interface, mismas que son: *Triángulo, Rectangulo* y

Cuadrado (la interface y las clases pueden encontrarse en el paquete cap3 del proyecto EstructurasDatosBásicas que complementa este libro).

Programa 5.6 **EjemploArrePolimInterface.java**

```
package cap5;
import cap3.Figura;
import cap3.Cuadrado;
import cap3.Triángulo;
import cap3.Rectangulo ;

/**
 * @author Silvia Guardati
 * Programa 5.6
 * Ejemplo de arreglo polimórfico usando interfaces. Se retoma la interface Figura
 * del capítulo 3, así como las clases: Triángulo, Rectangulo  y Cuadrado.
 */
public class EjemploArrePolimInterface {

  public static void main(String[ ] args) {
     final int MAX = 10;
     /* Se construye un arreglo de tipo Figura, por lo tanto podrá almacenar objetos
      * de cualquiera de las clases que implementa a la interface.
      */
     Figura figurasGeom[ ] = new Figura[MAX];

     // Se almacenan algunos triángulos, cuadrados y rectángulos.
     figurasGeom[0] = new Triángulo(3.5, 3.5, 3.5);
     figurasGeom[1] = new Triángulo(4.2, 3.5, 3.2);
     figurasGeom[2] = new Triángulo(5.5, 5.5, 4.5);
     figurasGeom[3] = new Triángulo(6.3, 6.3, 6.3);
     figurasGeom[4] = new Cuadrado(8.5);
     figurasGeom[5] = new Cuadrado(6.2);
     figurasGeom[6] = new Cuadrado(7.0);
     figurasGeom[7] = new Rectangulo (6.0, 8.5);
     figurasGeom[8] = new Rectangulo(9.6, 12.0);
     figurasGeom[9] = new Rectangulo(3.3, 2.5);
```

```
        // Obtiene e imprime el total de cuadrados.
        System.out.println("\nTotal de cuadrados: " + cuentaCuadrados(figurasGeom, 10));

        // Imprime el área de todos los rectángulos.
        imprimeAreaDe(figurasGeom, 10, "Rectangulo");

        // Imprime el área de todos los triángulos.
        imprimeAreaDe(figurasGeom, 10, "Triángulo");
    }

    /* Método que imprime el área de la figura geométrica indicada por el
     * parámetro "clase".
     */
    public static void imprimeAreaDe(Figura arre[ ], int n, String clase){
        int i;

        System.out.println("\n\nImpresión de las áreas de las figuras tipo: " + clase);
        for (i = 0; i < n; i++)
            if (arre[i].getClass( ).getSimpleName( ).equals(clase))
                System.out.printf("%6.2f ", arre[i].calculaÁrea( ));
    }

    // Método que cuenta el total de cuadrados almacenados en el arreglo.
    public static int cuentaCuadrados(Figura arre[ ], int n){
        int i, cont;

        cont = 0;
        for (i = 0; i < n; i++)
            if (arre[i] instanceof Cuadrado)
                cont++;
        return cont;
    }
}
```

En el programa 5.7 se presenta un ejemplo del uso de arreglos polimórficos implementados por medio de herencia. Se declara un arreglo de tipo CuentaBancaria, por lo tanto cada casilla puede almacenar objetos de las dos subclases de ésta (las clases pueden encontrarse en el paquete cap2 del proyecto EstructurasDatosBásicas que complementa este libro).

Programa 5.7 **EjemploArrePolimHerencia.java**

```
package cap5;
import cap2.CuentaBancaria;
import cap2.CuentaAhorro;
import cap2.CuentaCheques;

/**
 * @author Silvia Guardati
 * Programa 5.7
 * Ejemplo de un arreglo polimórfico. Se usan las clases CuentaBancaria, CuentaAhorro
 * y CuentaCheques del capítulo 2.
 */
public class EjemploArrePolimHerencia {

  public static void main(String[] args) {

    final int MAX = 10;
    // Se construye un arreglo de CuentaBancaria
    CuentaBancaria cuentas[] = new CuentaBancaria[MAX];

    // Se asignan objetos usando las dos subclases de CuentaBancaria
    cuentas[0] = new CuentaAhorro("Juan del Campo", 5000, 5);
    cuentas[1] = new CuentaAhorro("Adriana García", 18000, 5.5);
    cuentas[2] = new CuentaCheques("Martín Soto", 5000);
    cuentas[3] = new CuentaAhorro("Martina Suculini", 9500, 4.8);
    cuentas[4] = new CuentaCheques("Gabriel Márquez", 17000);
    cuentas[5] = new CuentaCheques("Sofía Lorca", 6900);

    // Se imprime la información de todas las cuentas
    for (int i= 0; i < 6; i++)
      System.out.println(cuentas[i]);
```

```
        /* Se realiza un depósito en una de las cuentas. No se requiere conocer
         * de qué tipo de cuenta se trata, ya que el método depósito( ) es de la
         * súper clase y en el caso de la CuentaCheque está sobrescrito.
         */
        if (cuentas[1].depósito(1000))
          System.out.println("\nNuevo saldo: " + cuentas[1].getSaldo( ));
        else
          System.out.println("\nNo se pudo hacer el depósito");

        /* Se obtiene el interés generado por una de las cuentas de ahorro. Para
         * ello se debe verificar primero si la cuenta es una cuenta de ahorro. En
         * caso afirmativo, convirtiendo explícitamente a CuentaAhorro, se invoca
         * al método calculaInterés( ).
         */
        if (cuentas[3] instanceof CuentaAhorro){
          double interés = ((CuentaAhorro)cuentas[3]).calculaInterés( );
          System.out.println("\nInterés obtenido: " + interés);
        }
        else
          System.out.println("\nEsa cuenta no es de ahorro");
      }
    }
```

Por último, es importante mencionar que con la clase *Object* también se pueden declarar arreglos polimórficos. Lo que se explicó en el capítulo 3 para variables simples es aplicable para los arreglos.

Muy importante

- ☑ Se pueden definir arreglos polimórficos parametrizando con interfaces y asignando objetos de cualquiera de las clases que las implementen.
- ☑ Se pueden definir arreglos polimórficos parametrizando con súper clases y asignando objetos de cualquiera de las clases que las deriven.
- ☑ Se pueden definir arreglos polimórficos utilizando arreglos genéricos sin parametrizar.

5.4 OTRAS OPERACIONES

En esta sección se presentan dos operaciones sobre arreglos, adicionales a las vistas en el capítulo anterior. La primera de ellas permite ordenar un arreglo y la segunda es una alternativa a los algoritmos de búsqueda ya estudiados.

Ordenación por selección directa

En el capítulo anterior se presentaron operaciones para realizar sobre arreglos que tuvieran todos sus elementos ordenados. En estos casos se suponía que los elementos eran dados en orden. Sin embargo, no siempre es posible darlos ordenados, por lo que puede ser necesario ordenarlos antes de manipularlos.

Existen varios métodos diseñados para ordenar los elementos dentro de un arreglo, ya sea en orden creciente o decreciente. A continuación estudiaremos uno de ellos llamado método por selección directa. La idea es muy simple; consiste en encontrar el elemento más pequeño del arreglo y ponerlo en la primera posición intercambiándolo con el valor que estuviera ahí. Luego, tomar el más pequeño de los restantes –sabiendo que el primer elemento ya está ordenado– y asignarlo en la segunda posición, también intercambiándolo con el dato de esa posición. Ahora los dos primeros elementos están ordenados. Se sigue así hasta completar el arreglo. Se muestra a continuación el código de ordenación por selección directa para arreglos genéricos. El lector puede hacer los ajustes necesarios para adaptarlo a tipos primitivos si así lo requiere.

```
// Ordena aplicando el método de selección directa a un arreglo genérico.
  public void ordenaSelecciónDirecta( ){
    int i, j, pos;
    T menor;
    for (i = 0; i < total - 1; i++){
      menor = colec[i]; // Toma al elemento de la posición i como el "menor"
      pos = i; //
      /* Al terminar el ciclo interno, la variable "menor" almacena al dato más pequeño
       * y la variable "pos" su posición.
       */
      for (j = i + 1; j < total; j++)
        if (colec[j].compareTo(menor) < 0){
          menor = colec[j];
          pos = j;
        }
      // Realiza el intercambio entre el menor y el dato que ocupa la posición i.
      colec[pos] = colec[i];
      colec[i] = menor;
    }
  }
```

El algoritmo presentado ordena el contenido del arreglo de manera creciente. Para hacerlo de manera decreciente sólo se requiere cambiar la condición donde se usa el *compareTo()*.

Este método se incluye en la clase *AGO.java* (*Arreglo Genérico Ordenado*), programa 5.3, cuya versión completa está en el paquete cap5 del proyecto EstructurasDatosBásicas. Sin embargo, también se pudo definir como un método estático de la clase *OAG.java*, programa 4.3, para lo cual se deberán hacer algunos ajustes en la firma del método. Quedan a cargo del lector estos cambios.

En la figura 5.4 se presenta el seguimiento del método de selección directa para un arreglo de enteros. Primero se muestra el arreglo en su estado original; es decir, desordenado. En una tabla se despliegan las variables usadas en el algoritmo de ordenación, mostrando los valores que van tomando a medida que se ejecuta el método. A la derecha aparece el arreglo con los cambios realizados luego de terminar el ciclo que controla la variable j. Cuando este ciclo concluye, se tiene al valor más pequeño en la variable menor y su posición en la variable pos; por lo tanto, se procede al intercambio con el valor de la posición i. Observe que el ciclo externo se ejecuta hasta el total de elementos menos 1, ya que el interno empieza una posición a la derecha, por lo que en la última iteración queda todo el arreglo ordenado.

(a) Arreglo desordenado:

9	2	5	1	3	6		
0	1	2	3	4	5	6	7

i	menor	pos	j
0	9	0	
	2	1	1
			2
	1	3	3
			4
			5
			6

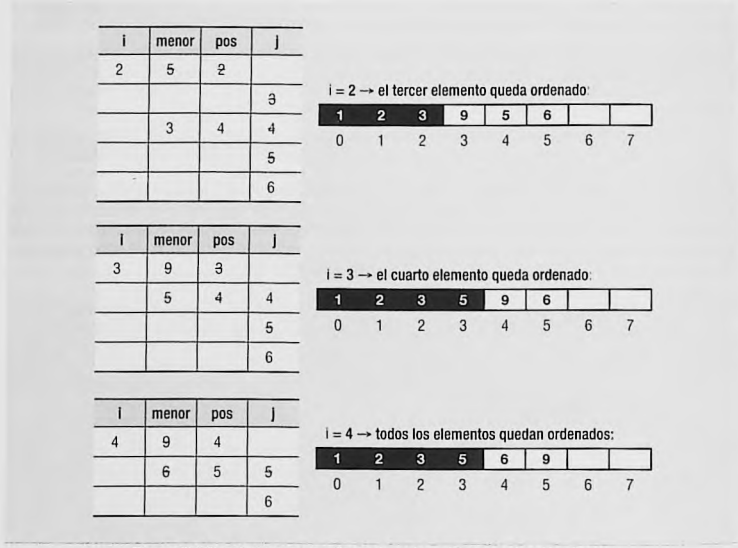
i = 0 → el primer elemento queda ordenado:

1	2	5	9	3	6		
0	1	2	3	4	5	6	7

i	menor	pos	j
1	2	1	
			2
			3
			4
			5
			6

i = 1 → el segundo elemento queda ordenado:

1	2	5	9	3	6		
0	1	2	3	4	5	6	7



i	menor	pos	j
2	5	2	
			3
	3	4	4
			5
			6

1	2	3	9	5	6		
0	1	2	3	4	5	6	7

i	menor	pos	j
3	9	3	
	5	4	4
			5
			6

1	2	3	5	9	6		
0	1	2	3	4	5	6	7

i	menor	pos	j
4	9	4	
	6	5	5
			6

1	2	3	5	6	9		
0	1	2	3	4	5	6	7

Figura 5.4 Seguimiento del método de ordenación por selección directa

Búsqueda binaria

Cuando se tiene la certeza de que el arreglo está ordenado se puede hacer uso de la búsqueda binaria, la cual es más eficiente que la secuencial ya analizada. Lógicamente este método se acerca a la manera en que las personas buscamos en un contexto en el cual hay orden, por ejemplo un directorio telefónico. Para buscar el teléfono de Pérez, Miguel seguramente no empezaríamos a buscar desde la "a", sino que lo haríamos desde la mitad del directorio aproximadamente. Observaríamos qué letra aparece y, en función de eso, volveríamos a "partir" el espacio de búsqueda para acercarnos al elemento buscado.

De manera similar, el algoritmo de este método plantea tomar, inicialmente, todo el arreglo como espacio de búsqueda. Es decir, el 0 como extremo izquierdo y el total de elementos -1 como el extremo derecho. Luego, obtener la mitad y comparar el dato buscado con el que ocupa esa posición. Si es igual, termina la búsqueda con éxito. En caso contrario puede ser mayor o menor. Si ocurre lo primero, entonces el espacio de búsqueda se redefine modificando el extremo izquierdo; ahora será la posición central más uno (ya que la posición central se descarta). Si fuera el otro caso, se redefine el extremo derecho asignádole la posición central menos uno. Cualquiera sea el caso presentado, el espacio de búsqueda se reduce y vuelven a repetirse los pasos: se calcula la posición central, se compara, etc. A continuación se presenta el algoritmo de la búsqueda binaria, el

cual se incluye en la clase *AGO* (*Arreglo Genérico Ordenado*), programa 5.3, cuya versión completa está en el paquete cap5 del proyecto EstructurasDatosBásicas. También se puede generar una versión del método para la clase *OAG* (programa 4.3), realizando algunos pequeños cambios en la firma del método. Quedan a cargo del lector estos cambios.

```
/* Búsqueda binaria. El arreglo debe estar ordenado.
 * aBuscar es el elemento a buscar en el arreglo.
 * Regresa la posición en la que lo encuentra o el negativo de la posición en la que
 * debería estar, más uno.
 */
public int buscaBinaria(T aBuscar){
  int izq, der, cen;

  izq = 0;
  der = total - 1;
  cen = (izq + der) / 2;
  // Busca mientras haya espacio de búsqueda y mientras no lo encuentre.
  while (izq <= der && !colec[cen].equals(aBuscar)){
    if (colec[cen].compareTo(aBuscar) < 0)
      izq = cen + 1;  // Recorre el extremo izquierdo.
    else
      der = cen - 1;  // Recorre el extremo derecho.
    cen = (izq + der) / 2;
  }
  if (izq > der)    // aBuscar no está en el arreglo.
    cen = -(izq + 1);
  return cen;
}
```

A continuación se presenta el seguimiento de la búsqueda binaria en un arreglo ordenado de 12 elementos. Se muestra en el arreglo, coloreando los índices correspondientes a los límites del espacio de búsqueda. Por ejemplo, en (a) se pinta de gris oscuro los extremos izquierdo y derecho, y de gris claro el centro. Más abajo, en la tabla 5.1, aparece el mapa de memoria con los distintos valores que van tomando las variables a medida que se avanza en la búsqueda. Se hace el seguimiento para 2 valores que están en el arreglo (éxito en la búsqueda) y para uno que no se ecuentra (fracaso).

(a) izq = 0 der = 11 cen = 5

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3	4	5	6	7	8	9	10	11

(b) izq = 0 der = 4 cen = 2

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3	4	5	6	7	8	9	10	11

(c) izq = 3 der = 4 cen = 3

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3	4	5	6	7	8	9	10	11

(d) izq = 6 der = 11 cen = 8

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3	4	5	6	7	8	9	10	11

(e) izq = 9 der = 11 cen = 10

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3	4	5	6	7	8	9	10	11

(f) izq = 11 der = 11 cen = 11

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3		5	6	7	8	9	10	11

(g) izq = 9 der = 9 cen = 9

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3	4	5	6	7	8	9	10	11

(h) izq = 10 der = 9 cen = 9

1	4	5	9	13	15	20	23	24	31	46	59
0	1	2	3	4	5	6	7	8	9	10	11

Figura 5.5 Seguimiento de la búsqueda binaria en un arreglo

Tabla 5.1 Seguimiento de la búsqueda binaria – mapa de memoria.

Izq	der	cen	aBuscar	total	Figura 5.5
0	~~11~~	5	9	12	(a)
	~~4~~	~~2~~			(b)
3		3			(c) termina con éxito (aBuscar = colec[cen])
0	11	~~5~~	59	12	(a)
~~6~~		~~8~~			(d)
~~9~~		~~10~~			(e)
11		11			(f) termina con éxito (aBuscar = colec[cen])
0		5	43	12	(a)
~~6~~		~~8~~			(d)
~~9~~		~~10~~			(e)
9	~~9~~	9			(g)
10		9			(h) termina con fracaso (izq>der)

Observe que en los casos en que se encuentra el elemento buscado (9 y 59), la posición de los mismos está dada por la variable cen, mientras que en el caso en que no está (43) la posición en que debería estar la indica la variable izq. Como en el caso de la búsqueda secuencial en arreglos ordenados, se regresa como resultado el negativo de la posición más uno. Recuerde que el negativo señala que no se encontró, y se le suma uno para que pueda convertirse a negativo en aquellos casos en que la posición sea cero.

5.5 ITERADORES Y ARREGLOS

Un iterador es un objeto que permite recorrer los elementos de una colección de datos sin necesidad de conocer la estructura interna de la misma. Por lo tanto, son muy útiles para generalizar ciertas operaciones sobre las estructuras de datos. En general, los iteradores tienen la funcionalidad necesaria para acceder a los elementos de la colección, así como para desplazarse dentro de la misma. En ciertos casos, pueden tener métodos adicionales.

La implementación puede variar de acuerdo al lenguaje usado. En Java se utilizan dos interfaces: Iterable e Iterator. La primera de ellas tiene un único método, el cual genera un objeto de tipo Iterator, garantizando así que la estructura que implementa dicha interface contará con un iterador sobre ella misma. Por su parte, la interface Iterator define comportamiento a través de tres métodos; el primero de ellos es para saber si hay algún elemento por visitar en la colección, el segundo para tener acceso a uno de los elementos y el último, que es opcional, para quitar un elemento. En la tabla 5.2 se listan las interfaces con sus métodos y una explicación de los mismos.

Tabla 5.2 Interfaces y métodos para el manejo de iteradores.

Interface	Método	Comentario
Iterable <T>	iterator( ): Iterator <T>	Regresa un objeto de tipo Iterator<T>.
Iterator <T>	hasNext( ): boolean	Regresa true si el iterador está sobre un elemento de la colección. En caso contrario, regresa false.
	next( ): T	Regresa el elemento apuntado por el iterador y desplaza el iterador al siguiente valor. Si ya no hubiera elementos, lanza la excepción NoSuchElementException.
	remove( ): void	Es una operación opcional. En caso de implementarse debe eliminar el último elemento dado por el iterador. Por lo tanto, sólo puede ejecutarse una vez por cada llamada al método next( ). Si no se implementa, lanza la excepción UnsupportedOperationException

Observe que se usan dos excepciones de Java para indicar situaciones especiales que pueden presentarse. La excepción *NoSuchElementException* se lanza cuando no existe un elemento que esté siendo apuntado por el iterador. En el segundo caso, la excepción *UnsupportedOperationException* señala que la operación solicitada no está implementada. Ambas excepciones son clases de Java.

Es importante destacar que los algoritmos usados para implementar las operaciones de los iteradores sí son dependientes de la estructura interna sobre la que se implementen. A continuación se presenta el código para iteradores sobre arreglos y en los siguientes capítulos se verán sobre otros tipos de estructuras de datos.

En el programa 5.8 se presenta el código de la clase *IteradorArreglo<T>*, la cual implementa la interface *Iterator<T>*, con lo cual contrae la obligación de implementar sus métodos. Junto a cada uno de los métodos se incluyen comentarios con la explicación de los mismos.

Programa 5.8 **IteradorArreglo.java**

```
package cap5;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * @author Silvia Guardati
 * Programa 5.8
 * Esta clase permite recorrer un arreglo elemento por elemento sin necesidad de
 * conocer la estructura interna del mismo.
 */
```

```
public class IteradorArreglo<T> implements Iterator<T>{
   private T colec[];
   private int actual;
   private int totalElem;

   /* Se construye un iterador sobre la colección dada como parámetro.
    * actual toma el valor de 0, así se empieza a iterar desde el primer
    * elemento de la colección.
    */
   public IteradorArreglo(T[] colec, int totalElem) {
      this.colec = colec;
      this.actual = 0;
      this.totalElem = totalElem;
   }

   // Regresa true si la colección tiene al menos un elemento más.
   public boolean hasNext() {
      return actual < totalElem;
   }

   // Regresa el elemento de la colección apuntado por actual y avanza el iterador.
   public T next() {
      if (hasNext())
         return colec[actual++];
      else
        throw new  NoSuchElementException();
   }

   /* Es una operación opcional. En caso de ser codificada debe regresar el último
    * valor dado por el iterador.
    */
   public void remove() {
      throw new UnsupportedOperationException("Operación no implementada");
   }
}
```

El programa 5.9, AGOIter.java, corresponde a una versión con iterador de la clase *AGO*. Se cambió el encabezado, agregando la implementación de la interface Iterable<T> la que exige tener el método *iterator()*, el que a su vez permite crear un objeto iterador sobre cada instancia del arreglo. Por razones de espacio no se incluye todo el código (mismo que se incluyó en la clase *AGO*), el cual puede consultarse en el paquete cap5,del proyecto EstructuraDatosBásicas.

Programa 5.9 **AGOIter.java**

```
package cap5;
import java.util.Iterator;
/**
 * @author Silvia Guardati
 * Programa 5.9
 * Clase que define un arreglo genérico cuyos elementos están ordenados.
 * Versión modificada del programa 5.3 para incluir el manejo de iteradores
 * sobre el arreglo.
 */
public class AGOIter <T extends Comparable<T>> implements Iterable<T>{
   private T colec[];
   private int total;
   private final int MAX = 10;

   /* Se instancia un arreglo de objetos que pueden ser comparados entre sí
    * y se convierte explícitamente a tipo T.
    */
   public AGOIter() {
      colec = (T[]) new Comparable[MAX];
      total = 0;
   }

   public AGOIter(int máximo) {
      colec = (T[]) new Comparable[máximo];
      total = 0;
   }

   /* Recibe un arreglo ya creado y copia sus elementos al
    * atributo correspondiente. Posteriormente ordena los
    * elementos del arreglo.
    */
```

```
public AGOIter(T[] arreglo, int total) {
   colec = (T[]) new Comparable[arreglo.length];
   for (int i = 0 ; i < total; i++)
      colec[i] = arreglo[i];
   this.total = total;
   ordenaSelecciónDirecta();
}

// Búsqueda secuencial en arreglo genérico ordenado.

// Inserta un dato en un arreglo ordenado, sin permitir elementos repetidos

/* Método que inserta un dato en un arreglo genérico ordenado, permitiendo
 * elementos repetidos.
 */
/* Método auxiliar que recorre todos los elementos del arreglo una posición a la
 * derecha, sin alterar el contenido de las casillas.
 */

// Elimina un elemento de un arreglo genérico ordenado.

/* Método auxiliar que recorre todos los elementos del arreglo una posición a la
 * izquierda, sin alterar el contenido de las casillas.
 */

// Ordena aplicando el método de selección directa a un arreglo genérico.

// Búsqueda binaria.

// Crea un iterador sobre el arreglo.
   public Iterator<T> iterator() {
      return new IteradorArreglo(colec, total);
   }

   /* Regresa una cadena formada con el contenido del arreglo genérico.
    * Se reemplazó el ciclo for por un ciclo while, usando el iterador
    * para recorrer todas las casillas del arreglo.
    */
```

```
    public String toString( ){
      int i;
      Iterator <T> it = iterator( ); // Se construye un iterador sobre el arreglo
      StringBuilder cad = new StringBuilder( );

      while (it.hasNext( ) // Mientras haya elementos en la colección
        cad.append(it.next( ) + " "); // Pega el dato que regresa el iterador
      return cad.toString( );
    }
}
```

En el programa 5.10 se presenta un ejemplo del uso de los iteradores para recorrer una colección de datos sin necesidad de conocer su estructura interna. En este programa se despliega la información de un objeto tipo AGOIter por medio del método toString(), por medio del ciclo for all (usado para recorrer los elementos de una colección) y por medio de un iterador.

Programa 5.10 | **EjemploAGOIter.java**

```
package cap5;

import java.util.Iterator;

/**
 * @author Silvia Guardati
 * Programa 5.10
 * Ejemplo del uso de iteradores sobre arreglos.
 */
public class EjemploAGOIter {

  public static void main(String[ ] args) {
    /* Se construye un objeto tipo arreglo genérico ordenado -con iterador-,
     * parametrizado con la clase Cliente.
     */
    AGOIter <Cliente> arreCli = new AGOIter( );
    boolean resp;
    double suma;

    // Se construyen 3 objetos de tipo Cliente y se insertan en el arreglo.
    Cliente c1 = new Cliente("Rubalcaba, Juan", "Reforma 128", 1000);
    Cliente c2 = new Cliente("Martínez, Alicia", "García Lorca 584", 8500);
```

```
        Cliente c3 = new Cliente("Gómez, Marina", "Estrella 324", 3729.5);
        resp = arreCli.insertaOrdenadoSinRepetidos(c1);
        if (!resp)
           System.out.println("\nNo se pudo dar de alta el cliente: " + c1);
        resp = arreCli.insertaOrdenadoSinRepetidos(c2);
        if (!resp)
           System.out.println("\nNo se pudo dar de alta el cliente: " + c2);
        resp = arreCli.insertaOrdenadoSinRepetidos(c3);
        if (!resp)
           System.out.println("\nNo se pudo dar de alta el cliente: " + c3);

        /* Impresión del contenido del arreglo por medio del toString de la clase
         * AGOIter.
         */
        System.out.println("\nLista de clientes -toString:\n" + arreCli);

        // Impresión del contenido del arreglo por medio del ciclo for all
        System.out.println("\nLista de clientes -for all");
        for (Cliente c: arreCli)
           System.out.println(c);

        /* Impresión del contenido del arreglo por medio del iterador. Observe que
         * al construir el objeto it de tipo Iterator se parametriza con el mismo
         * tipo usado en el arreglo.
         */
        System.out.println("\nLista de clientes -iterador");
        Iterator <Cliente> it = arreCli.iterator();
        while (it.hasNext())
           System.out.println(it.next());

        /* Obtiene e imprime la suma de los saldos de todos los clientes. Cada uno
         * de los saldos se obtiene a partir del iterador; en este caso es fundamental
         * parametrizar con la clase Cliente, si no se debería convertir explícitamente
         * antes de obtener el saldo.
         */
        it = arreCli.iterator();
        suma = 0;
        while (it.hasNext())
           suma = suma + it.next().getSaldo();
        System.out.println("\nSaldo total: $" + suma);
     }
  }
```

Muy importante

- ☑ Los iteradores permiten tener acceso a la información sin tener acceso a la estructura interna del arreglo.
- ☑ Los métodos hasNext() y next() se deben programar de acuerdo a la estructura de datos sobre la cual van a trabajar.
- ☑ El método remove() es opcional.
- ☑ El objeto tipo Iterator se debe parametrizar si se necesita conservar el tipo del dato almacenado en el arreglo.
- ☑ Se usan las interfaces de Java Iterable e Iterator.

Con los métodos hasNext() y next() se pudo revisar el contenido del arreglo sin tener que acceder a cada una de sus casillas. En realidad, el acceso a la información fue independiente de la estructura de datos usada. En los siguientes capítulos se volverá sobre el uso de los iteradores.

5.6 ARREGLOS MULTIDIMENSIONALES

Los arreglos vistos hasta el momento permiten representar datos en una sola dimensión, por ejemplo una calificación para un grupo de n alumnos, o bien las m calificaciones de un alumno. Sin embargo, si se tienen muchos datos de más de una dimensión, entonces se necesitaría trabajar con muchos arreglos, lo cual dificultaría la solución de los problemas.

Para manejar situaciones como éstas se utilizan los arreglos multidimensionales. En particular, en esta sección se presentan los arreglos bidimensionales o matrices; es decir, aquellos que tienen información sobre dos dimensiones. Siguiendo con los alumnos y las calificaciones, para almacenar las m calificaciones de un grupo de n alumnos, un arreglo bidimensional es la estructura de datos adecuada. Gráficamente, un arreglo bidimensional se representa como se muestra en la figura 5.6.

Los arreglos, independientemente de la cantidad de dimensiones que tengan, tienen las características ya mencionadas para los arreglos de una dimensión. Es decir, son estructuras estáticas, finitas, homogéneas y ordenadas de datos. Al momento de crear un arreglo multidimensional se debe indicar el máximo valor para cada una de las dimensiones. En el caso particular de los arreglos bidimensionales, primero se establece el máximo para los renglones y después el correspondiente a las columnas.

El arreglo tiene un nombre único y cada uno de sus elementos se identifica por el nombre seguido de dos índices, el primero de ellos para renglones y el segundo para columnas. Los renglones y las columnas se enumeran de 0 en adelante. Como en el caso de los arreglos unidimensionales, los índices se escriben entre corchetes:

```
nombreArreglo[índiceRenglón] [índiceColumna]
```

Siendo los índices valores comprendidos entre 0 y el máximo definido para los renglones y entre 0 y el máximo definido para las columnas, respectivamente.

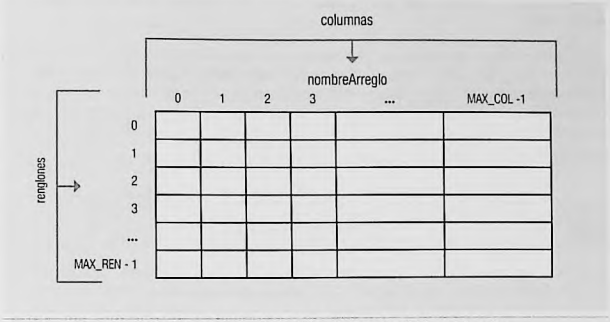


Figura 5.6 Representación gráfica de arreglos bidimensionales

La mayoría de las operaciones se aplica a cada una de las casillas del arreglo. Se manipula todo el arreglo cuando se asigna a otra variable del mismo tipo o cuando se pasa como parámetro a un método. A continuación se presentan las operaciones más comunes en arreglos bidimensionales.

5.6.1 Declaración e instanciación de arreglos bidimensionales

La declaración e instanciación de un arreglo se puede hacer en uno o en dos pasos. Observe el siguiente código.

```
// Declaración e instanciación
tipo nombreArreglo[ ][ ] = new tipo [MAX_RENGLÓN][MAX_COLUMNA];
                                    o
// Declaración
tipo nombreArreglo[ ][ ];
// Instanciación
nombreArreglo = new tipo [MAX_RENGLÓN][MAX_COLUMNA];
```

Donde:

- tipo es cualquier tipo de dato válido en Java. Determina el tipo de dato que almacenará el arreglo.
- Los [][] junto a nombreArreglo pueden ir a su derecha o a su izquierda.

- MAX_RENGLÓN indica la cantidad máxima de renglones que tendrá el arreglo. Los renglones se enumeran a partir de 0.
- MAX_COLUMNA indica la cantidad máxima de columnas que tendrá el arreglo. Las columnas se enumeran a partir de 0.

Una vez declarado el arreglo se reserva en memoria tanto espacio como se requiera para almacenar MAX_RENGLÓN x MAX_COLUMNA elementos del tipo de dato especificado. Sin embargo, esto no quiere decir que necesariamente todas las casillas deban estar ocupadas. El problema determina el total de elementos con los que se trabajará. Por lo tanto, es importante destacar que, en general, cuando se tiene un arreglo bidimensional también se tiene el total de renglones y el total de columnas que se ocupan.

Si se desea conocer el máximo número de renglones reservados para un arreglo bidimensional se puede usar:

```
nombreArreglo.length
```

En cambio, si se quiere conocer el máximo número de columnas reservadas primero se debe hacer referencia a uno de los renglones (no necesariamente el 0). Observe la siguiente sintaxis:

```
nombreArreglo[0].length
```

Ejemplo 5.1

El código de estos ejemplos puede encontrarse en el programa PruebaArreglosBidim.java (puede consultarse en el proyecto EstructurasDatosBásicas, paquete cap5, que complementa este libro). Se sugiere al lector que lo analice y lo pruebe para reafirmar los conceptos estudiados en esta sección.

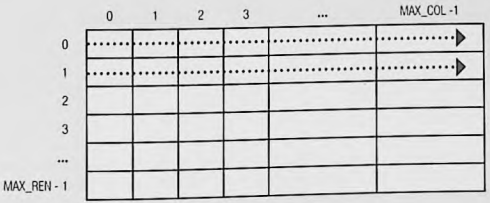
```
// Declaración de constantes
final int MAX_ALUM = 30;
final int MAX_EXAM = 4;

/* Declaración e instanciación de un arreglo de 30 renglones y 4 columnas.
 * Se usan las constantes previamente declaradas para dar el tamaño máximo.
 */
int calif[][] = new int[MAX_ALUM][MAX_EXAM];

// Declaración de un arreglo tipo double
double [][]prodPesq;
// Instanciación del arreglo. Se usan números para establecer el límite.
prodPesq = new double[20][12];
```

```
/* Declaración, instanciación e inicialización de un arreglo bidimensional
 * de tipo boolean. El tamaño queda determinado por la cantidad de valores
 * asignados. En este caso, 2 renglones y 3 columnas.
 */
boolean mat[][] = {{true, true, true}, {false, true, false}};

// Imprime el máximo de renglones reservados para calif (30)
System.out.println("\nTotal de renglones - primer arreglo: " + calif.length);
// Imprime el máximo de columnas reservadas para calif (4)
System.out.println("\nTotal de columnas - primer arreglo: " + calif[0].length);

// Imprime el máximo de renglones reservados para mat(2)
System.out.println("\nTotal de renglones - tercer arreglo: " + mat.length);
// Imprime el máximo de columnas reservadas para mat (3)
System.out.println("\nTotal de columnas - tercer arreglo: " + mat[1].length);
```

5.6.2 Lectura, impresión e inicialización de arreglos bidimensionales

Considerando que los arreglos bidimensionales son una generalización de los arreglos unidimensionales, para leer, imprimir o inicializar estos arreglos se requerirá un ciclo por cada una de las dimensiones. El orden en el cual se muevan los índices determinará la manera en que se visitarán los elementos del arreglo. Es decir, los arreglos podrán recorrerse por renglón o por columna. Si es el primer caso, para cada renglón se recorren todas las columnas; mientras que si es el otro caso, para cada columna se visitan todos los renglones.

a. Por renglones: para el renglón 0 se recorren todas las columnas, luego para el renglón 1 se recorren todas las columnas, y así hasta el final.



b. Por columnas: para la columna 0 se recorren todos los renglones, luego para la columna 1 se recorren todos los renglones, y así hasta el final.

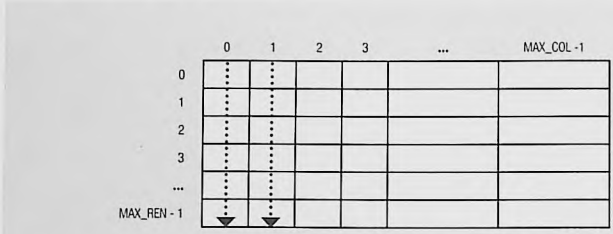


Figura 5.7 Recorrido de arreglos bidimensionales

Como ya se mencionó, para recorrer todo el arreglo (para leer, imprimir o asignar) se necesitan dos ciclos, uno para los renglones y otro para las columnas. Los ciclos se anidan dependiendo del recorrido. Si es por renglón, entonces el ciclo externo es el de los renglones y el interno el de las columnas. En cambio, si el recorrido es por columna, el ciclo externo será el de las columnas y el interno el de los renglones. Observe el siguiente código, el cual se encuentra en el programa PruebaArreglosBidim.java mencionado más arriba.

```
// Lectura de un arreglo bidimensional (por renglón)
for (i= 0; i < numR; i++)
    for (j= 0; j < numC; j++){
        System.out.println("Calificación: " + (j+1) + " del alumno " + (i+1));
        calif[i][j] = lee.nextInt( );
    }
```

La tabla 5.3 es el mapa de memoria correspondiente al código anterior. La variable i toma el valor 0, luego j también toma el valor 0 y se lee un dato que se almacena en calif[0][0], posteriormente j se incrementa y el dato leído se guarda en calif[0][1] y así hasta que j alcance el valor de numC-1, que es cuando el ciclo interno se interrumpe. Se sigue en el ciclo externo, ahora con i = 1, y vuelve a ejecutarse el ciclo interno (con j desde 0 hasta numC-1), así hasta que la i llegue a numR-1.

Tabla 5.3 Seguimiento de la lectura de arreglos bidimensionales –por renglón.

I	J	Lee y asigna en:	Comentario
0	0	calif[0][0]	Renglón 0, columna 0
	1	calif[0][1]	Renglón 0, columna 1
	2	calif[0][2]	Renglón 0, columna 2
	...	...	...
1	0	calif[1][0]	Renglón 1, columna 0
	1	calif[1][1]	Renglón 1, columna 1
	2	calif[1][2]	Renglón 1, columna 2
	...	...	...

La lectura presentada más arriba se lleva a cabo por renglones. Como se muestra en la tabla 5.3, para cada renglón se visitan todas las columnas (antes de pasar al siguiente renglón). A continuación se presenta el código correspondiente a la impresión por renglón de un arreglo. El código se encuentra en el programa PruebaArreglosBidim.java.

```
// Impresión de un arreglo bidimensional (por renglón)
System.out.println("\n\nArreglo bidimensional:\n");
for (i= 0; i < 2; i ++){
    for (j= 0; j < 3; j++)
        System.out.print(mat[i][j] + " ");
    System.out.println("\n");
}
```

En este ejemplo se imprimen todos los valores de un renglón (para ello se recorren todas las columnas) y se salta a la siguiente línea. De esta manera, se logra que la impresión tenga "forma" rectangular. Por último, se da un ejemplo de asignación de un valor a todos los elementos de un arreglo bidimensional. En este caso, el recorrido se hace por columna.

```
// Asignación de 0 a todos los elementos de un arreglo bidimensional (por columna)
for (j= 0; j < 12; j++)
    for (i= 0; i < 20; i++)
        prodPesq[i][j] = 0;
```

La tabla 5.4 muestra el mapa de memoria correspondiente al código anterior. La variable j toma el valor 0, luego i también toma el valor 0 y se asigna un dato en prodPesq[0][0], posteriormente i se incrementa y el dato se guarda en prodPesq[1][0] y así hasta que i alcance el valor de 20, que es cuando el ciclo interno se interrumpe sin llegar a asignar. Se sigue en el ciclo externo, ahora con j = 1 y vuelve a ejecutarse el ciclo interno (con i desde 0 hasta 19 inclusive), así hasta que la j llegue a 12, interrumpiéndose el ciclo.

Tabla 5.4 Seguimiento de asignación de arreglos bidimensionales –por columna.

j	i	Asigna en:	Comentario
0	0	prodPesq [0][0]	Columna 0, renglón 0
	1	prodPesq [1][0]	Columna 0, renglón 1
	2	prodPesq [2][0]	Columna 0, renglón 2
	...	...	...
1	0	prodPesq [0][1]	Columna 1, renglón 0
	1	prodPesq [1][1]	Columna 1, renglón 1
	2	prodPesq [2][1]	Columna 1, renglón 2
	...	...	...

En las operaciones presentadas más arriba se recorre todo el arreglo, al menos los renglones y las columnas acotados por los límites establecidos, que no siempre coinciden con los máximos usados para la instanciación. Sin embargo, hay problemas en los cuales sólo se necesita visitar un renglón o una columna. En la siguiente sección se analizarán algunas operaciones que, por su utilidad, merecen especial atención.

5.6.3 Otras operaciones con arreglos bidimensionales

Las operaciones más frecuentes y requeridas son aquellas que se aplican sobre un cierto renglón o sobre una cierta columna. Por ejemplo, retomando el caso de las calificaciones de un grupo de alumnos obtenidas en varios exámenes, puede ser muy útil calcular el promedio de las calificaciones de un alumno o de uno de los exámenes, o encontrar la calificación más alta de un alumno o de un examen. En todos estos ejemplos, se debe recorrer sólo un renglón (el del alumno en cuestión) o una columna (la del examen que interesa). A continuación se presentan estas operaciones, cada una de ellas codificada como un método estático. Estas operaciones se presentan en el programa 5.11, OABE.java (Operaciones con Arreglos Bidimensionales de

Enteros). Al igual que con los arreglos unidimensionales, no todas las operaciones son posibles para todos los tipos de datos. Por lo tanto, habría que hacer los ajustes necesarios para adecuar esta clase según el tipo de dato almacenado en el arreglo.

Programa 5.11 **OABE.java**

```
package cap5;

import java.util.Scanner;

/**
 * @author Silvia Guardati
 * Programa 5.11
 * Clase que concentra las operaciones más frecuentes sobre un arreglo bidimensional
 * de enteros.
 * OABE: Operaciones con Arreglos Bidimensionales de Enteros.
 */
public class OABE {

  /* Método auxiliar utilizado para leer y validar tanto el número de renglones
   * como el de columnas.
   * Recibe el máximo permitido, dado por el valor con el cual se instanció al
   * arreglo.
   */
  public static int leeTamaño(int max){
    Scanner lee = new Scanner(System.in);
    int n;

    do{
      System.out.println("\nDebe ser un entero positivo menor que: " + max);
      n = lee.nextInt();
    } while (n <= 0 || n > max);
    return n;
  }

  // Lee -por renglón- un arreglo bidimensional
  public static void leeArre(int arre[][], int numR, int numC){
```

```
    Scanner lee = new Scanner(System.in);
    int i,j;

    for (i = 0; i < numR; i++)
       for (j = 0; j < numC; j++){
          System.out.print("\nIngresa dato: ");
          arre[i][j] = lee.nextInt( );
       }
  }

  // Imprime -por renglón- un arreglo bidimensional
  public static void imprimeArre(int arre[ ][ ], int numR, int numC){
    Scanner lee = new Scanner(System.in);
    int i,j;

    for (i = 0; i < numR; i++){
       for (j = 0; j < numC; j++)
          System.out.print(arre[i][j] + " ");
       System.out.println("\n");
    }
  }

  // Obtiene y regresa la suma del renglón dado como parámetro (ren).
  public static int sumaRenglón(int arre[ ][ ], int ren, int numC){
    int suma, c;

    suma = 0;
    for(c = 0; c < numC; c++)
       suma = suma + arre[ren][c];
    return suma;
  }

  // Calcula y regresa el promedio del renglón dado como parámetro (ren).
  public static double calculaPromRen(int arre[ ][ ], int ren, int numC){
    return (double)sumaRenglón(arre, ren, numC) / numC;
  }
```

```
// Obtiene y regresa la suma de la columna dada como parámetro (col).
public static int sumaColumna(int arre[][], int col, int numR){
   int suma, r;

   suma = 0;
   for(r = 0; r < numR; r++)
      suma = suma + arre[r][col];
   return suma;
}

// Calcula y regresa el promedio de la columna dada como parámetro (col).
public static double calculaPromCol(int arre[][], int col, int numR){
   return (double)sumaColumna(arre, col, numR) / numR;
}

/* Busca y regresa la posición del elemento más grande de un renglón dado
 * como parámetro (ren). La posición es la columna en la que encuentra al mayor.
 */
public static int buscaPosMayorRenglón(int arre[][], int ren, int numC){
   int c, col;

   col = 0;
   for (c = 1; c < numC; c++)
      if (arre[ren][c] > arre[ren][col])
         col = c;
   return col;
}

/* Busca y regresa la posición del elemento más grande de una columna dada
 * como parámetro (col). La posición es el renglón en el que encuentra al mayor.
 */
public static int buscaPosMayorColumna(int arre[][], int col, int numR){
   int r, ren;

   ren = 0;
   for (r = 1; r < numR; r++)
```

```
            if (arre[r][col] > arre[ren][col])
                ren = r;
        return ren;
    }

    /* Cuenta y regresa el total de veces que el dato se encuentra en el renglón
     * dado como parámetro (ren).
     */
    public static int cuentaPorRenglón(int arre[][], int ren, int numC, int dato){
        int c, cont;

        cont = 0;
        for (c = 0; c < numC; c++)
            if (arre[ren][c] == dato)
                cont = cont + 1;
        return cont;
    }

    /* Cuenta y regresa el total de veces que el dato se encuentra en la columna
     * dada como parámetro (col).
     */
    public static int cuentaPorColumna(int arre[][], int col, int numR, int dato){
        int r, cont;

        cont = 0;
        for (r = 0; r < numR; r++)
            if (arre[r][col] == dato)
                cont = cont + 1;
        return cont;
    }
}
```

En el programa 5.12 se usa la clase anterior para resolver un problema. Se retoma el caso de las calificaciones de un grupo de alumnos que han presentado varios exámenes. Con esos datos se genera información útil para tomar decisiones; por ejemplo, el promedio de cada alumno y de cada examen, así como el examen en el cual cada alumno obtuvo su calificación más alta.

Programa 5.12 **UsaOABE.java**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.12
 * Ejemplo del uso de la clase OABE para procesar las calificaciones de un grupo
 * de alumnos. Las calificaciones se almacenan en un arreglo bidimensional y mediante
 * los métodos de OABE se obtiene la información requerida.
 */
public class UsaOABE {
   public static void main(String[ ] args) {
      int numAl, numEx, al, ex, res;
      double prom;
      // Declaración de constantes
      final int MAX_ALUM = 30;
      final int MAX_EXAM = 4;

      /* Declaración e instanciación de un arreglo de 30 renglones y 4 columnas.
       * Se usan las constantes previamente declaradas para dar el tamaño máximo.
       */
      int calif[ ][ ] = new int[MAX_ALUM][MAX_EXAM];

      System.out.println("\nIngresa el total de alumnos");
      numAl = OABE.leeTamaño(MAX_ALUM);
      System.out.println("\nIngresa el total de exámenes");
      numEx = OABE.leeTamaño(MAX_EXAM);

      // Lectura del arreglo de calificaciones
      System.out.println("\nIngresa las calificaciones, por alumno.");
      OABE.leeArre(calif, numAl, numEx);

      // Imprime el promedio obtenido por cada alumno
      for (al = 0; al < numAl; al++){
         prom = OABE.calculaPromRen(calif, al, numEx);
         System.out.println("\nEl promedio del alumno: " + (al+1) + " es: " + prom);
```

```
        }

        // Imprime el promedio de calificaciones por examen
        for (ex = 0; ex < numEx; ex++){
           prom = OABE.calculaPromCol(calif, ex, numAl);
           System.out.println("\nEl promedio del examen: " + (ex+1) + " es: " + prom);
        }

        // Imprime el alumno que obtuvo la calificación más alta por examen
        for (ex = 0; ex < numEx; ex++){
           res = OABE.buscaPosMayorColumna(calif, ex, numAl)+1;
           System.out.println("\nEl alumno que obtuvo la mayor calificación en el examen: " +
           (ex+1) + " es: " + res);
        }

        // Para cada alumno, imprime el examen en el que obtuvo la calificación más alta
        for (al = 0; al < numAl; al++){
           res = OABE.buscaPosMayorRenglón(calif, al, numEx)+1;
           System.out.println("\nEl alumno: " + (al+1) + " obtuvo la mayor calificación en el
           examen: " + res);
        }

        // Cuenta, por examen, cuántos alumnos sacaron 10
        for (ex = 0; ex < numEx; ex++){
           res = OABE.cuentaPorColumna(calif, ex, numAl, 10);
           if (res != 0)
              System.out.println("\n" + res + " alumnos sacaron 10 en el examen: " + (ex+1));
        }

        // Cuenta, por alumno, en cuántos exámenes sacó 6
        for (al = 0; al < numAl; al++){
           res = OABE.cuentaPorRenglón(calif, al, numEx, 6);
           if (res != 0)
              System.out.println("\nEl alumno: " + (al+1) + " sacó 6 en " + res + " exámenes");
        }
     }
  }
```

Muy importante

- La primera dimensión es para los renglones o filas.
- La segunda dimensión es para las columnas.
- Un arreglo bidimensional se puede recorrer por renglones o por columnas.
- El tamaño, en cuanto a renglones, se determina con .length.
- El tamaño, en cuanto a columnas, se determina con nombreArreglo[i].length.
- Los renglones y las columnas se enumeran desde 0.

5.7 LA CLASE ARREGLO BIDIMENSIONAL

Como en el caso de los arreglos unidimensionales, estos pueden representarse siguiendo el paradigma orientado a objetos. Con el objeto de ganar generalidad se usará el tipo T para declarar el arreglo. La figura 5.8 muestra el diagrama de clase UML correspondiente a esta clase. Como puede observarse, tiene como atributos la colección de elementos, el total de renglones ocupados, el total de columnas ocupadas y un máximo para renglones y otro para columnas. Estos dos últimos valores se requieren para que en el constructor por omisión pueda instanciarse el arreglo. Se incluyeron algunos métodos que le dan comportamiento a la clase: asignar un valor en una casilla, obtener el valor de una casilla, imprimir todo el arreglo, entre otros.

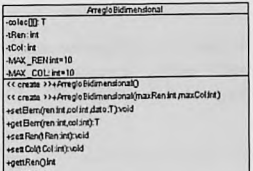


Figura 5.8 Ejemplo de arreglos paralelos

Algunos de los métodos de la clase *OABE* pueden incluirse debido a que no son exclusivos para el manejo de enteros. Por ejemplo el que cuenta cuántas veces un cierto dato está en un renglón o en una columna dada. En el programa 5.13 se presenta la clase ArregloBidimensional con algunos de estos métodos. Como puede observarse, se utiliza la excepción de Java *IndexOutOfBoundsException* para indicar un error ocasionado por un valor fuera del rango permitido para los índices del arreglo.

Progama 5.13 **ArregloBidimensional.java**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.13
 * Se define una clase para representar a los arreglos bidimensionales genéricos.
 * Se incluyen aquellas operaciones que puedan aplicarse a cualquier tipo de datos.
 */
public class ArregloBidimensional <T> {
   private T colec[ ][ ];
   private int tRen, tCol;
   private final int MAX_REN = 10;
   private final int MAX_COL = 10;

   // Constructor por omisión. Se usan las constantes para instanciar al arreglo.
   public ArregloBidimensional( ) {
      colec = (T[][]) new Object[MAX_REN][MAX_COL];
      tRen = 0;
      tCol = 0;
   }

   // Se recibe el máximo de renglones y de columnas que tendrá el arreglo.
   public ArregloBidimensional(int maxRen, int maxCol) {
      colec = (T[][]) new Object[maxRen][maxCol];
      tRen = 0;
      tCol = 0;
   }

   /* Valida que el índice dado como renglón sea mayor o igual que 0 y menor que el
    * total de renglones asignados al arreglo.
    */
   private boolean validaRen(int índice){
      return índice >= 0 && índice < tRen;
   }
```

```
    /* Valida que el índice dado como columna sea mayor o igual que 0 y menor que el
     * total de columnas asignadas al arreglo.
     */
    private boolean validaCol(int índice){
       return índice >= 0 && índice < tCol;
    }

   /* Se asigna el dato en la casilla indicada. Si los índices están fuera de rango
     * se lanza una excepción.
     */
    public void setElem(int ren, int col, T dato){
       if (validaRen(ren) && validaCol(col))
          colec[ren][col] = dato;
       else
          throw new IndexOutOfBoundsException( );
    }

    /* Regresa el elemento de la casilla indicada. Si los índices están fuera de rango
     * se lanza una excepción.
     */
    public T getElem(int ren, int col){
       if (validaRen(ren) && validaCol(col))
          return colec[ren][col];
       else
          throw new IndexOutOfBoundsException( );
    }

    /* Asigna el total de renglones. Si el valor dado está fuera de rango se lanza
     * una excepción.
     */
    public void settRen(int tRen) {
       if (tRen > 0 && tRen <= colec.length)
          this.tRen = tRen;
       else
          throw new IndexOutOfBoundsException( );
    }
```

```
/* Asigna el total de columnas. Si el valor dado está fuera de rango se lanza
 * una excepción.
 */
public void settCol(int tCol) {
   if (tCol > 0 && tCol <= colec[0].length)
      this.tCol = tCol;
   else
      throw new IndexOutOfBoundsException( );
}

// Regresa el total de renglones.
public int gettRen( ) {
   return tRen;
}

// Regresa el total de columnas.
public int gettCol( ) {
   return tCol;
}

// Genera una cadena con el contenido del arreglo, por renglón.
public String toString( ){
   int r, c;
   StringBuilder cad = new StringBuilder( );

   for (r = 0; r < tRen; r++){
      for (c = 0; c < tCol; c++)
         cad.append(colec[r][c] + "   ");
      cad.append("\n");
   }
   return cad.toString( );
}

/* Cuenta y regresa el total de veces que el dato se encuentra en el renglón
 * dado como parámetro (ren). Si ren está fuera de rango lanza una excepción.
 */
```

```
    public int cuentaPorRenglón(int ren, T dato){
      if (validaRen(ren)){
        int c, cont;

        cont = 0;
        for (c = 0; c < tCol; c++)
          if (colec[ren][c].equals(dato))
            cont = cont + 1;
        return cont;
      }
      else
        throw new IndexOutOfBoundsException( );
    }

    /* Cuenta y regresa el total de veces que el dato se encuentra en la columna
     * dada como parámetro (col). Si col está fuera de rango lanza una excepción.
     */
    public int cuentaPorColumna(int col, T dato){
      if (validaCol(col)){
        int r, cont;

        cont = 0;
        for (r = 0; r < tRen; r++)
          if (colec[r][col].equals(dato))
            cont = cont + 1;
        return cont;
      }
      else
        throw new IndexOutOfBoundsException( );
    }
  }
```

Existen métodos que, por su naturaleza, no pueden incluirse en la clase anterior, por ejemplo los que calculan la suma o el promedio de un renglón o columna. Por otra parte, en el caso de los métodos que buscan la posición del mayor (o menor) de un renglón o columna, sí pueden agregarse, pero exigen que la clase que se use para darle valor a T implemente la interface *Comparable* de Java. De esta manera se garantiza que la clase tenga un método compareTo(). A continuación se muestra el código con los cambios necesarios:

Para garantizar que el tipo T tenga un *compareTo()*, el encabezado de la clase debe cambiar a:

```
public class ArregloBidimensional <T extends Comparable<T>>
```

El método que busca y regresa la posición del dato más grande en un renglón dado como parámetro (ren), queda como se muestra más abajo. Observe que se usa el método *compareTo()* para comparar dos objetos y poder decidir cuál de los dos es el más grande.

```
public int buscaPosMayorRenglón(int ren){
   int c, col;

   col = 0;
   for (c = 1; c < tCol; c++)
      if (colec[ren][c].compareTo(colec[ren][col]) > 0)
         col = c;
   return col;
}
```

En caso de no hacerse de la manera que se mostró más arriba, al momento de usarse el método *compareTo()* se requiere convertir explícitamente el objeto que se compara a tipo *Comparable*. Sin embargo, si la clase usada no implementara la interface se generaría un error en tiempo de corrida. El código del método anterior, con el cambio mencionado es el siguiente:

```
public int buscaPosMayorRenglón(int ren){
   int c, col;

   col = 0;
   for (c = 1; c < tCol; c++)
      if (((Comparable)colec[ren][c]).compareTo(colec[ren][col]) > 0)
         col = c;
   return col;
}
```

En el programa 5.14 se presenta un ejemplo de aplicación de la clase *ArregloBidimensional*. En este caso, en una clase que representa una oficina de Recursos Humanos, el arreglo bidimensional almacena cadenas de caracteres. Analice el código.

Programa 5.14 **RecursosHumanos.java**


```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.14
 * Ejemplo del uso de un objeto tipo ArregloBidimensional (programa 5.13),
 * parametrizado con la clase String.
 * Se usa para almacenar los nombres de los empleados que se destacaron en cada uno
 * de los departamentos (columnas) de una tienda a lo largo de los últimos meses
 * (renglones).
 */
public class RecursosHumanos {
   private String nomGerente, tel;
   private ArregloBidimensional<String> empDestacados;
   private final int MESES = 12;
   private final int DEPTOS = 6;

   // Se construye un objeto tipo ArregloBidimensional.
   public RecursosHumanos( ) {
      empDestacados = new ArregloBidimensional(MESES, DEPTOS);
   }

   // Se construye un objeto tipo ArregloBidimensional y se asignan algunos atributos.
   public RecursosHumanos(String nomGerente, String tel) {
      this( );
      this.nomGerente = nomGerente;
      this.tel = tel;
   }

   /* Asigna el total de meses de los cuales se almacenarán datos (renglones).
    * Si el método settRen( ) lanza una excepción (no se pudo asignar), regresa false.
    */
   public boolean asignaMeses(int meses){
      boolean resp = true;
      try{
```

```
            empDestacados.settRen(meses);
        }
        catch (Exception e){
            resp = false;
        }
        return resp;
    }

    /* Asigna el total de departamentos (columnas). Si el método settCol( ) lanza una
     * excepción (no se pudo asignar), regresa false.
     */
    public boolean asignaDeptos(int deptos){
        boolean resp = true;
        try{
            empDestacados.settCol(deptos);
        }
        catch (Exception e){
            resp = false;
        }
        return resp;
    }

    /* Asigna el nombre del empleado que se destacó en un cierto departamento y en
     * un cierto mes. Si el método setElem( ) lanza una excepción, regresa false.
     */
    public boolean asignaEmpleado(int mes, int depto, String nombreEmp){
        boolean resp = true;
        try{
            empDestacados.setElem(mes, depto, nombreEmp);
        }
        catch(Exception e){
            resp = false;
        }
        return resp;
    }
```

```
    // Genera y regresa una cadena con los nombres de los empleados destacados.
    public String generaListaEmpleados(){
       return empDestacados.toString();
    }

    /* Dado el nombre de un empleado y un mes, regresa true si dicho empleado se
     * destacó en ese mes. Si el "mes" está fuera de rango, regresa false.
     */
    public boolean consultaEmpleadoyMes(int mes, String nombreEmp){
       boolean resp;
       try {
          resp = empDestacados.cuentaPorRenglón(mes, nombreEmp) > 0;
       }
       catch (Exception e){
          resp = false;
       }
       return resp;
    }

    /* Dado el número de un depto y el nombre de un empleado, regresa el total de
     * meses en los cuales dicho empleado se destacó. Si el "depto" está fuera de
     * rango, regresa -1.
     */
    public int consultaPorDepto(int depto, String nombreEmp){
       int cont;
       try {
          cont = empDestacados.cuentaPorColumna(depto, nombreEmp);
       }
       catch (Exception e){
          cont = -1;
       }
       return cont;
    }
}
```

Observe que en el programa anterior se manejaron las excepciones en aquellos métodos que utilizan métodos de la clase ArregloBidimensional que lanzan excepciones, por ejemplo *setElem()* en *asignaEmpleado()* y *cuentaPorColumna()* en *consultaPorDepto()*.

A continuación, en el programa 5.15 se muestra un método main muy simple en el cual se crea un objeto de tipo RecursosHumanos y, por medio de él, se genera información sobre los empleados que se han destacado.

Programa 5.15 | **EjemploArregloBidimensional.java**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.15
 * Uso de la clase RecursosHumanos, la cual tiene un atributo tipo ArregloBidimensional.
 */
public class EjemploArregloBidimensional {
   public static void main(String[] args) {

      // Se construye un objeto tipo RecursosHumanos.
      RecursosHumanos oficRH = new RecursosHumanos("Yolanda Pérez", "59762455");

      /* Se asigna el total de departamentos que maneja la tienda y el total de
       * meses de los cuales se tiene registro.
       */
      if (oficRH.asignaDeptos(3) && oficRH.asignaMeses(6)) {
          /* Se asignan los nombres de los empleados que se destacaron a lo largo de
           * los últimos 6 meses en cada uno de los departamentos.
           */
          oficRH.asignaEmpleado(0, 0, "Silvia Martínez");
          oficRH.asignaEmpleado(0, 1, "Matías Uriarte");
          oficRH.asignaEmpleado(0, 2, "Alicia Noble");

          oficRH.asignaEmpleado(1, 0, "Bruno Bertolucci");
          oficRH.asignaEmpleado(1, 1, "Pedro Ramírez");
          oficRH.asignaEmpleado(1, 2, "Luis Gallici");
```

```
        oficRH.asignaEmpleado(2, 0, "Silvia Martínez");
        oficRH.asignaEmpleado(2, 1, "María DallaCosta");
        oficRH.asignaEmpleado(2, 2, "Dolores Urreaga");

        oficRH.asignaEmpleado(3, 0, "Silvia Martínez");
        oficRH.asignaEmpleado(3, 1, "Luis Gallici");
        oficRH.asignaEmpleado(3, 2, "Martín Delgado");

        oficRH.asignaEmpleado(4, 0, "Sonia Vázquez");
        oficRH.asignaEmpleado(4, 1, "Inés Pérez");
        oficRH.asignaEmpleado(4, 2, "Martín Delgado");

        oficRH.asignaEmpleado(5, 0, "Irene Sánchez");
        oficRH.asignaEmpleado(5, 1, "Analía García");
        oficRH.asignaEmpleado(5, 2, "Juan Lamas");

        // Despliega el nombre de todos los empleados destacados, por mes.
        System.out.println("\n " + oficRH.generaListaEmpleados( ));

        // Si el empleado dado como parámetro se destacó en el mes dado, se informa.
        if (oficRH.consultaEmpleadoyMes(4, "Inés Pérez"))
            System.out.println("\nInés Pérez sí se destacó en mayo.");

        /* Obtiene e imprime el total de meses en los que se destacó la empleada
         * Silvia Martínez, del primer departamento.
         */
        int n;
        n = oficRH.consultaPorDepto(0, "Silvia Martínez");
        System.out.println("\nSilvia Martínez: " + n + " veces");
      }
      else
        System.out.println("\nDatos fuera de rango.");
    }
}
```

5.8 LAS CLASES ARRAYLIST Y VECTOR DE JAVA

En Java existen dos clases que representan arreglos: *ArrayList* y *Vector*. Ambas son genéricas, pueden guardar cualquier tipo de datos e incluso funcionar como variables polimórficas; y usan un arreglo interno para almacenar los datos. También tienen la particularidad de que si su capacidad inicial se satura, entonces crecen de tamaño. Es importante destacar que el arreglo interno no crece de tamaño, ya que, como se dijo, los arreglos son estructuras estáticas. Java crea un nuevo arreglo de mayor capacidad y copia los elementos en este nuevo arreglo cambiando posteriormente la referencia.

A pesar de las similitudes entre estas dos clases, existen algunas diferencias que justifican que se elija una clase sobre otra. La principal es que la clase *Vector* es sincronizada[1] y la clase *ArrayList* no lo es. En general, ante cierto problema, una vez entendido e identificados todos los requisitos que se deben cumplir, es conveniente elegir la estructura de datos que mejor se adecúe. Esto aplica a todas las estructuras de datos, las vistas y las que se verán a lo largo del libro.

5.8.1 Clase ArrayList [2]

Esta clase hereda e implementa una clase y algunas interfaces de Java, respectivamente. Se sugiere consultar la documentación del lenguaje si se requiere mayor información al respecto. Dos de los constructores usados para crear objetos de esta clase son:

```
public ArrayList( )
```

Construye un arreglo vacío, con una capacidad máxima de 10 elementos.

```
public ArrayList(int capInicial)
```

Construye un arreglo vacío, con una capacidad determinada por el valor capInicial dado como parámetro.

En la tabla 5.5 se presentan algunos[3] de los métodos de la clase *ArrayList*. En la primera columna se muestra la firma del método y en la segunda una explicación del mismo.

Tabla 5.5 Métodos de la clase ArrayList.

Método	Comentario
add(T dato):boolean	Agrega el dato al final del arreglo. Regresa true.
add(int índice, T dato): void	Agrega el dato en la posición indicada por el índice dado. Desplaza hacia la derecha el dato que está en esa posición y a todos los que estén a su derecha. Lanza la excepción IndexOutOfBoundsException si índice es < 0 o > size( ).

[1] El término sincronizada (Synchronized) hace referencia a la capacidad de una colección para que sus miembros puedan ser accesados desde diversas fuentes

[2] Para poder usarse se debe incluir: import java.util.ArrayList;

[3] Todos los constructores y métodos pueden consultarse en: http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

clear( ): void	Elimina todos los elementos del arreglo.
contains(Object o): boolean	Regresa true si el objeto o está en el arreglo.
ensureCapacity(int minCap): void	Aumenta la capacidad del arreglo para asegurar que, al menos, puede almacenar un total de elementos igual al valor dado como parámetro.
get(int índice): T	Regresa el elemento almacenado en el índice dado. Lanza la excepción IndexOutOfBoundsException si índice es < 0 o >= size( ).
indexOf(Object o): int	Regresa el índice de la primera ocurrencia del objeto dado como parámetro. Si no está en el arreglo, regresa -1.
lastIndexOf(Object o): int	Regresa el índice de la última ocurrencia del objeto dado como parámetro. Si no está en el arreglo, regresa -1.
isEmpty( ): boolean	Regresa true si el arreglo no tiene elementos; false en caso contrario.
remove(int índice): T	Elimina y regresa el elemento que ocupa la posición del índice dado. Lanza la excepción IndexOutOfBoundsException si índice es < 0 o >= size( ).
remove(Object o): boolean	Elimina el elemento dado como parámetro, si se encuentra en el arreglo. En este caso, regresa true.
set(int índice, T dato): T	Reemplaza el elemento actual del índice por el valor dado en dato. Regresa el valor que fue reemplazado. Lanza la excepción IndexOutOfBoundsException si índice es < 0 o >= size( ).
size( ): int	Regresa el número de elementos del arreglo.

A continuación se presenta la clase *Fábrica*, programa 5.16, en la cual uno de sus atributos es una lista de objetos tipo *Cliente* (programa 5.4). Para almacenar a los clientes se usó un objeto *ArrayList*. Observe que en todos los métodos, las operaciones sobre la lista de clientes quedaron a cargo de los respectivos métodos de la clase *ArrayList*.

Programa 5.16 **Fábrica.java**

```
package cap5;

import java.util.ArrayList;
import java.util.Iterator;

/**
```

```
 * @author Silvia Guardati
 * Programa 5.16
 * Ejemplo de aplicación de la clase ArrayList para almacenar una colección
 * de objetos tipo Cliente.
 */
public class Fábrica {
   private String nombre, domicilio;
   private ArrayList <Cliente> lisClientes;

   // Se construye un objeto tipo ArrayList.
   public Fábrica( ) {
      lisClientes = new ArrayList( );
   }

   // Se construye un objeto tipo ArrayList y se asignan valores a algunos atributos.
   public Fábrica(String nombre, String domicilio) {
      this( );
      this.nombre = nombre;
      this.domicilio = domicilio;
   }

   /* Regresa una cadena con todos los datos de la fábrica. Usa el método toString( )
    * de la clase ArrayList.
    */
   public String toString( ) {
      StringBuilder cad = new StringBuilder( );

      cad.append("\nNombre: " + nombre + "\nDomicilio: " + domicilio);
      cad.append("\nLista de clientes:" + lisClientes);
      return cad.toString( );
   }

   /* Agrega un nuevo cliente a la lista de clientes, asegurándose de que no se
    * repita. Regresa true si lo inserta y false en caso contrario. Usa los métodos
    * contains( ) y add( ) de la clase ArrayList.
    */
```

```
public boolean altaCliente(String nombre, String domic, double saldo){
   boolean resp = false;
   Cliente aInsertar = new Cliente(nombre, domic, saldo);

   if (!lisClientes.contains(aInsertar)){
      resp = true;
      lisClientes.add(aInsertar);
   }
   return resp;
}

/* Elimina un objeto tipo Cliente si está en la lista y regresa true.
 * En caso contrario, regresa false. Usa el método remove ( ) de la clase ArrayList.
 */
public boolean bajaCliente(String nombre){
   boolean resp;
   Cliente aQuitar = new Cliente(nombre);
   resp = lisClientes.remove(aQuitar);
   return resp;
}

/* Regresa una cadena con los datos de un cliente o un mensaje adecuado si
 * no está en la lista. La búsqueda se hace por nombre (revise el método equals( ) de la
 * clase Cliente). Usa los métodos indexOf( ) y get( ) de la clase ArrayList.
 */
public  String consultaCliente(String nombre){
   String res= "\n" + nombre + " no está registrado";
   int índice;
   Cliente aConsultar = new Cliente(nombre);

   índice = lisClientes.indexOf(aConsultar);
   if (índice >= 0)
      res = lisClientes.get(índice).toString( );
   return res;
}
```

```
    /* Regresa el número actual de clientes de la fábrica. Usa el método size( ) de la clase
     * ArrayList.
     */
    public int totalClientes( ){
       return lisClientes.size( );
    }

    /* Regresa la suma de los saldos de todos los clientes. Usa el método iterator( )
     * de la clase ArrayList, ya que todas las colecciones de datos de Java lo tienen.
     */
    public double calcTotalSaldo( ){
       Iterator <Cliente> it = lisClientes.iterator( );
       double totalSaldo = 0;

       while (it.hasNext( ))
          totalSaldo = totalSaldo + it.next( ).getSaldo( );
       return totalSaldo;
    }
}
```

En el programa 5.17 se construye un objeto tipo *Fábrica* y se muestra el uso de sus métodos.

Programa 5.17 **UsaArrayList**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.17
 * Ejemplo del uso de la clase Fábrica que tiene, entre sus atributos, un objeto
 * tipo ArrayList.
 */
public class UsaArrayList {
   public static void main(String[ ] args) {
```

```
boolean resp;
// Construye un objeto tipo Fábrica
Fábrica unaFábrica = new Fábrica("Muebles y Accesorios, S.A.", "Calle 13-235");

/* Agrega 3 clientes a la fábrica. Observe que se dan los datos
 * disponibles de los clientes, NO los objetos tipo Cliente.
 */
resp = unaFábrica.altaCliente("Juan Pérez", "Reforma 180", 5890.50);
if (!resp)
   System.out.println("\nJuan Pérez ya está registrado como cliente.");
resp = unaFábrica.altaCliente("Alicia Domínguez", "Alcorta 350", 12320);
if (!resp)
   System.out.println("\nAlicia Domínguez ya está registrado como cliente.");
resp = unaFábrica.altaCliente("José Castro", "España 987", 8970.25);
if (!resp)
   System.out.println("\nJosé Castro ya está registrado como cliente.");
// Dado que no se quieren clientes repetidos, deberá imprimir el mensaje.
resp = unaFábrica.altaCliente("Juan Pérez", "Reforma 180", 5890.50);
if (!resp)
   System.out.println("\nJuan Pérez ya está registrado como cliente.");

// Se despliega toda la información de la fábrica.
System.out.println("\nDatos de la fábrica: " + unaFábrica);

// Intenta eliminar un cliente que NO existe.
resp = unaFábrica.bajaCliente("Mario Saenz");
if (resp)
   System.out.println("\nEl cliente Mario Saenz fue dado de baja.");
else
   System.out.println("\nMario Saenz no está registrado como cliente.");

 // Intenta eliminar un cliente que existe.
resp = unaFábrica.bajaCliente("Juan Pérez");
if (resp)
   System.out.println("\nEl cliente Juan Pérez fue dado de baja.");
else
```

```
        System.out.println("\nJuan Pérez no está registrado como cliente.");
      // Se despliega toda la información de la fábrica.
      System.out.println("\nDatos de la fábrica: " + unaFábrica);

      // Se consultan los datos de clientes, dando los nombres de los mismos.
      System.out.println("\nConsulta " + unaFábrica.consultaCliente("Juan Pérez"));
      System.out.println("\nConsulta " + unaFábrica.consultaCliente("José Castro"));

      // Se despliega el total de clientes de la fábrica.
      System.out.println("\nTotal de clientes: " + unaFábrica.totalClientes( ));

      // Se despliega el acumulado de los saldos de todos los clientes.
      System.out.println("\nTotal saldos: $" + unaFábrica.calcTotalSaldo( ));
   }
}
```

5.8.2 Clase Vector[4]

Esta clase tiene tres atributos, el primero de ellos (T elementData[]) es para almacenar la colección de datos, el segundo (int elementCount) representa el número de elementos guardados y el tercero (int capacityIncrement) indica la cantidad de casillas que debe incrementarse cada vez que su tamaño se satura. Si este último atributo es 0, entonces la capacidad del arreglo se duplica cada vez que se requiere más espacio.

Considerando los atributos mencionados, para crear objetos tipo Vector se cuenta con los siguientes constructores:

```
public Vector( )
```

Construye un arreglo vacío, con un tamaño igual a 10 y un incremento de capacidad igual a 0.

```
public Vector(int capInicial)
```

Construye un arreglo vacío, con un tamaño determinado por el parámetro capInicial y un incremento de capacidad igual a 0.

```
public Vector(int capInicial, int incremCapac)
```

Construye un arreglo vacío, con un tamaño determinado por el parámetro capInicial y un incremento de capacidad dado por el parámetro *incremCapac*.

[4] Para poder usarse se debe incluir: import java.util.Vector;

En la tabla 5.6 se presentan algunos [5] de los métodos de la clase *Vector*. En la primera columna se muestra la firma del método y en la segunda una explicación del mismo.

Tabla 5.6 Métodos de la clase Vector

Método	Comentario
add(T dato):boolean	Agrega el dato al final del arreglo.
add(int índice, T dato): void	Agrega el dato en la posición indicada por índice. Desplaza hacia la derecha el dato que está en esa posición y a todos los que estén a su derecha. Lanza la excepción ArrayIndexOutOfBoundsException si índice es < 0 o > tamaño.
capacity( ): int	Regresa el tamaño del arreglo.
clear( ): void	Elimina todos los elementos del arreglo.
contains(Object o): boolean	Regresa true si el objeto o está en el arreglo.
elementAt(int índice): T	Regresa el componente que está en índice. Lanza la excepción ArrayIndexOutOfBoundsException si índice es < 0 o >= tamaño.
ensureCapacity(int minCap): void	Aumenta la capacidad del arreglo para asegurar que, al menos, puede almacenar un total de elementos igual al valor dado como parámetro.
firstElement( ): T	Regresa el elemento que está en el índice 0 del arreglo. Lanza la excepción NoSuchElementException si el arreglo está vacío.
lastElement( ): T	Regresa el elemento que está en el índice tamaño - 1 del arreglo. Lanza la excepción NoSuchElementException si el arreglo está vacío.
get(int índice): T	Regresa el elemento que está en la posición del índice. Lanza la excepción ArrayIndexOutOfBoundsException si índice es < 0 o >= tamaño.
indexOf(Object o): int	Regresa el índice de la primera ocurrencia del objeto dado como parámetro. Si no está en el arreglo, regresa -1.
isEmpty( ): boolean	Regresa true si el arreglo no tiene elementos.
size( ): int	Regresa el número de elementos almacenados en el arreglo.
toString( ): String	Regresa una cadena formada por la representación en cadena de cada uno de los elementos del arreglo.
remove(int índice): T	Elimina el dato almacenado en la posición indicada por índice. Lanza la excepción ArrayIndexOutOfBoundsException si índice es < 0 o >= tamaño.

[5] Todos los constructores y métodos pueden consultarse en http://docs.oracle.com/javase/6/docs/api/java/util/Vector.html

remove(Object o): boolean	Elimina la primera ocurrencia del objeto en el arreglo. Regresa true si se pudo eliminar.
setElementAt(T dato, int índice): void	Asigna el dato en la posición indicada por índice. Lanza la excepción ArrayIndexOutOfBoundsException si índice es < 0 o >= tamaño.
set(int índice, T dato): T	Reemplaza el elemento actual del índice por el valor dado en dato. Regresa el valor que fue reemplazado. Lanza la excepción ArrayIndexOutOfBoundsException si índice es < 0 o >= tamaño.

En el programa 5.18 se presenta la clase *Estudiante* que tiene un atributo de tipo *Vector*, en este caso de datos tipo *Double*. Observe cómo se emplean los métodos de la tabla 5.6 para darle funcionalidad a la clase *Estudiante*.

Programa 5.18 **Estudiante.java**

```
package cap5;
import java.util.Vector;

/**
 * @author Silvia Guardati
 * Programa 5.18
 * En la clase Estudiante se usa un objeto de la clase Vector para almacenar las
 * calificaciones. En los métodos se aprovechan las funcionalidades ofrecidas por
 * esta clase.
 */
public class Estudiante{
   private String nombre, carrera, telef;
   private int clave;
   private Vector<Double> calif;
   private final int MAX_CAL = 50;

  /* Se construye el objeto tipo Vector, indicando una capacidad inicial igual a
   * MAX_CAL.
   */
   public Estudiante( ){
      calif = new Vector(MAX_CAL);
   }
```

```
// Se construye el objeto tipo Vector y se asignan valores a los demás atributos.
public Estudiante(int clave, String nombre, String carrera, String telef) {
   this( );
   this.clave = clave;
   this.nombre = nombre;
   this.carrera = carrera;
   this.telef = telef;
}

// Se construye un objeto sólo con la clave. Auxiliar para búsquedas.
public Estudiante(int cla){
   clave = cla;
}

//Actualiza la carrera del alumno.
public  void  setCarrera(String  carrera){
   this.carrera = carrera;
}

// Regresa la carrera del alumno.
public String getCarrera( ) {
   return carrera;
}

// Regresa el nombre del alumno.
public String getNombre( ) {
   return nombre;
}

// Regresa el teléfono del alumno.
public String getTelef( ) {
   return telef;
}

/* Agrega una calificación al arreglo de calificaciones del alumno.
 * Usa el método add( ) de la clase Vector.
```

```
 */
 public void altaCalif(double cal){
    calif.add(cal);
 }

 /* Regresa una cadena con los datos del alumno. Usa el método toString( ) de
  * la clase Vector.
  */
 public String toString( ){
    StringBuilder cad = new StringBuilder( );

    cad.append("\n\nDatos del estudiante \nNombre: " + nombre);
    cad.append("\nCarrera: " + carrera + "\nCalificaciones: " + calif);
    return cad.toString( );
 }

 /* Calcula y regresa el promedio del alumno. Usa los métodos isEmpty( ), size( )
  * y get( ) de la clase Vector.
  */
 public  double  calculaPromedio( ){
    double  promedio;
    int i, totalCal;

    promedio = 0;
    if (!calif.isEmpty( )){
       totalCal = calif.size( );
       for (i = 0; i < totalCal; i++)
          promedio = promedio + calif.get(i);
       promedio = promedio / totalCal;
    }
    return promedio;
 }

 /* Se sobrescribe el método equals( ) de tal manera que pueda ser usado en clases
  * genéricas para comparar dos objetos tipo Estudiante.
  */
 public boolean equals(Object otro){
```

```
        boolean resp = false;

        if (otro != null && otro instanceof Estudiante)
            resp = clave == ((Estudiante)otro).clave;
        return resp;
    }
}
```

En el siguiente programa se presenta la clase *Salón* que también utiliza un objeto tipo *Vector*, pero ahora para almacenar objetos tipo *Estudiante*.

Programa 5.19 | **Salón.java**

```
package cap5;

import java.util.Iterator;
import java.util.Vector;

/**
 * @author Silvia Guardati
 * Programa 5.19
 * Ejemplo de uso de la clase Vector. En esta clase se usa un objeto tipo Vector
 * para almacenar una colección de objetos tipo Estudiante (programa 5.18).
 * Analice el uso del objeto tipo Vector y de sus métodos.
 */
public class Salón {
    private String nomProf, nomMateria;
    private int numSalón;
    private Vector<Estudiante> alumnos;
    private final int MAX_ALUM = 30;

    /* Se construye un objeto de tipo Vector indicando una capacidad inicial por medio
     * de MAX_ALUM y una capacidad de incremento igual a 5.
     */
    public Salón( ){
        alumnos = new Vector(MAX_ALUM, 5);
    }
```

```
public Salón(String nomProf, String nomMateria, int numSalón) {
   this( );
   this.nomProf = nomProf;
   this.nomMateria = nomMateria;
   this.numSalón = numSalón;
}

/* Agrega un estudiante al salón, siempre que no se repita. Usa los métodos
 * contains( ) y add( ) de la clase Vector.
 */
public boolean altaEstudiante(int clave, String nom, String car, String tel){
   Estudiante aInsertar = new Estudiante(clave, nom, car, tel);
   boolean resp;

   resp = !alumnos.contains(aInsertar);
   if (resp)    // Si el alumno no está registrado en el salón.
      resp = alumnos.add(aInsertar);
   return resp;
}

/* Agrega una nueva calificación al alumno. Recibe como parámetros la clave del
 * alumno y la nueva calificación. Si el alumno está en el salón, se agrega la
 * calificación -en el objeto tipo Vector usado para tal fin- y regresa true.
 * Usa los métodos indexOf( ) y get( ) de la clase Vector.
 */
public boolean altaCalifAlum(int clave, double cal){
   Estudiante unEstudiante = new Estudiante(clave);
   int índice;
   boolean resp = false;

   índice = alumnos.indexOf(unEstudiante);
   if (índice >= 0){
      resp = true;
      alumnos.get(índice).altaCalif(cal);
   }
   return resp;
```

```
}

/* Elimina un estudiante del salón. Regresa true si puede eliminarlo. Usa el
 * método remove( ) de la clase Vector.
 */
public boolean bajaEstudiante(int clave){
   Estudiante aQuitar = new Estudiante(clave);

   return alumnos.remove(aQuitar);
}

/* Obtiene y regresa el promedio de un alumno. Si el alumno no está en el
 * salón regresa -1. Usa los métodos indexOf( ) y get( ) de la clase Vector.
 */
public double calculaPromedio(int clave){
   Estudiante unEst = new Estudiante(clave);
   int índice;
   double promedio = -1; // Si no se encuentra al alumno, regresara -1.

   índice = alumnos.indexOf(unEst);
   if (índice >= 0)
      promedio = alumnos.get(índice).calculaPromedio( );
   return promedio;
}

/* Regresa el teléfono del alumno cuya clave se recibe. Si no se encuentra
 * regresa un mensaje. Usa los métodos indexOf( ) y get( ) de la clase Vector.
 */
public String consTeléfono(int clave){
   Estudiante unEst = new Estudiante(clave);
   String tel;
   int índice;

   índice = alumnos.indexOf(unEst);
   if (índice >= 0)
      tel = alumnos.get(índice).getTelef( );
```

```
        else
            tel = "-Ese estudiante no está en este salón-\n";
        return tel;
    }

    /* Genera una cadena con todos los datos del salón. Usa los métodos isEmpty( )
     * y toString( ) de la clase Vector.
     */
    public String toString( ){
        StringBuilder cad = new StringBuilder("\nDATOS DEL SALÓN\n");

        cad.append("Profesor: " + nomProf);
        cad.append("\nMateria: " + nomMateria);
        cad.append("\nNúmero salón: " + numSalón);
        if (alumnos.isEmpty( ))
            cad.append("\nNo hay estudiantes asignados a este salón.");
        else
            cad.append("\nLista de estudiantes: " + alumnos);
        return cad.toString( );
    }

    /* Obtiene y regresa el promedio de calificaciones de todos los estudiantes.
     * Usa los métodos isEmpty( ), size( ) e iterator( )de la clase Vector, este último
     * porque todas las colecciones de datos de Java lo tienen.
     */
    public double calcPromedioSalón( ){
        Iterator <Estudiante> it = alumnos.iterator( );
        double promedio = 0;
        if (!alumnos.isEmpty( )){
            while (it.hasNext( ))
                promedio = promedio + it.next( ).calculaPromedio( );
            promedio = promedio / alumnos.size( );
        }
        return promedio;
    }
}
```

Por último, en el programa 5.20 se presenta un ejemplo simple de aplicación de las clases *Estudiante* y *Salón* con sus correspondientes atributos tipo *Vector*.

Programa 5.20 **UsaVector**

```
package cap5;

/**
 * @author Silvia Guardati
 * Programa 5.20
 * Ejemplo del uso de objetos tipo Vector. Se prueba a través de un objeto tipo
 * Salón, programa 5.19, el cual tiene un atributo que es tipo Vector.
 */
public class UsaVector {
   public static void main(String[] args) {
      boolean resp;
      Salón unSalón = new Salón("Bruno Bertoluzzi", "Programación II", 103);

      // Imprime el estado inicial del salón.
      System.out.println(unSalón);

      /* Se dan de alta algunos estudiantes en el salón. Las 3 primeras son exitosas.
       * La última no.
       */
      resp = unSalón.altaEstudiante(4130, "Luis Urquiza", "Ing. en Computación", "56784083");
      if (resp)
         System.out.println("\nAlta exitosa: estudiante registrado en el salón");
      else
         System.out.println("\nAlumno repetido. No se registró.");
      resp = unSalón.altaEstudiante(5004, "Juan García", "Lic. en Derecho", "84755891");
      if (resp)
         System.out.println("\nAlta exitosa: estudiante registrado en el salón");
      else
         System.out.println("\nAlumno repetido. No se registró.");
      resp = unSalón.altaEstudiante(3954, "Carla Duarte", "Ing. Industrial", "39561823");
      if (resp)
         System.out.println("\nAlta exitosa: estudiante registrado en el salón");
      else
```

```
        System.out.println("\nAlumno repetido. No se registró.");
        resp = unSalón.altaEstudiante(4130, "Luis Urquiza", "Ingeniería en Computación",
        "56784083");
    if (resp)
        System.out.println("\nAlta exitosa: estudiante registrado en el salón");
    else
        System.out.println("\nAlumno repetido. No se registró.");

    // Se dan de alta algunas calificaciones para los estudiantes registrados en el salón
    resp = unSalón.altaCalifAlum(4130, 8.5);
    if (!resp)
        System.out.println("\nAlumno no registrado en el salón, no se pudo agregar calificación.");
    resp = unSalón.altaCalifAlum(4130, 9.3);
    if (!resp)
        System.out.println("\nAlumno no registrado en el salón, no se pudo agregar calificación.");
    resp = unSalón.altaCalifAlum(3954, 7.8);
    if (!resp)
        System.out.println("\nAlumno no registrado en el salón, no se pudo agregar calificación.");
    resp = unSalón.altaCalifAlum(3958, 10.0);
    if (!resp)  // Debe imprimir el mensaje.
        System.out.println("\nAlumno no registrado en el salón, no se pudo agregar calificación.");

    /* Imprime el salón luego de las altas de algunos estudiantes y de algunas
     * calificaciones.
     */
    System.out.println(unSalón);

    // Obtiene e imprime el promedio de un estudiante.
    System.out.println("\nPromedio de Luis Urquiza: " + unSalón.calculaPromedio(4130));

    // Obtiene e imprime el teléfono de un estudiante.
    System.out.println("\nTeléfono de Carla Duarte: " + unSalón.consTeléfono(3954));

    // Obtiene e imprime el promedio de los estudiantes del salón.
    System.out.printf("\nPromedio del salón: %6.2f ", unSalón.calcPromedioSalón( ) ) ;
```

```
        // Bajas de estudiantes. La primera con éxito, la segunda no.
        resp = unSalón.bajaEstudiante(4130);
        if (resp)
          System.out.println("\nBaja del estudiante exitosa.");
        else
          System.out.println("\nNo se pudo dar de baja: estudiante no registrado");
        resp = unSalón.bajaEstudiante(4786);
        if (resp)
          System.out.println("\nBaja del estudiante exitosa.");
        else
          System.out.println("\nNo se pudo dar de baja: estudiante no registrado");

        // Imprime el salón luego de las bajas.
        System.out.println(unSalón);
    }
}
```

5.9 RESUMEN

En este capítulo se presentaron los arreglos y se analizó su implementación por medio de la programación orientada a objetos. Asimismo, se estudiaron las principales operaciones que pueden realizarse sobre arreglos tanto lineales como bidimensionales.

Considerando el paradigma de programación seguido en este libro, se explicaron conceptos como iteradores y arreglos polimórficos, los cuales permiten generar soluciones algorítmicas más flexibles y generales.

Todos los temas de este capítulo se complementaron con ejemplos para favorecer la comprensión de los mismos.

5.10 EJERCICIOS

5.1 Escriba el método *equals()* en la clase AGO que reciba un arreglo genérico ordenado. Regrese *true* si los arreglos son iguales y *false* en caso contrario. Dos arreglos son iguales si tienen la misma cantidad de elementos y si estos son del mismo tipo, con la misma información, en contenido y orden.

5.2 Escriba el método buscaBinaria( ) en la clase OAG (programa 4.3, del capítulo anterior). Debe recibir como parámetro un arreglo genérico, el total de elementos y un dato a buscar. El resultado es un entero positivo que indica la posición en que el dato se encontró, o un entero negativo que indica la posición en la que debería estar.

5.3 Escriba el método ordenaSelecDirecta( ) en la clase OAG (programa 4.3, del capítulo anterior). Debe recibir como parámetro un arreglo genérico y el total de elementos. El método ordena el arreglo recibido aplicando el método de ordenación por selección directa.

5.4 En la clase OABE agregue un método que genere y regrese como resultado una cadena con el contenido de la diagonal principal. El método recibe un arreglo bidimensional y su tamaño, y regresa una cadena.

5.5 En la clase OABE agregue un método que genere y regrese como resultado una cadena con el contenido de la diagonal secundaria. El método recibe un arreglo bidimensional y su tamaño, y regresa una cadena.

5.6 En la clase OABE agregue un método que sume dos arreglos bidimensionales (matrices) de enteros. El método recibe las matrices a sumar y sus tamaños, y regresa como resultado la nueva matriz.

5.7 En la clase OABE agregue un método que multiplique dos arreglos bidimensionales (matrices) de enteros. El método recibe las matrices a multiplicar y sus tamaños, y regresa como resultado la nueva matriz.

5.8 En la clase OABE agregue un método que genere la transpuesta de una matriz dada. El método recibe una matriz y regresa como resultado la transpuesta de dicha matriz.

5.9 En la clase OABE agregue un método que invierta el orden de los renglones de una matriz dada. Es decir, el renglón 0 se intercambia con el renglón m-1, el 1 con el m-2, etc. El método no genera resultado, sino que la matriz es la que se queda con los cambios.

5.10 Defina la clase *CuadradoMágico* que tenga como uno de sus atributos un arreglo bidimensional de n x n, siendo n un número entero, positivo e impar (n es el número de renglones y de columnas). En la clase se debe escribir un método que "forme un cuadrado mágico" y lo almacene en el arreglo antes mencionado. Un cuadrado mágico de tamaño n se caracteriza por contener los números del 1 al n^2, y porque la suma de cada uno de los renglones, columnas y diagonales son iguales entre sí, todos iguales a: n (n^2 + 1)/2. Para formar el cuadrado se deben aplicar las siguientes reglas:

- **a.** El 1 se coloca en el primer renglón, columna central.
- **b.** El siguiente número se coloca en el renglón anterior, columna siguiente.
- **c.** El renglón anterior al primero es el último.
- **d.** La columna siguiente de la última es la primera.
- **e.** Si el número es un sucesor de un múltiplo de n se coloca en el renglón siguiente, misma columna.

Observe el siguiente ejemplo:

n = 5

Total de números a asignar = 1, 2, ..., 25

Suma renglón/columna/diagonal = 5 (25 + 1) / 2 = 5 (26)/2 = 130 / 2 = 65

	0	1	2	3	4
0	**17**	**24**	**1**	**8**	**15**
1	**23**	**5**	**7**	**14**	**16**
2	**4**	**6**	**13**	**20**	**22**
3	**10**	**12**	**19**	**21**	**3**
4	**11**	**18**	**25**	**2**	**9**

5.11 Defina una clase que tenga como atributo un arreglo de números con punto decimal para almacenar el total de lluvias registradas a lo largo de los 12 meses del último año, en cada uno de los estados del país. En la clase debe incluir métodos para obtener la siguiente funcionalidad:

a. Dado un mes, indicar cuál fue el estado en el que más llovió.

b. Dado un estado, indicar el mes en el que más llovió.

c. Indicar si hay un estado en el cual no haya llovido en todo el año.

d. Indicar si hay un estado en el cual se hayan registrado las máximas precipitaciones, en todos los meses del año.

e. Promedio anual de lluvias por estado.

f. Mes (si hubiera) en el cual, en todos los estados, llovió más de una cierta cantidad (esta cantidad se recibe como parámetro).

5.12 En una matriz de enteros se ha almacenado la cantidad (en grados centígrados) de calor registrado en cierta región. Es decir, considerando que los renglones y las columnas se corresponden a las coordenadas X y Y de un punto, la casilla en la posición X,Y representa la cantidad de calor en dicho punto. Se debe analizar esa información para calificar los puntos según la temperatura. Para ello defina una clase que tenga como atributo un arreglo bidimensional de enteros, el máximo de renglones, el máximo de columnas y el total de renglones y columnas. Además, en la clase debe escribir un método que, a partir del arreglo bidimensional, genere un arreglo unidimensional de PuntoConColor (programa 2.29 del capítulo 2), de acuerdo con lo que se describe a continuación:

- Si temperatura ≤ 5 °C → color verde
- Si temperatura > 5 °C y ≤ 10 °C → color amarillo
- Si temperatura > 10 °C → color rojo

Por lo tanto, cada punto "coloreado" guardará las coordenadas del punto junto con el color que indica qué nivel de temperatura se registró en ese punto. En la clase puede incluir otros atributos y métodos si lo necesita. Pruebe su solución.

5.13 Tomando como base la clase *ArregloBidimensional* (programa 5.13), escriba una clase *ABD* (*Arreglo BidimensionaldeDoble*) para representar un arreglo bidimensional de números con punto decimal. Decida qué atributos (tipo y cantidad) debe incluir. Además, defina todos aquellos métodos que crea conveniente para dotar a la clase de máxima funcionalidad. Se sugiere incluir métodos que obtengan y regresen la suma de un renglón/columna, que encuentren la posición del mayor/menor por renglón y por columna, etcétera.

5.14 Retome la clase del problema 8, ¿puede reescribirla usando la clase *ABD*, del problema 23? ¿Por qué sí/no? ¿Si su respuesta es sí, qué ventajas/desventajas encontró?

5.15 Defina la clase *Tienda* en la cual uno de sus atributos es una colección de productos. Puede incluir otros, si así lo requiere. Defina la clase *Producto* de tal manera que se ajuste a lo planteado más abajo. Para almacenar los productos use la clase *ArrayList*. La clase *Tienda* debe tener la siguiente funcionalidad:

a. Agregar nuevos productos. Los productos no pueden repetirse.
b. Eliminar un producto existente.
c. Consultar la existencia de un producto, dada la clave del mismo.
d. Generar una lista con todos los productos que vende la tienda.
e. Generar una lista con los nombres y los precios de todos los productos que cuestan más de $50.
f. Generar una lista de productos con todos aquellos productos de los cuales se tengan menos de 5 unidades. Cuando se agrega un producto a la nueva lista se debe quitar de la lista original.

¿Pudo usar los métodos de la clase *ArrayList*? ¿La solución resultó más simple que si hubiera usado la clase *AGD*? ¿Resultó eficiente para la solución del inciso a)? ¿Por qué sí/no? ¿Podría almacenar los productos de manera ordenada, por clave o nombre? ¿Por qué sí/no?

5.16 Retome el problema anterior y realice los cambios necesarios para usar un objeto de la clase *Vector* para almacenar los productos. ¿Cambia la lógica de la solución? Conteste las mismas preguntas que se formularon al final del ejercicio mencionado.

5.17 Escriba una clase *IteradorArregloBidimensional* que represente el concepto de un iterador que permita acceder a los elementos de un arreglo bidimensional sin tener que conocer su estructura.

5.18 Modifique la clase *ArregloBidimensional.java* (programa 5.13) incluyendo el método *iterator()* de la interface Iterable. El método debe generar un objeto de tipo *Iterator* usando la clase definida en el ejercicio anterior. Escriba un método main para probar su solución.

ESTRUCTURAS ENLAZADAS

6

Contenido

Competencias

- Explicar el concepto de estructuras enlazadas.
- Distinguir entre estructuras de datos estáticas y dinámicas.
- Presentar el diseño y la implementación de estructuras enlazadas usando el paradigma de programación orientada a objetos.
- Aplicar las estructuras enlazadas en el diseño de soluciones algorítmicas.

6.1 INTRODUCCIÓN

En este capítulo se presenta un nuevo tipo de estructura de datos: las estructuras enlazadas, las cuales se caracterizan por su dinamismo en cuanto al manejo de la memoria. En el capítulo anterior, al estudiar los arreglos se mencionó que son estáticos porque no pueden cambiar de tamaño durante la ejecución del programa, lo que en ciertos problemas puede ser una limitante importante. En el caso de las estructuras enlazadas, el dinamismo es su principal ventaja; pueden ocupar espacio y liberarlo a medida que se agreguen o eliminen datos, respectivamente.

Un **nodo** almacena un dato (tan simple o tan complejo como se desee) y la dirección o referencia a otro nodo. A partir de este concepto se define una estructura enlazada como una colección de elementos llamados *nodos*. Por medio de las referencias que guardan los nodos, estos se van enlazando y forman un todo, que es propiamente la estructura o colección de datos.

Una estructura enlazada, además de **dinámica**, es **lineal** y **homogénea**. *Lineal* porque cada nodo tiene un único predecesor y un único sucesor, con excepción del primero y del último, que no tienen predecesor y sucesor, respectivamente. Es *homogénea* porque todos los elementos son del mismo tipo. Por último, es importante destacar que los nodos sólo se pueden accesar por medio de la referencia almacenada por su predecesor; es decir, estas estructuras no permiten el acceso directo, sino que debe hacerse elemento por elemento, comenzando con el primero. Por lo anterior, es de fundamental importancia conocer (y no perder) la referencia del primer nodo.

Muy importante

- ☑ Colección compuesta por nodos.
- ☑ Colección dinámica de datos: se agregan o quitan elementos a medida que se necesitan. El máximo de elementos queda determinado por la aplicación o por las capacidades de manejo de memoria de la computadora.
- ☑ Colección homogénea de datos: todos los elementos son del mismo tipo.
- ☑ Colección lineal: cada elemento tiene un único predecesor y un único sucesor, con excepción del primero y del último, respectivamente.
- ☑ No permiten el acceso directo a cada uno de sus nodos.
- ☑ Es importante conocer SIEMPRE la dirección del primer nodo, ya que es lo que permitirá el acceso a todos los demás.

6.2 COMPONENTES DE UNA ESTRUCTURA ENLAZADA

Una estructura enlazada está compuesta por nodos. A la vez, cada nodo está compuesto por dos partes: una, para almacenar el dato –razón de ser de la estructura de datos–; otra, para almacenar la dirección, referencia o enlace a otro nodo. Gráficamente, un nodo puede verse como muestra la figura 6.1.

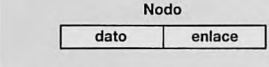


Figura 6.1 Representación gráfica de un nodo

Si la estructura está formada por varios nodos, gráficamente puede representarse como aparece en la figura 6.2.

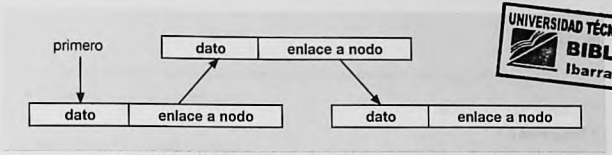


Figura 6.2 Representación gráfica de una estructura enlazada

Como se observa en la figura 6.2, la única variable que se requiere conocer para tener acceso a toda la estructura es aquella que guarda la dirección del primer nodo. A partir de ella se llega al primer nodo y éste nos lleva al segundo, el que a su vez nos lleva al tercero, y así hasta el final. ¿Cómo saber que se llegó al último nodo? En general, se usa algún valor –compatible con las referencias– que indique que no hay ningún enlace guardado y, por tanto, no hay un siguiente nodo. En el caso particular de Java, se usa el valor **null**, ya que es compatible con cualquier tipo de objetos.

El diagrama de clase UML, que se presenta en la figura 6.3, representa la clase *Nodo*, la cual tiene dos atributos. Uno de ellos, de tipo *T*, se utiliza para almacenar al dato, y el otro, tipo *Nodo*, se emplea para guardar la referencia o dirección de otro nodo. Cuando esto suceda, se tendrá una colección de nodos enlazados, es decir, una estructura enlazada. Las operaciones sobre los nodos son muy básicas: recuperar el dato o el enlace, asignar un nuevo valor al dato o al enlace e imprimir el dato.

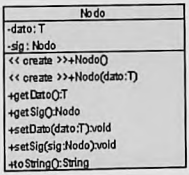


Figura 6.3 Diagrama de la clase *Nodo*

El programa 6.1 corresponde a la clase *Nodo*. Para asegurar la generalidad se utiliza el tipo *T* para definir el atributo *dato*. De esta forma, el nodo podrá emplearse en la construcción de estructuras enlazadas capaces de almacenar información de diversos tipos.

Programa 6.1 **Nodo.java**

```
package cap6;

/**
 * @author Silvia Guardati
 * Programa 6.1
 * Definición de la clase Nodo, la cual tiene dos atributos. Uno de ellos para
 * almacenar un dato y el otro para almacenar la dirección de un objeto tipo Nodo.
 */
public class Nodo <T> {
   private T dato;
   private Nodo<T> sig;

   // Al construir un objeto tipo Nodo, éste no referencia a ningún otro nodo.
   public Nodo( ) {
      sig = null;
   }

   // Se construye un objeto, asignándole valor al dato.
   public Nodo(T dato) {
      this.dato = dato;
      sig = null;
   }

   // Regresa el dato almacenado.
   public T getDato( ) {
      return dato;
   }

   // Regresa la dirección del nodo a quien referencia.
   public Nodo<T> getSig( ) {
      return sig;
   }

   // Asigna un nuevo valor al dato.
   public void setDato(T dato) {
```

```
        this.dato = dato;
    }

    /* Asigna la dirección de un nodo, de tal manera que ahora el nodo que invoca
     * al método referencia a otro nodo.
     */
    public void setSig(Nodo<T> sig) {
        this.sig = sig;
    }

    // Regresa en forma de cadena el dato almacenado.
    public String toString() {
        return "Dato: " + dato;
    }
}
```

Para construir un nuevo objeto tipo *Nodo*, haciendo uso de los constructores incluidos, se empleará:

```
Nodo apNodo = new Nodo();
                O
Nodo apNodo = new Nodo(dato);
```

6.3 OPERACIONES EN ESTRUCTURAS ENLAZADAS

Como ya se mencionó, estas estructuras son muy flexibles en cuanto a las operaciones que pueden realizarse en ellas. Es posible agregar o quitar elementos sin necesidad de recorrimientos, ya que los nodos se agregan o se quitan actualizando solamente los enlaces entre ellos.

A continuación se analizan las principales operaciones. Otras, por ser más simples, se presentan directamente en el programa 6.3, correspondiente a la clase *EstructuraEnlazada* (*EE*).

6.3.1 Inserción

En una estructura enlazada se puede agregar nuevos datos en cualquier parte de la misma, afectando solamente las referencias o direcciones en los nodos involucrados. Por tanto, se podrá agregar nuevos nodos al inicio, al final o en una posición intermedia. Puede haber otras variantes como, por ejemplo, agregar nodos

antes o después de cierto dato dado como referencia. En esta sección se analiza la inserción al inicio y al final de la estructura, y se presentan los principales pasos del algoritmo en pseudocódigo. En el programa 6.3 se incluyen los métodos correspondientes a estas operaciones.

Cualquiera que sea el orden en el cual se vaya a insertar el nuevo nodo, es importante considerar si la estructura está vacía: no tiene nodos y la variable que apunta al primer nodo almacena el valor **null**. Si así fuera, se aplica la secuencia de pasos que permite construir un nuevo nodo y redefinir el *primero* con la dirección de dicho nodo. Los pasos para llevar a cabo esta operación son:

1. Construir un nuevo nodo (figura 6.4 - a).
2. Asignar el nuevo nodo a *primero* (figura 6.4 - b).

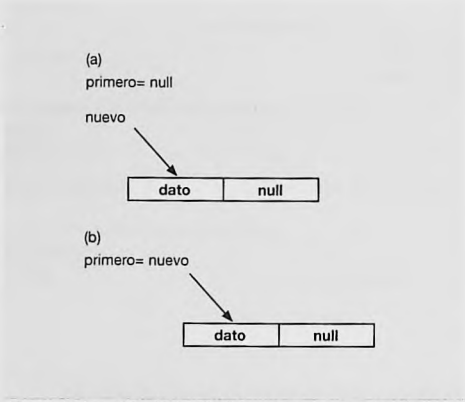


Figura 6.4 Inserción al inicio/final de una estructura enlazada vacía

Como resultado de los pasos señalados, se tendrá una estructura formada por un único nodo, que es el insertado en esta operación.

Si la estructura tuviera uno o más nodos, el algoritmo cambia, ya que se debe tomar en cuenta la posición en la cual se deberá agregar el nuevo nodo. Si fuera al inicio de la estructura, el nuevo nodo se debe ligar con el primer nodo de ésta y, posteriormente, redefinir el primero. Los pasos para llevar a cabo esta operación son:

1. Construir un nuevo nodo (figura 6.5 - a).
2. Asignar primero al atributo *sig* del nuevo nodo (línea punteada, figura 6.5 - a).
3. Asignar el nuevo nodo a *primero* (figura 6.5 - b).

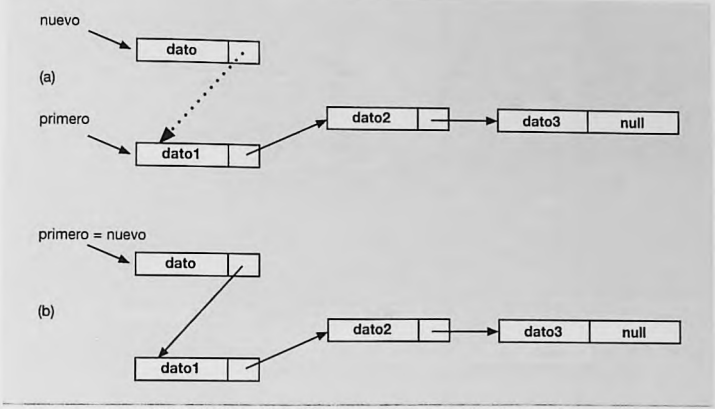


Figura 6.5 Inserción al inicio de una estructura enlazada no vacía

Cuando el nuevo nodo se debe insertar al final de la estructura, se debe llegar al último nodo de la misma, de tal manera que se pueda ligar a éste con el nuevo nodo. Recuerde que el apuntador al primer nodo no debe perderse; por tanto, es necesario usar un apuntador auxiliar que pueda ir recorriendo los nodos hasta llegar al último. Los pasos para llevar a cabo esta operación son:

1. Asignar un apuntador *auxiliar* al primer nodo de la estructura (figura 6.6 - a).
2. Recorrer el apuntador *auxiliar* hasta llegar al último nodo (figura 6.6 - b).
3. Construir un nuevo nodo (figura 6.6 - c).
4. Asignar el nuevo nodo al atributo *sig* del *auxiliar* (figura 6.6 - c).

Muy importante

- ☑ Se construye un nuevo nodo con el dato que se quiere agregar.
- ☑ Si la estructura está vacía, el nuevo nodo es el *primero*.
- ☑ Si no está vacía y se quiere agregar al inicio de la estructura, se liga el nuevo con el *primero* y se redefine *primero* con el valor del nuevo.
- ☑ Si no está vacía y se quiere agregar al final de la estructura, se debe usar un apuntador auxiliar para llegar al último nodo y luego ligar a éste con el nuevo.
- ☑ No perder el valor de *primero*.

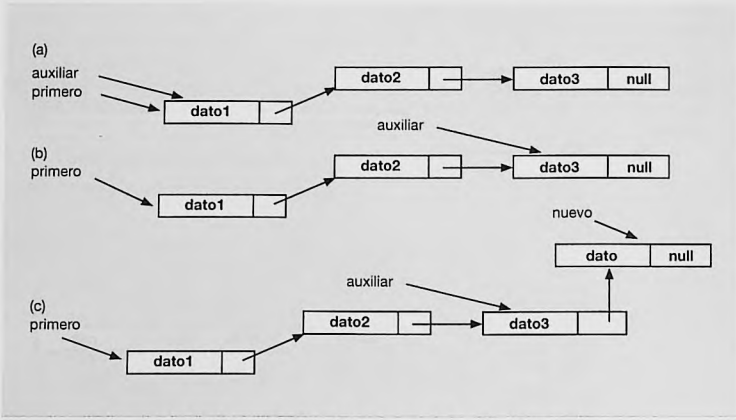


Figura 6.6 Inserción al final de una estructura enlazada no vacía

6.3.2 Eliminación

Dada una estructura enlazada, se puede eliminar un nodo de acuerdo a su posición, o bien, de acuerdo con su contenido. Ejemplo del primer caso es la eliminación del primero o del último nodo de la estructura; mientras que en el otro caso sería eliminar el nodo que almacena un cierto dato, teniendo en cuenta que el nodo podría estar en cualquier posición de la estructura. En ambos casos, la estructura no debe estar vacía para poder eliminar un nodo. A continuación se presentan gráficamente y en pseudocódigo estas operaciones, suponiendo que la estructura está formada por, al menos, un nodo.

Para eliminar el primer nodo se debe colocar un apuntador auxiliar sobre dicho nodo, de tal manera que luego se pueda redefinir el valor de *primero* con la dirección del nodo que le sigue, la cual se encuentra almacenada en su atributo *sig*. Por último, se rompe la relación entre el dato eliminado y la estructura enlazada. Los pasos a seguir son:

1. auxiliar = primero (figura 6.7 - a)
2. primero = primero.getSig() (figura 6.7 - b)
3. auxiliar.setSig(null) (figura 6.7 - c)

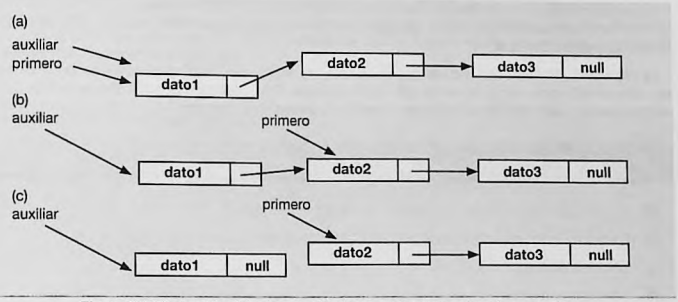


Figura 6.7 Eliminación del primer nodo de una estructura enlazada no vacía

Observe que la secuencia de pasos propuesta es válida para estructuras que tengan un solo nodo. En este caso, en el paso 2 se le asigna el valor **null** a *primero*, indicando así que la estructura está vacía.

Si se desea quitar el último nodo, será necesario llegar hasta el nodo anterior a éste y asignarle a su atributo *sig* el valor de **null**. En consecuencia, se romperá la relación entre la estructura y el nodo eliminado; además, el valor de **null** indicará que ahora ése es el último nodo de la estructura. Los pasos a seguir son:

1. Asignar un apuntador *auxiliar* al primer nodo de la estructura (figura 6.8 - a).
2. Recorrer *auxiliar* hasta que guarde la dirección del penúltimo nodo (figura 6.8 - b).
3. Asignar al atributo *sig* del *auxiliar* el valor **null** (figura 6.8 - c).

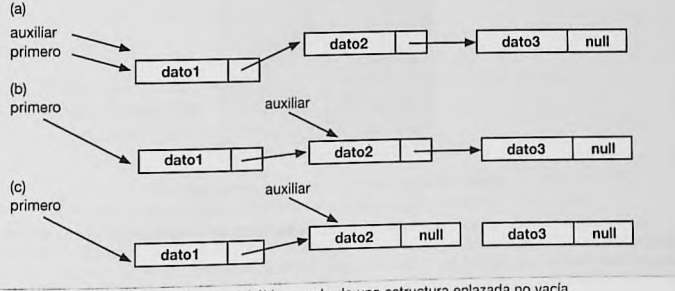


Figura 6.8 Eliminación del último nodo de una estructura enlazada no vacía

La secuencia de pasos planteada no es válida si la estructura estuviera formada por un único nodo. Por tanto, para tener una solución general es necesario verificar explícitamente si éste es el caso; si es así, actualizar el valor de *primero* con **null**.

Es importante destacar que una vez que se quita la referencia a un nodo, el espacio de memoria ocupado por éste será liberado por el recolector de basura de Java. Esto refuerza la característica señalada de estas estructuras en cuanto a la flexibilidad para el manejo dinámico de la memoria.

Muy importante

- Si la estructura tiene un solo nodo y se elimina, se debe asignar *null* a *primero*.
- Si el nodo a eliminar es el primero, se debe redefinir el valor de *primero* con la dirección del siguiente nodo.
- Si el nodo es el *último* se debe considerar si, además, es el único.
- Los espacios de memoria ocupados por los nodos eliminados serán liberados por el recolector de basura de Java.

6.4 IMPLEMENTACIÓN DE UNA ESTRUCTURA ENLAZADA EN JAVA

Para implementar una estructura enlazada en Java, aplicando el paradigma orientado a objetos, es necesario definir primero la clase *Nodo*, porque, como ya se dijo, la estructura es una colección de nodos. Es decir, en términos del desarrollo de *software* orientado a objetos, se dice que la clase *EstructuraEnlazada* es una composición de la clase *Nodo*. En la figura 6.9 se presenta el diagrama UML de la clase *EstructuraEnlazada* en la cual hay un único atributo, de tipo Nodo, usado para guardar la dirección del primer nodo de la estructura.

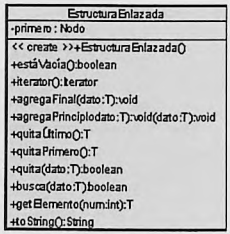


Figura 6.9 Diagrama de la clase *EstructuraEnlazada* (*EE*)

Además de la clase *Nodo*, programa 6.9, es conveniente definir otra clase que complemente a la clase *EstructuraEnlazada*. Como se vio en el capítulo 5, es posible crear un iterador externamente desde una clase usuaria para recorrer una estructura de datos sin conocer la organización interna de la misma. Para ello, la estructura debe tener un iterador diseñado especialmente, de acuerdo con sus propias características. Para diferenciarlo del iterador creado para arreglos (IteradorArreglo, programa 5.8) se lo llamará *IteradorEE*.

Es importante recordar que un iterador, que implementa la interface *Iterador* de Java, debe contar con los métodos *hasNext(), next()* y *remove()*, siendo este último opcional; por tanto, estos métodos deberán implementarse para funcionar sobre una estructura enlazada de nodos. A continuación se presenta el programa correspondiente a la clase *IteradorEE.java*.

Programa 6.2 | **IteradorEE.java**

```
package cap6;

import java.util.*;

/**
 * @author Silvia Guardati
 * Programa 6.2
 * Se define un iterador para estructuras enlazadas (EE), por lo tanto se implementan
 * los métodos hasNext( ) y next( ) de acuerdo a esta estructura de datos.
 */

public class IteradorEE<T> implements Iterator <T> {
   private Nodo <T> actual;

   /* Al construirse un objeto tipo IteradorEE se posiciona sobre un nodo, mismo
    * que será el primero de la estructura enlazada.
    */
   public IteradorEE(Nodo <T> actual) {
      this.actual = actual;
   }

   // Regresa true si existe un elemento.
   public boolean hasNext( ) {
      return actual != null;
   }

   /* Regresa el objeto apuntado por el iterador y se desplaza una posición.
    * En caso de no existir dicho elemento, se lanza una excepción.
    */
   public T next( ) {
```

```
        if (!hasNext())
            throw new NoSuchElementException();
        else {
            T resul = actual.getDato();
            actual = actual.getSig();
            return resul;
        }
    }

    /* Elimina el último elemento devuelto por el iterador.
     * Es una operación opcional que no se implementa.
     */
    public void remove() {
        throw new UnsupportedOperationException("No está implementada.");
    }
}
```

En el programa 6.3 se presenta el código de la clase *EstructuraEnlazada*, a la cual, por razones de comodidad, se la llamó *EE*. Para relacionar la clase *EE* con la clase *IteradorEE* se estableció que la misma implemente la interface *Iterable*. De esta forma contrae la obligación de tener un método *iterador()*, el cual construye y regresa como resultado un objeto de tipo *IteradorEE*. Usa la clase ExcepciónColecciónVacía.jav (programa 6.4).

Programa 6.3 **EE.java (EstructuraEnlazada)**

```
package cap6;

import java.util.Iterator;

/**
 * @author Silvia Guardati
 * Programa 6.3
 * Se define la clase EE (Estructura Enlazada). Tiene un único atributo, que es la
 * dirección o referencia al primer nodo.
 */
public class EE <T > implements Iterable<T> {
```

```
private Nodo <T> primero;

// Se construye un objeto tipo EE, indicando que la estructura está vacía inicialmente.
public EE ( ) {
   primero = null;
}

// Regresa true si la estructura está vacía (no tiene ningún nodo).
public boolean estáVacía( ) {
   return primero == null;
}

// Regresa un iterador sobre la estructura enlazada, partiendo del primer nodo.
public Iterator<T> iterator( ) {
   return new IteradorEE<T> (primero);
}

// Agrega un dato al final de la estructura enlazada.
public void agregaFinal(T dato) {
   Nodo <T> nuevo = new Nodo <T> (dato);
   if (estáVacía( ))
      primero = nuevo;
   else{
      Nodo <T> auxiliar = primero;
      while (auxiliar.getSig( ) != null)
         auxiliar = auxiliar.getSig( );
      auxiliar.setSig(nuevo);
   }
}

//Agrega un elemento al principio de la estructura enlazada.
public void agregaPrincipio(T dato) {
   Nodo <T> nuevo = new Nodo <T> (dato);
   nuevo.setSig(primero);
   primero = nuevo;
}
```

```
/* Elimina el primer nodo de la estructura, regresando el dato que almacena.
 * Si la estructura está vacía lanza una excepción.
 */
public T quitaPrimero( ) {
  if (estáVacía( ))
    throw new ExcepciónColecciónVacía("No hay elementos");
  else {
    T eliminado;
    Nodo <T> auxiliar = primero;
    eliminado = primero.getDato( );
    primero = primero.getSig( );
    auxiliar.setSig(null);
    return eliminado;
  }
}

/* Si el dato está en la estructura de datos, se elimina el nodo que almacena
 * el dato y regresa true. En caso contrario, regresa false.
 * Si la estructura está vacía lanza una excepción.
 */
public boolean quita(T dato){
  boolean resp = false;

  if (estáVacía( ))
    throw new ExcepciónColecciónVacía("No hay elementos");
  else
    if (dato.equals(primero.getDato( )))
      quitaPrimero( );
    else {
      Nodo<T> actual = primero;
      Nodo<T> anterior = primero;
      while (actual != null && !actual.getDato( ).equals(dato)){
        anterior = actual;
        actual = actual.getSig( );
      }
      if (actual != null){ // El dato está en la estructura enlazada.
        resp = true;
```

```
            anterior.setSig(actual.getSig());
            actual.setSig(null);
          }
        }
      return resp;
    }

    /* Elimina el último nodo de la estructura y regresa el dato que almacena.
     * Si la estructura está vacía lanza una excepción.
     */
    public T quitaUltimo(){
      if (estáVacía())
        throw new ExcepciónColecciónVacía("No hay elementos");
      else{
        Nodo <T> actual = primero;
        Nodo <T> anterior = primero;
        while (actual.getSig() != null){
          anterior = actual;
          actual = actual.getSig();
        }
        T dato = actual.getDato();
        if (primero == actual) // Hay un solo nodo
          primero = null; // La estructura queda vacía
        else {
          anterior.setSig(null);
          actual = null;
        }
        return dato;
      }
    }

    /* Busca el dato en la estructura enlazada. Regresa true en caso de éxito y
     * false si no lo encuentra.
     */
    public boolean busca(T dato){
      boolean resp = false;
      Nodo<T> auxiliar = primero;
```

```
        while (auxiliar != null && !auxiliar.getDato().equals(dato))
            auxiliar = auxiliar.getSig();
        if (auxiliar != null)
            resp = true;
        return resp;
    }

    /* Regresa el dato del nodo que ocupa la posición num dentro de la estructura
     * enlazada. Si la estuctura está vacía o tiene un número de nodos menor a num
     * entonces regresa null.
     */
    public T getElemento(int num){
        Nodo <T> auxiliar = primero;
        int cont = 1;
        T elemento;

        while (auxiliar != null && cont < num){
            auxiliar = auxiliar.getSig();
            cont++;
        }
        if (cont < num || auxiliar == null)
            elemento = null;
        else
            elemento = auxiliar.getDato();
        return elemento;
    }

    // Regresa en forma de cadena la información almacenada en los nodos.
    public String toString(){
        Iterator <T> it = iterator();
        StringBuilder cad = new StringBuilder();

        while (it.hasNext())
            cad.append(it.next() + " ");
        return cad.toString();
    }
}
```

Se define una clase auxiliar para representar una excepción en caso de que la estructura esté vacía y se intente quitar un elemento. La excepción se declara como una subclase de la clase *RuntimeException* de Java. El código correspondiente se presenta en el programa 6.4. Como se puede observar, el código es muy simple, sólo consta de dos constructores.

Programa 6.4 **ExcepciónColecciónVacía.java**

```
package cap6;

/**
 * @author Silvia Guardati
 * Programa 6.4
 * Se define una clase propia para el manejo de excepción en los casos en que la
 * colección o estructura de datos esté vacía.
 */
public class ExcepciónColecciónVacía extends RuntimeException{

   public ExcepciónColecciónVacía( ){
      super("Colección vacía");
   }

   public ExcepciónColecciónVacía(String mensaje){
      super(mensaje);
   }
}
```

6.5 APLICACIONES DE ESTRUCTURAS ENLAZADAS

La estrucutura de datos que se presenta en este capítulo se utilizará en capítulos subsecuentes como base para implementar estructuras de datos abstractas. Es decir, estructuras de datos que definen conceptualmente la manera de almacenar y recuperar datos, pero que no proveen una herramienta para llevarlo a cabo y, por tanto, al momento de implementarlas se debe utilizar arreglos o estructuras enlazadas.

En el programa 6.5 se presenta una aplicación de la clase *EE* para almacenar varios objetos de tipo *Persona* (programa 2.14, del paquete cap2). El programa es una aplicación muy simple que permite observar la construcción de un objeto tipo *EE* y el uso de sus métodos para agregar, buscar y eliminar objetos dentro de la estructura enlazada.

Programa 6.5 **UsaEstructuraEnlazada.java**

```
package cap6;

import cap2.Persona;
/**
 * @author Silvia Guardati
 * Programa 6.5
 * Ejemplo de aplicación de la clase EE (EstructuraEnlazada) para almacenar varios
 * objetos tipo Persona (progama 2.14, paquete cap2). Se hace uso de los métodos de la
 * clase EE para procesar los datos guardados.
 */
public class UsaEstructuraEnlazada {

   public static void main(String[ ] args) {
      // Se construye un objeto tipo EE, parametrizando con la clase Persona.
      EE<Persona> invitados = new EE( );

      // Se construyen 4 objetos tipo Persona.
      Persona p1 = new Persona("Jazmín García", "11/12/2001", "España 123", 'f');
      Persona p2 = new Persona("Mabel Osorio", "18/06/2000", "Junín 425", 'f');
      Persona p3 = new Persona("Matías Ulloa", "23/05/2002", "Santa Fe 890", 'm');
      Persona p4 = new Persona("Fernando Farrel", "02/04/2001", "Juárez 367", 'm');

      // Se agregan los objetos al final de la estructura enlazada.
      invitados.agregaFinal(p1);
      invitados.agregaFinal(p2);
      invitados.agregaFinal(p3);
      invitados.agregaFinal(p4);

      // Se imprime el contenido de la estructura enlazada.
      System.out.println("\nDatos de los invitados\n" + invitados);

      /* Se busca una persona en la estructura enlazada, indicando si está o no
       * dentro de los invitados.
       */
      if (invitados.busca(p3))
```

```
            System.out.println("\n" + p3.getNombre( ) + " sí está invitado.");
        else
            System.out.println("\n" + p3.getNombre( ) + " no está invitado.");

        // Se elimina la primera persona almacenada en la estructura enlazada.
        System.out.println("\nSe quita al primer invitado: " + invitados.quitaPrimero( ));

        // Se imprime el contenido de la estructura enlazada, luego de la eliminación.
        System.out.println("\nDatos de los invitados\n" + invitados);
    }
}
```

6.6 RESUMEN

En este capítulo se explicaron las estructuras de datos dinámicas llamadas *estructuras enlazadas*. Considerando su dinamismo, constituyen una alternativa muy útil para aquellas aplicaciones que requieren un manejo eficiente del espacio de memoria.

También se presentó su diseño e implementación usando la programación orientada a objetos y su aplicación en la solución de problemas. Sobre este último punto se volverá extensamente en los siguientes capítulos.

6.7 EJERCICIOS

6.1 Retome la clase del programa 6.3. Agregue un método entero que cuente y regrese como resultado el total de nodos que tiene la estructura.

6.2 Retome la clase del programa 6.3. Agregue un método entero que cuente y regrese como resultado el total de ocurrencias almacenadas de un cierto dato dado como parámetro.

6.3 Retome la clase del programa 6.3. Agregue los métodos que se describen a continuación. Ambos métodos reciben dos datos tipo T: uno de ellos es la referencia (*ref*) y otro el dato a insertar (*nuevo*). Los métodos son booleanos: regresan *true* si se lleva a cabo la inserción en la estructura enlazada o *false* en caso contrario. Considere todos los casos que puedan presentarse. Pruebe su solución.

a. Método para insertar un dato (*nuevo*) antes de otro dato dado como referencia (*ref*).

b. Método para insertar un dato (*nuevo*) después de otro dato dado como referencia (*ref*).

PILAS Y COLAS

7

Contenido

Competencias

- Explicar el concepto de pila como estructura de datos abstracta.
- Analizar diferentes alternativas de implementación de las pilas.
- Presentar las operaciones válidas en una pila.
- Explicar el concepto de cola como estructura de datos abstracta.
- Analizar diferentes alternativas de implementación de las colas.
- Presentar las operaciones válidas en una cola.

7.1 INTRODUCCIÓN

Este capítulo está dedicado al estudio de dos estructuras de datos abstractas (ADT por sus siglas del inglés, Abstract Data Type) que se conocen con el nombre de *pila* y *cola* respectivamente. Se dice que una estructura de datos es abstracta porque se describen sus características y su comportamiento, pero no su implementación. Por lo tanto, no hay una única manera de implementarlas, como sí sucede, por ejemplo, con los arreglos, sino que se debe usar alguna otra estructura para implementarla, siempre garantizando que se comporte según lo especificado.

7.2 PILA

Una pila es una estructura de datos lineal en la cual las operaciones sólo pueden hacerse por uno de los extremos, lo cual determina que el último elemento insertado sea el primero que puede quitarse. De ahí que a estas estructuras también se las conozca con el nombre de LIFO (Last-In, First-Out). El extremo recibe generalmente el nombre de *tope* y el mismo se actualiza luego de una inserción o una eliminación.

En la vida real encontramos numerosos ejemplos donde se usa este concepto. Por ejemplo, piense en una pila de platos (figura 7.1); si se necesita uno seguramente se tomará el que está encima de todos; a su vez, si se desocupa uno, el mismo deberá colocarse sobre todos los demás. En el área de computación hay varias aplicaciones importantes de las pilas, las cuales se tratarán un poco más adelante en este capítulo.

Figura 7.1 Ejemplo de pila

Independientemente del tipo de dato que almacene, la pila mostrará el mismo comportamiento. Observe la figura 7.2 (a) en la cual se insertaron los elementos 1, 2 y 3 en ese orden, por lo tanto el tope queda apuntando al 3. Si se hace una eliminación, el que saldrá será el 3, quedando como se muestra en (b). Por último, si se inserta el 4 la pila queda como se presenta en (c).

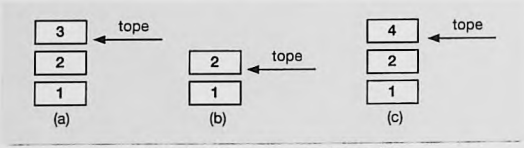


Figura 7.2 Comportamiento de una pila

7.2.1 Implementación de una pila

Antes de estudiar con mayor detalle las operaciones que pueden hacerse en una pila, es necesario decidir cómo implementarlas, ya que de esto dependerá en gran medida la manera de llevar a cabo cada una de las operaciones en esta estructura de datos.

La figura 7.3 presenta el diagrama UML de la interface *PilaADT* y de las clases *PilaA* y *PilaE*. Con la interface se define el comportamiento esperado de una pila por medio de las firmas de los métodos: *pop()*, *push()*, *peek()* y *isEmpty()*. El primero de ellos para quitar un dato de la pila, el segundo para agregar un dato a la pila, el tercero para consultar el dato que está en el tope, y el último para conocer si la pila está vacía. Las dos clases representan el concepto de una pila, la primera de ellas implementándola por medio de un arreglo y la segunda a través de una estructura enlazada.

Muy importante

- Estructura de datos abstracta.
- Estructura lineal.
- Un solo extremo se usa para hacer las operaciones de inserción y de eliminación.
- El extremo se llama *tope*.
- El último elemento almacenado es el primero en salir.
- Las pilas también se conocen como estructuras LIFO (Last-In, First-Out).

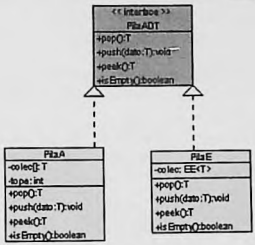


Figura 7.3 Diagrama UML para representar la interface *PilaADT* y las clases que la implementan

En la interface se establece el comportamiento esperado de una pila, sin importar la estructura de datos que se elija para su implementación. En el programa 7.1 se presenta esta interface. Observe que la interface está prevista para almacenar cualquier tipo de datos y que se incluyeron métodos para agregar y quitar un elemento, así como dos métodos auxiliares para consultar el elemento que está en el tope y si la pila está vacía, respectivamente.

Programa 7.1 | **PilaADT.java**

```
package cap7;

/**
 * @author Silvia Guardati
 * Programa 7.1
```

```
 * Se define el comportamiento esperado de una pila a través de una interface.
 */
    public interface PilaADT<T>{
        public T pop( ); // Debe quitar el elemento que está en el tope y regresarlo.
        public void push(T dato); // Agrega el dato en el tope de la pila.
        public T peek( ); // Regresa el elemento que está en el tope, sin quitarlo.
        public boolean isEmpty( ); // Regresa true si la pila no tiene elementos.
}
```

¿Cómo implementar una pila, considerando que es una estructura abstracta? Antes se mencionó que es una estructura de datos lineal, por lo tanto existen dos maneras para hacerlo: por medio de arreglos y por medio de estructuras enlazadas. La figura 7.3 presenta estas dos posibles implementaciones. Las ventajas y desventajas de cada una de las implementaciones estarán determinadas por la estructura base elegida. En el caso de los arreglos son muy fáciles de utilizar pero son estáticos, por lo tanto puede suceder que el espacio reservado inicialmente para almacenar se llene. Si se quiere compensar esta limitación construyendo un arreglo más grande y copiando todos los elementos del arreglo original a éste, se tendrá cierto desperdicio de memoria y mayor cantidad de procesamiento. Por su parte, las estructuras enlazadas pueden ser un poco más difíciles de entender y de manipular, pero al ser dinámicas son más eficientes en cuanto al manejo de memoria.

A continuación se muestra la implementación por medio de un arreglo. Observe que sólo se incluyeron los atributos y los constructores. Los métodos que implementan las operaciones antes mencionadas se agregarán luego de que se analicen en la siguiente sección. El programa 7.2 (que se presenta más adelante) corresponde a la clase *PilaA* completa, atributos y métodos.

```
public class PilaA<T> implements PilaADT<T>{
   private T colec[ ];
   private int tope;
   private final int MAX = 10;

   /* Se construye un arreglo de objetos y se lo convierte explícitamente a tipo T.
    * Inicialmente la pila está vacía, lo que se indica con tope igual a -1.
    */
   public PilaA( ) {
      colec = (T[ ]) (new Object[MAX]);
      tope = -1;
   }

   /* Se construye un arreglo de objetos y se lo convierte explícitamente a tipo T.
    * El espacio máximo reservado queda determinado por el parámetro max.
```

```
     * Inicialmente la pila está vacía, lo que se indica con tope igual a -1.
     */
    public PilaA(int max) {
        colec = (T[]) (new Object[max]);
        tope = -1;
    }
}
```

La clase tiene dos atributos, uno para almacenar la colección de datos, para lo cual usa un arreglo, y otro para guardar el valor del tope. Adicionalmente se define una constante como auxiliar en la construcción del arreglo en caso de que el usuario no dé un valor para el máximo número de casillas. En los constructores el tope se inicializa con el valor de -1 para indicar que la pila está vacía. Es importante tener en cuenta que el tope **apunta** al último elemento insertado; y no hay que confundir con el total que se maneja en los arreglos que almacena cuántos elementos fueron guardados en el arreglo (ocupando las posiciones de 0 a total -1).

A continuación se presenta la implementación de la pila por medio de una estructura enlazada. Observe que sólo se incluyeron los atributos y los constructores. Los métodos que implementan las operaciones antes mencionadas se agregarán luego de que se analicen en la siguiente sección. El programa 7.3 (que se presenta más adelante) corresponde a la clase *PilaE* completa, con atributos y métodos.

```
public class PilaE<T> implements PilaADT<T>{
  private EE <T>colec;

  // Se construye una estructura enlazada genérica.
  public PilaE( ) {
      colec = new EE( );
  }
}
```

Muy importante

- En los constructores – implementación con arreglo – el *tope* se inicializa con -1.
- En el constructor – implementación con estructura enlazada – se debe construir un objeto de tipo EE.
- El *tope* apunta al último elemento insertado en la pila.
- En una implementación por medio de arreglo se tendrán *tope* + 1 elementos.

La clase tiene un único atributo, que es la estructura enlazada en la cual se almacenará la colección de datos. En este caso no se define un atributo para el tope, ya que la misma estructura se encargará de manejar un *puntero* al último dato insertado, el cual será el atributo *primero* de la estructura enlazada. Si el lector no tiene presente la clase *EE* se sugiere revisar el material del capítulo 6 y el código del programa 6.3, EE.java, que puede encontrarse en el paquete cap6 del proyecto de NetBeans que complementa este libro.

7.2.2 Operaciones en una pila

Al inicio del capítulo se presentó el comportamiento esperado de las pilas, mencionando que las mismas deben poder agregar o quitar elementos por un único extremo. Además de estas operaciones, generalmente se definen algunas auxiliares. Una de ellas se utiliza para determinar si la pila está vacía y otra para consultar el elemento que está en el tope sin necesidad de quitarlo.

« Agregar un elemento a la pila

Este método, mejor conocido como *push*, agrega un elemento en el extremo de la pila, actualizando el valor del *tope*. Para implementar esta operación se debe considerar la estructura de datos sobre la que se trabajará. Por lo tanto, primero se muestra la versión para arreglos y, posteriormente, la correspondiente a estructuras enlazadas.

En el caso del arreglo, teniendo en cuenta su naturaleza estática, se debe verificar que la pila no esté llena. Si es así, se incrementa en 1 el *tope* y en esa nueva posición se asigna el dato. Recuerde que el *tope* debe apuntar al último elemento insertado. En el caso de que el arreglo esté lleno, se deberá construir un arreglo de mayor capacidad y copiar en éste los valores ya almacenados. Luego se sigue con los pasos ya mencionados; es decir, se incrementa en 1 el *tope* y en esa posición se asigna el nuevo valor. En la figura 7.4 se ejemplifica la operación *push* en una pila implementada por medio de un arreglo.

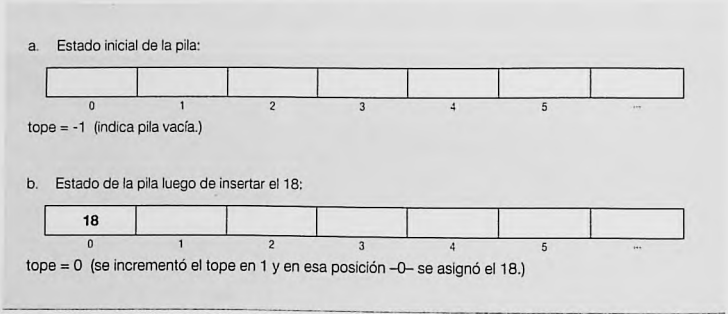


Figura 7.4 Operación *push( )* en una pila implementada con un arreglo

El algoritmo en pseudocódigo es el siguiente:

1. Si pila está llena, se construye un arreglo de mayor capacidad y se copian en él los elementos ya almacenados.
2. Se incrementa en 1 el tope.
3. Se asigna el nuevo valor en la posición señalada por tope.

A continuación se presenta el código en Java para implementar el método *push()* correspondiente a la clase *PilaA*.

```
/* Agrega el dato en el tope, redefiniendo el valor de éste. Si la pila está llena,
* se crea un arreglo de mayor capacidad y se copian los elementos de la pila a éste.
*/
public void push(T dato) {
   if (tope == colec.length - 1)
        aumentaCapacidad( );
   tope++;
   colec[tope] = dato;
}
```

Si la implementación está basada en una estructura enlazada, entonces se procederá a invocar el método *agregaPrincipio()* de la estructura, ya que el inicio será el único extremo por el cual se agregarán o se quitarán elementos. Recuerde que este método crea un nuevo nodo que almacena el dato dado y posteriormente liga este nodo con el *primero* y, por último, redefine el *primero* con el valor del nuevo nodo. Gráficamente se ve como muestra la figura 7.5.

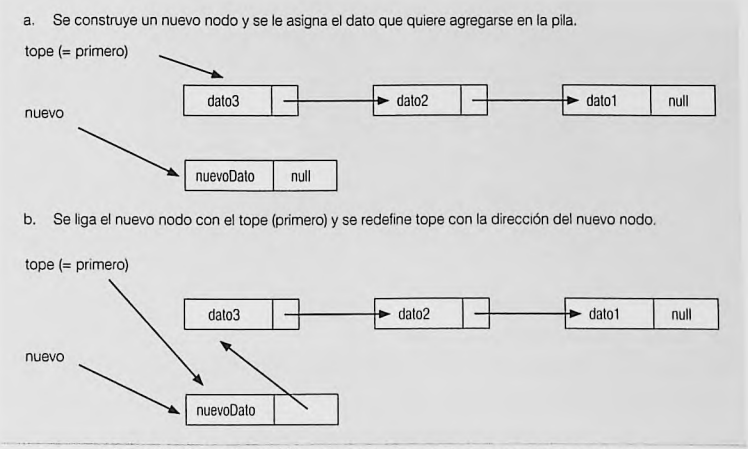


Figura 7.5 Operación *push()* en una pila implementada con una estructura enlazada

A continuación se presenta el código en Java correspondiente al método *push()* de la clase *PilaE*.

```
// Agrega el dato en el tope de la pila.
public void push(T dato) {
colec.agregaPrincipio(dato);
}
```

« Quitar un elemento de la pila

Este método, mejor conocido como *pop*, quita el elemento que está en el tope de la pila, actualizando el valor de éste. Si la pila tiene al menos un elemento, el método regresa el valor que se quitó; en caso contrario, lanza una excepción indicando que la colección está vacía. Para la excepción se reusa la clase *ExcepciónColecciónVacía* definida en el capítulo anterior (puede consultarse en el paquete cap6, programa 6.4).

En el caso de la clase implementada por medio del arreglo, si la pila está vacía se lanza una excepción terminando la operación. En caso contrario, se guarda en una variable auxiliar el contenido del tope y se actualiza el valor de éste, disminuyendo en 1, de tal manera que ahora apunte al siguiente elemento disponible de la pila. Observe la figura 7.6. En (a) se muestra el estado de la pila antes de ejecutar el *pop*, donde *tope* es igual a 3. Luego de eliminar el elemento del *tope* la pila queda como se muestra en (b). El valor de *tope* ahora es 2.

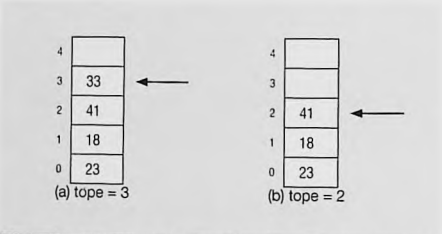


Figura 7.6 Operación *pop()* en una pila implementada con un arreglo

El algoritmo en pseudocódigo es el siguiente:

1. Si pila está vacía, se lanza una excepción.
2. Se almacena en una variable auxiliar el contenido de la casilla cuyo índice es *tope*.
3. Se redefine el valor de *tope* restándole 1.

El código en Java correspondiente al método pop de la clase *PilaA* queda como se muestra a continuación.

```
/* Elimina y regresa el elemento que está en el tope de la pila, redefiniendo el valor
* del tope. Si la pila está vacía lanza una excepción.
*/
  public T pop( ) {
    if (isEmpty( ))
      throw new ExcepciónColecciónVacía("Pila vacía");
    else{
      T dato = colec[tope];
      tope--;
      returndato;
    }
  }
```

Cuando se trata de una pila implementada por medio de una estructura enlazada, si la misma está vacía se lanza una excepción terminando la operación. En caso contrario, el método regresa el resultado del método *quitaPrimero()* de la clase *EE*. Recuerde que este último regresa el contenido del primer nodo y mueve el puntero hacia el siguiente nodo de la estructura. En la figura 7.7 se puede observar en (a) el estado original de la pila y en (b) el estado de la pila luego de eliminar el elemento que está en el tope. El tope apunta al siguiente elemento y se perdió la relación entre el dato3 (eliminado) y los demás datos.

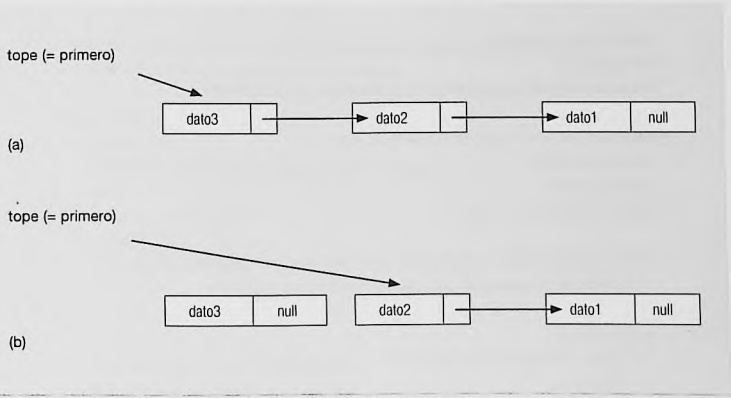


Figura 7.7 Operación *pop()* en una pila implementada con una estructura enlazada

El código en Java correspondiente al método *pop* de la clase *PilaE* queda como se muestra a continuación.

```
/* Elimina y regresa el elemento del tope de la pila.
 * Si la pila está vacía lanza una excepción.
*/
  public T pop( ) {
    if (colec.estáVacía( ))
       throw new ExcepciónColecciónVacía("Pila vacía");
    else
       return colec.quitaPrimero( );
  }
```

Una vez analizadas las operaciones de inserción y eliminación en una pila, se presentan las clases *PilaA* y *PilaE* con todos sus métodos en los programas 7.2 y 7.3, respectivamente.

Programa 7.2 | **PilaA.java**

```
package cap7;
import cap6.ExcepciónColecciónVacía;

/**
 * @author Silvia Guardati
 * Programa 7.2
 * Clase que implementa una pila genérica usando un arreglo genérico.
*/
public class PilaA<T> implements PilaADT<T>{
  private T colec[ ];
  private int tope;
  private final int MAX = 10;

  /* Se construye un arreglo de objetos y se lo convierte explícitamente a tipo T.
   * Inicialmente la pila está vacía, lo que se indica con tope igual a -1.
   */
  public PilaA( ) {
       colec = (T[ ]) (new Object[MAX]);
       tope = -1;
  }
```

```
/* Se construye un arreglo de objetos y se lo convierte explícitamente a tipo T.
 * El espacio máximo reservado queda determinado por el parámetro max.
 * Inicialmente la pila está vacía, lo que se indica con tope igual a -1.
 */
public PilaA(int max) {
    colec = (T[]) (new Object[max]);
    tope = -1;
}

/* Agrega el dato en el tope, redefiniendo el valor de éste. Si la pila está llena,
 * se construye un arreglo de mayor capacidad y se copian los elementos de la pila a éste.
 */
public void push(T dato) {
    if (tope == colec.length - 1)
        aumentaCapacidad();
    tope++;
    colec[tope] = dato;
}

/* Elimina y regresa el elemento que está en el tope de la pila, redefiniendo el valor
 * del tope. Si la pila está vacía lanza una excepción.
 */
public T pop( ) {
    if (isEmpty( ))
      throw new ExcepciónColecciónVacía("Pila vacía");
    else{
      T dato = colec[tope];
      tope--;
      return dato;
    }
}

/* Regresa el elemento que está en el tope.
 * Si la pila está vacía lanza una excepción.
 */
```

```
public T peek( ) {
    if (isEmpty( ))
        throw new ExcepciónColecciónVacía("Pila vacía");
    else
        return colec[tope];
}

// Regresa true si la pila está vacía.
public booleanisEmpty( ) {
    return tope == -1;
}

/* Método auxiliar que construye un arreglo de mayor tamaño y copia en él todos
 * los elementos de la pila, asignando al arreglo colec la referencia del nuevo arreglo.
 */
  private void aumentaCapacidad( ){
    T nuevo[ ] = (T[ ]) (new Object[colec.length * 2]);
    inti;

    for (i= 0; i<= tope; i++)
        nuevo[i] = colec[i];
    colec = nuevo;
  }
}
```

Programa 7.3 | **PilaE.java**

```
package cap7;
import cap6.EE;
import cap6.Excepción ColecciónJava;
/**
 * @author Silvia Guardati
* Programa 7.3
```

```
* Clase que implementa una pila genérica usando una estructura enlazada.
*/
public class PilaE<T> implements PilaADT<T>{
   private EE <T>colec;

   // Se construye una estructura enlazada genérica.
   public PilaEE( ) {
      colec = new EE( );
   }

// Agrega el dato en el tope de la pila.
public void push(T dato) {
      colec.agregaPrincipio(dato);
}

/* Elimina y regresa el elemento del tope de la pila.
 * Si la pila está vacía lanza una excepción.
 */
public T pop( ) {
      if (colec.estáVacía( ))
         throw new ExcepciónColecciónVacía("Pila vacía");
      else
         return colec.quitaPrimero( );
}

/* Regresa el dato almacenado en el tope de la pila (primer nodo de la EE).
 * Si la pila está vacía lanza una excepción.
 */
public T peek( ) {
      if (colec.estáVacía( ))
         throw new ExcepciónColecciónVacía("Pila vacía");
      else
         return colec.getElemento(1);
}

// Regresa true si la pila está vacía.
```

```
    public boolean isEmpty( ) {
        return colec.estáVacía( );
    }
}
```

Como se puede apreciar en el programa 7.3, la clase *PilaE* reusa varios métodos de la clase *EE*.

7.2.3 Aplicaciones de pilas: calculadora

El concepto de *pila* se utiliza mucho en computación. Una de las aplicaciones más interesantes y evidentes son en los métodos recursivos. Sobre este tema se volverá en el capítulo 8, por lo que aquí sólo se menciona brevemente.

En esta sección se presenta un problema de aplicación de pilas que consiste en simular una calculadora que:

- Evalúa si una expresión dada en notación infija está correctamente escrita. En particular, se evaluará si los paréntesis están bien balanceados (cada paréntesis izquierdo tiene su correspondiente derecho).
- Convierte la expresión dada en notación infija a su equivalente en notación postfija.
- Evalúa la expresión.

Por lo tanto, básicamente la calculadora contará con tres métodos que le darán la funcionalidad señalada, más algunos métodos auxiliares o complementarios. Es importante mencionar que la calculadora implementada es muy básica, tanto en las operaciones que maneja (suma, resta, multiplicación y división) como en las revisiones de sintaxis que hace antes de evaluar una expresión. No obstante esto, es una aplicación que permite mostrar claramente el uso de las pilas en la solución de problemas.

« Evaluación de los paréntesis

Para llevar a cabo esta operación se usará una pila como auxiliar. Se analizará la expresión (dada en forma de cadena) de izquierda a derecha identificando los paréntesis. A continuación se presenta el algoritmo en pseudocódigo:

```
Mientras no se haya evaluado toda la expresión hacer:
   Si el carácter analizado es un paréntesis izquierdo
   entonces se guarda en la pila.
   en caso contrario
      Si el carácter analizado es un paréntesis derecho
      entonces
         Si la pila no está vacía
         entonces se quita un elemento de la pila
         en caso contrario se interrumpe el ciclo
```

Fin del ciclo Mientras
Si se recorrió toda la expresión y la pila está vacía
entonces la expresión tiene los paréntesis bien balanceados
en caso contrario los paréntesis no están bien balanceados.

El código correspondiente a este algoritmo puede consultarse en el proyecto EstructurasDatosBásicas de NetBeans, en el paquete cap7, clase *Calculadora* (programa 7.4). Como puede observarse, en el método *revisa()* se utiliza una pila de tipo *Character* (clase de Java) para almacenar los paréntesis izquierdos temporalmente mientras se hace el análisis de la expresión. Esto se debe a que la pila es genérica y, por lo tanto, exige una clase para darle valor a *T*.

« Convierte la expresión infija a postfija

Por medio de esta operación se deberá obtener el equivalente de la expresión dada en su correspondiente en notación postfija. Recuerde que una expresión en notación postfija se caracteriza porque los operandos preceden al operador. Además, no se usan los paréntesis, sino que la prioridad de los operadores queda determinada por la ubicación de los mismos. Observe los siguientes ejemplos:

Expresión infija	Expresión postfija
a + b	a b +
a + b * c	a b c * +
(a + b) * c	a b + c *
a * b – c + d / e	a b * c – d e / +

El algoritmo que convierte la expresión infija a postfija es el siguiente:

Repetir con cada elemento de la expresión dada en infija:
  Si el elemento es un paréntesis izquierdo
  entonces se guarda en la pila.
  en caso contrario
    Si el elemento es un paréntesis derecho
    entonces
      Mientras el elemento que está en el tope de la pila sea ≠ paréntesis izquierdo
        Quitar un elemento de la pila
        Agregar el elemento quitado a la expresión postfija
      Fin del ciclo Mientras
        Quitar el paréntesis izquierdo (no se agrega a la expresión postfija)

```
en caso contrario
  Si es un operando
  entonces se agrega a la expresión postfija
  en caso contrario //Es un operador
    Mientras la pila no esté vacía y la prioridad del operador sea ≤ que la
                    prioridad del  operador que está en el tope de la pila hacer:
        Quitar un elemento de la pila
        Agregar el elemento quitado a la expresión postfija
    Fin del ciclo Mientras
    Agregar a la pila el operador
Fin del ciclo Repetir
Mientras la pila no esté vacía hacer:
      Quitar un elemento de la pila
      Agregar el elemento quitado a la expresión postfija
Fin del ciclo Mientras
```

Este algoritmo puede aplicarse una vez que se haya verificado que la expresión está correctamente escrita. En la solución que se presenta en este libro sólo se verifica que los paréntesis estén bien balanceados, sin embargo sería conveniente revisar que la expresión no tuviera otro tipo de errores como, por ejemplo, dos operadores juntos (a */ b) o que no empezara con un operador de multiplicación o división. Estas posibles mejoras quedan a cargo del lector. Asimismo, se puede generalizar el manejo de operadores de tal manera que se permitan otros, por ejemplo potencia, valor absoluto, etcétera.

El código correspondiente a este algoritmo puede consultarse en el proyecto EstructurasDatosBásicas de *NetBeans*, en el paquete cap7, clase *Calculadora* (programa 7.4). Como puede observarse, en el método *conviertePostfija()* se utiliza una pila de tipo *String* (clase de Java) para almacenar algunos elementos (paréntesis izquierdo, operandos y operadores) temporalmente mientras se realiza la conversión de la expresión. Para que este método pueda aplicarse es necesario que antes se obtengan los componentes de la expresión. En la solución que se presenta en el programa 7.4 se usa el método *obtieneTokens()*, el cual genera un arreglo de cadenas que almacena en cada casilla uno de los elementos de la expresión original.

« Evalúa la expresión

Una vez convertida la expresión se debe evaluar para dar el resultado correspondiente. Para ello se utiliza el siguiente algoritmo:

```
Repetir con cada elemento de la expresión (convertida a notación postfija):
  Si es un operando
  entonces guardar en la pila (en formato de número)
  en caso contrario
```

Quitar un elemento de la pila (será el operando 2)
Quitar un elemento de la pila (será el operando 1)
Operar los operandos según el operador (suma, resta, producto, división)
Agregar a la pila el resultado obtenido
Fin del ciclo Repetir
Quitar un elemento de la pila, el cual es el resultado

El código correspondiente a este algoritmo puede consultarse en el proyecto EstructurasDatosBásicas de NetBeans, en el paquete cap7, clase *Calculadora* (programa 7.4). Como puede observarse, en el método *evalúa()* se utiliza una pila de tipo *Double* (clase de Java) para almacenar los operandos y los resultados parciales que se van obteniendo a medida que se evalúa la expresión.

En los tres algoritmos previamente analizados se usaron pilas para almacenar temporalmente algunos valores. Esto se debe a que la característica de las pilas –el último dato insertado es el primero que puede quitarse– facilita de manera implícita las operaciones deseadas. Se sugiere revisar la clase *Calculadora* (programa 7.4) y la clase *UsaCalculadora* (programa 7.5) para comprender mejor lo expuesto en esta sección. En la primera de ellas se encuentran todos los métodos explicados y algunos auxiliares, mientras que la segunda presenta algunos ejemplos de uso de la *Calculadora*, en los casos de que la expresión esté correctamente formada o no. Las dos clases se encuentran en el cap7 del paquete antes paquete mencionado.

7.3 COLA

Una cola es una estructura de datos lineal en la cual se identifican dos extremos y las operaciones de inserción y eliminación se asocian a cada uno de ellos. Esta característica determina que el primer elemento insertado sea el primero que puede quitarse. De ahí que a estas estructuras también se las conozca con el nombre de FIFO (First-In, First-Out). Uno de los extremos es el principio de la cola y se utiliza para quitar elementos, mientras que el otro es el fin de la misma y se usa para agregar nuevos elementos.

En la vida real encontramos numerosos ejemplos donde se usa este concepto. Por ejemplo, piense en una cola de personas que están esperando para comprar una entrada para el cine o una cola de clientes de un banco que están esperando ser atendidos por el cajero, o una cola de autobuses escolares que se disponen a dejar la escuela. En el área de computación hay varias aplicaciones importantes de las colas, mismas que se tratarán un poco más adelante en este capítulo.

Figura 7.8 Ejemplo de cola: cola de camiones escolares

Independientemente del tipo de dato que almacene, la cola mostrará el mismo comportamiento. Observe la figura 7.9 (a) en la cual se insertaron los elementos 1, 2 y 3 en ese orden, por lo tanto el *frente* apunta al primer elemento guardado en la cola (1), y el *fin* al último (3). Si se hace una eliminación, el que saldrá será el 1, quedando la cola como se muestra en (b). Por último, si se inserta el 4 la cola queda como se muestra en (c).

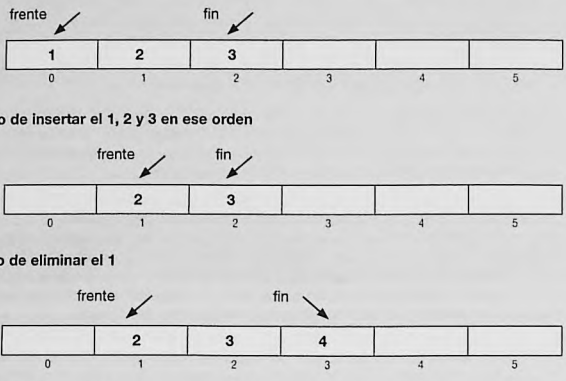


Figura 7.9 Comportamiento de una cola

Muy importante

- Estructura de datos abstracta.
- Estructura lineal.
- Tiene dos extremos: uno para insertar y otro para eliminar elementos.
- Los extremos se llaman *frente* y *fin*.
- El primer elemento almacenado es el primero en salir.
- Las colas también se conocen como estructuras FIFO (First-In, First-Out).

7.3.1 Implementación de una cola

Antes de estudiar con mayor detalle las operaciones que pueden hacerse en una cola, es necesario decidir cómo implementarlas, ya que de esto dependerá en gran medida cómo llevar a cabo cada una de las operaciones en esta estructura de datos.

En la figura 7.10 se presenta el diagrama UML de la interface *ColaADT* y de las clases *ColaA* y *ColaE*. Con la interface se define el comportamiento esperado de una cola por medio de las firmas de los métodos: *quita()*, agrega(), *primero()* y *estáVacía()*. Estos métodos permiten quitar un elemento, agregar un nuevo dato, consultar el valor guardado en el frente y consultar si la cola está vacía, respectivamente. Las dos clases representan el concepto de una cola, la primera de ellas implementándola por medio de un arreglo y la segunda a través de una estructura enlazada.

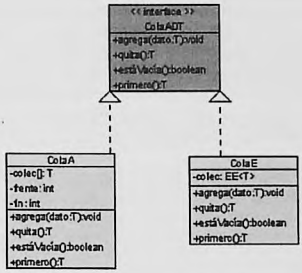


Figura 7.10 Diagrama UML para representar la interface *ColaADT* y las clases que la implementan

En la interface se establece el comportamiento esperado de una cola, sin importar la estructura de datos que se elija para su implementación. En el programa 7.6 se presenta esta interface. Observe que la interface está prevista para almacenar cualquier tipo de datos y que además incluye métodos para agregar y quitar elementos, así como dos métodos auxiliares para consultar si la cola está vacía y el elemento que está en el frente, respectivamente.

Programa 7.6 **ColaADT.java**

```
package cap7;

/**
 * @author Silvia Guardati
 * Programa 7.6
 * Se define el comportamiento esperado en una estructura de datos tipo Cola.
 */
public interface ColaADT<T> {
    public void agrega(T dato); // Agrega un elemento al final de la cola.
    public T quita( ); // Quita y regresa el elemento que está en el frente de la cola.
    public boolean estáVacía( ); // Regresa true si no hay elementos
    public T primero( ); // Regresa el elemento que está en el frente, sin quitarlo.
}
```

¿Cómo implementar una cola, considerando que es una estructura abstracta? Anteriormente se dijo que es una estructura de datos lineal, por lo que existen dos maneras para hacerlo: por medio de arreglos y por medio de estructuras enlazadas. Según lo indicado cuando se presentaron las pilas, las ventajas y desventajas de cada una de las implementaciones estarán determinadas por la estructura base elegida.

La implementación por medio de un arreglo se muestra más abajo. Observe que sólo se incluyeron los atributos y los constructores. Los métodos que implementan las operaciones antes mencionadas se agregarán luego de que se analicen en la siguiente sección. Un poco más adelante, en el programa 7.7, ColaA.java, se presenta la clase completa, con atributos y métodos.

```
public class ColaA<T> implements ColaADT<T>{
    private T cola[ ];
    privateint frente, fin;
    private final int MAX = 10;

/* Se construye un arreglo de Object y se lo convierte explícitamente a tipo T.
 * Inicialmente la cola está vacía, lo cual se indica con -1 en frente y fin.
 */
  public ColaA( ) {
    cola = (T[ ]) new Object[MAX];
    frente = -1;
    fin = -1;
  }

/* Se construye un arreglo de Object y se lo convierte explícitamente a tipo T. El tamaño
  * máximo para el arreglo está dado por el parámetro max.
  * Inicialmente la cola está vacía, lo cual se indica con -1 en frente y fin.
  */
  public ColaA(int max) {
    cola = (T[ ]) new Object[max];
    frente = -1;
    fin = -1;
  }
}
```

La clase cuenta con tres atributos: uno para almacenar la colección de datos, para lo cual usa un arreglo, y los otros dos para guardar el valor del *frente* y del *fin*. Adicionalmente se define una constante como auxiliar en la construcción del arreglo en caso de que el usuario no dé un valor para el máximo número de casillas. En los constructores los extremos se inicializan con el valor de -1 para indicar que la cola está vacía. Es importante tener en cuenta que *frente* y *fin* **apuntan** al primero y último elementos insertados, respectivamente.

Se retoma la cola de la figura 7.9 y se le inserta el 5 y el 6 y, posteriormente, se le quitan 2 elementos. Luego de estas operaciones la cola queda como se muestra a continuación:

Observe que si ahora se quisiera agregar un nuevo elemento no se podría porque el *fin* está en el límite del arreglo. Debido a eso ya no es posible incrementarlo para asignar el nuevo valor en esa posición. Sin embargo, como es evidente, hay 3 lugares disponibles. Para resolver esta inconsistencia se debe tratar el arreglo como una estructura circular, es decir, el siguiente elemento del último es el primero. Para lograrlo se debe incrementar el puntero por medio de la expresión:

$$fin = (fin + 1) \% cola.length$$

Observe el siguiente ejemplo basado en el arreglo anterior:

fin	cola.length	fin luego de actualizarse
5	6	(5 + 1) % 6 = 0
0	6	(0 + 1) % 6 = 1
1	6	(1 + 1) % 6 = 2

En el ejemplo se puede apreciar que, con el incremento y el módulo (%), el puntero va tomando todos los valores posibles de los índices. Además, cuando llega al último valor se pasa al 0 (primera posición del arreglo). Lo mismo aplica para el *frente* cuando se elimina un dato.

A continuación se presenta la implementación de la cola por medio de una estructura enlazada. Observe que sólo se incluyeron los atributos y los constructores. Los métodos que implementan las operaciones antes mencionadas se agregarán luego de que se analicen en la siguiente sección. Un poco más adelante, en el programa 7.8, ColaE.java, se presenta la clase completa, con atributos y métodos.

```
public class ColaE<T> implements ColaADT<T>{
  private EE <T> cola;

  // Se construye un objeto tipo estructura enlazada (EE).
  publicColaE( ) {
     cola = new EE( );
  }
}
```

7.3.2 Operaciones en una cola

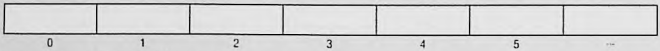
Al inicio del capítulo se presentó el comportamiento esperado de las colas, mencionando que las mismas deben poder agregar o quitar elementos utilizando un extremo diferente para cada una de las operaciones. Además de estas operaciones, generalmente se definen algunas auxiliares. Una de ellas se utiliza para determinar si la cola está vacía y otra para consultar el elemento que está en el frente sin necesidad de quitarlo.

« Agregar un elemento a la cola

Este método agrega un elemento al final de la cola, actualizando el valor del *fin*. Para implementar esta operación se debe tener en cuenta la estructura de datos sobre la que se trabajará. Por lo tanto, primero se muestra la versión para arreglos y, posteriormente, la correspondiente a estructuras enlazadas.

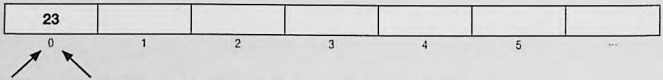
En el caso del arreglo, considerando su naturaleza estática, se debe verificar que la cola no esté llena. Si es así, se incrementa el puntero al final (*fin*) y en esa posición se asigna el dato. Recuerde que el *fin* debe apuntar al último elemento insertado. En el caso de que el arreglo esté lleno, se deberá construir un arreglo de mayor capacidad y copiar en éste los valores ya almacenados. Luego se sigue con los pasos ya indicados; es decir, se incrementa en 1 el *fin* y en esa posición se asigna el nuevo valor. En la figura 7.11 se muestra esta operación en una cola implementada por medio de un arreglo.

a. Estado inicial de la cola:



frente = fin = -1 (indica cola vacía.)

b. Estado de la cola luego de insertar el 23. Observe que cuando es el primer elemento se debe modificar también el valor de *frente*.



c. Estado de la cola luego de insertar el 41.

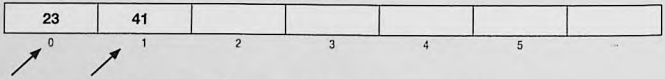


Figura 7.11 Operación ***agrega()*** en una cola implementada con un arreglo

El algoritmo en pseudocódigo es el siguiente:

1. Si cola está llena, se construye un arreglo de mayor capacidad y se copian en él los elementos ya almacenados. Se actualizan los valores del *frente* y *fin*.
2. Si está vacía a *frente* se le asigna 0.
3. Se incrementa en 1 el *fin*.
4. Se asigna el nuevo valor en la posición señalada por *fin*.

Antes de presentar el código se analiza en qué situaciones la cola puede estar llena; observe la figura 7.12. En el primer caso el *frente* está al inicio del arreglo y el *fin* en la última posición, mientras que en el segundo caso el *frente* está una posición a la derecha del *fin* (esto sucede por el tratamiento circular del arreglo).

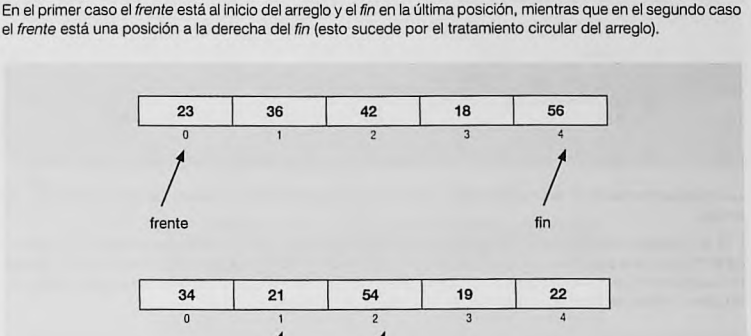


Figura 7.12 Cola llena – situaciones posibles

En ambos casos, el estado de cola llena se detecta evaluando la expresión:

$$\textbf{(fin + 1) \% arreglo.length == frente}$$

Observe que para el primer arreglo *fin* = 4, por lo tanto (fin +1) es 5, y 5 % 5 = 0 que es el valor de *frente*. En el segundo, *fin* = 1, por lo tanto (*fin* + 1) = 2 y 2 % 5 = 2 que es el valor de *frente*.

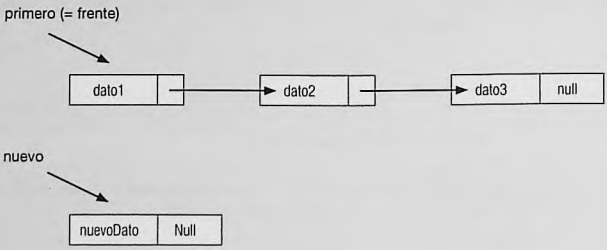
A continuación se presenta el código en Java para implementar el método *agrega()* correspondiente a la clase *ColaA*.

```
/* Agrega un dato al final de la cola, actualizando el puntero correspondiente.
 * Si la cola estuviera vacía, debe también actualizar el frente. Si la cola está llena,
 * primero se crea un arreglo de mayor tamaño y luego se agrega el nuevo elemento.
 */
   public void agrega(T dato) {
      if ((fin + 1) % cola.length  == frente) // Cola llena
          aumentaCapacidad( );
      else
          if (estáVacía( ))  // Cola vacía
              frente = 0;
      fin = (fin + 1) % cola.length;
      cola[fin] = dato;
   }
```

La actualización del *fin* se hace de acuerdo con lo mencionado más arriba, logrando así el manejo circular del arreglo.

Si la implementación se basa en una estructura enlazada, entonces se procederá a invocar el método *agregaFinal()* de la estructura, ya que el final será el único extremo por el cual se agregarán elementos. Recuerde que este método crea un nuevo nodo que almacena el dato dado y posteriormente liga al último nodo con el nuevo. Gráficamente se ve como muestra la figura 7.13.

a. Se construye un nuevo nodo y se le asigna el dato que quiere agregarse en la cola.



b. Se accede al último nodo y se establece como su sucesor al nuevo nodo.

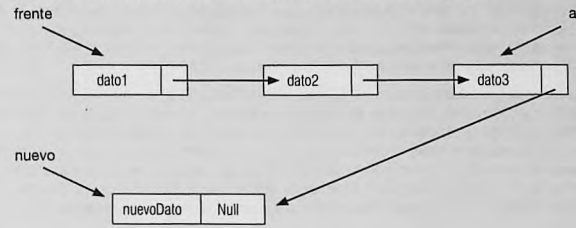


Figura 7.13 Operación *agrega()* en una cola implementada con una estructura enlazada

A continuación se presenta el código en Java para implementar el método *agrega()* correspondiente a la clase *ColaE*.

```
// Se agrega el dato al final de la cola.
public void agrega(T dato) {
  cola.agregaFinal(dato);
}
```

Muy importante

- Para obtener mayor eficiencia se sugiere al lector que modifique la clase *EE* de tal manera que tenga un puntero al primer nodo y otro al último. De esta forma, el método *agregaFinal()* no deberá recorrer todos los nodos para llegar al último.
- Revise todos los métodos de la clase *EE* y modifíquelos si corresponde, considerando el nuevo atributo agregado.

« Quitar un elemento de la cola

Este método quita el elemento que está en el *frente* de la cola, actualizando el valor del puntero. Si la cola tiene al menos un elemento, el método regresa el valor quitado; en caso contrario, lanza una excepción indicando que la colección está vacía. Para la excepción se reusa la clase *ExcepciónColecciónVacía* definida en el capítulo anterior (puede consultarse en el paquete cap6, programa 6.4).

En el caso de la clase implementada por medio del arreglo, si la cola está vacía se lanza una excepción terminando la operación. En caso contrario, se guarda en una variable auxiliar el contenido del *frente* y se actualiza el valor de éste, incrementándolo en 1, de tal manera que ahora apunte al siguiente elemento disponible de la cola. Al arreglo, como en el caso de la inserción, se lo trata como a una estructura circular para lograr así mayor eficiencia en el manejo de la memoria. Observe la figura 7.14. En (a) se muestra el estado de la cola antes de ejecutar eliminar un elemento, siendo el *frente* = 0 e indicando éste el primer valor que podrá quitarse. Luego de eliminar el elemento del frente, la cola queda como se muestra en (b). El valor de *frente* ahora es 1.

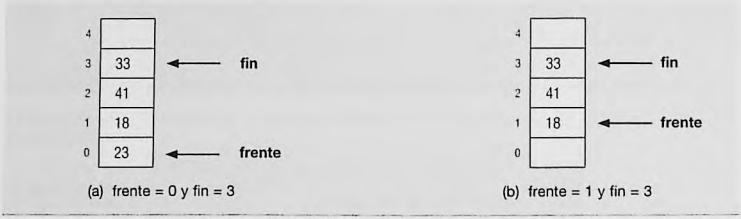


Figura 7.14 Operación *quita()* en una cola implementada con un arreglo

El algoritmo en pseudocódigo es el siguiente:

1. Si cola está vacía, se lanza una excepción.
2. Se almacena en una variable auxiliar el contenido de la casilla cuyo índice es *frente*.
3. Si la cola tiene un único elemento

   entonces se le asigna -1 a *frente* y a *fin*, indicando que la cola quedó vacía;

   en caso contrario se redefine el valor de *frente* incrementándolo en 1.

El código en Java correspondiente al método *quita()* de la clase *ColaA* queda como se muestra a continuación.

```
/* Elimina y regresa el elemento que está en el frente de la cola.
 * Si la cola está vacía, lanza una excepción.
 */
  public T quita( ) {
```

```
        if (estáVacía( ))
            throw new ExcepciónColecciónVacía("Cola vacía");
        else {
            T resul = cola[frente];
            if (frente == fin){  // Tiene un único elemento.
                frente = -1;
                fin = -1;
            }
            else
                frente = (frente + 1) % cola.length;
            return resul;
        }
    }
```

Cuando se trata de una cola implementada por medio de una estructura enlazada, si la misma está vacía se lanza una excepción terminando la operación. En caso contrario, el método regresa el resultado del método *quitaPrimero()* de la clase *EE*. Recuerde que este último regresa el contenido del primer nodo y mueve el puntero hacia el siguiente nodo de la estructura. En la figura 7.15 se puede observar en (a) el estado original de la cola y en (b) el estado de la misma luego de eliminar el elemento que está en el *frente*. Ahora *frente* apunta al siguiente elemento (dato2) y se perdió la relación entre el dato1 (eliminado) y los demás datos.

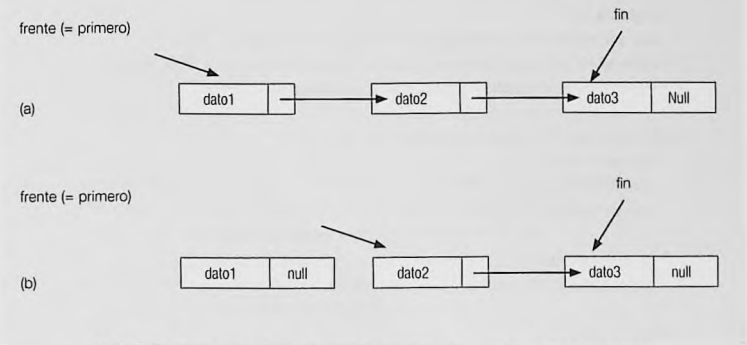


Figura 7.15 Operación *quita()* en una cola implementada con una estructura enlazada

El código en Java correspondiente al método *quita()* de la clase *ColaE* queda como se muestra a continuación.

```
/* Elimina y regresa el dato que está en el frente de la cola.
 * Si la cola está vacía lanza una excepción.
 */
  public T quita( ) {
    if (estáVacía( ))
       throw new ExcepciónColecciónVacía("Cola vacía");
    else
      return cola.quitaPrimero( );
  }
```

Una vez analizadas las operaciones de inserción y eliminación en una cola, se presentan las clases *ColaA* y *ColaE* con todos sus métodos en los programas 7.7 y 7.8, respectivamente.

Programa 7.7 | **ColaA.java**

```
package cap7;
import cap6.ExcepciónColecciónVacía;

/**
 * @author Silvia Guardati
 * Programa 7.7
 * Clase que implementa una cola genérica basándose en un arreglo.
 * Para un mejor aprovechamiento del espacio, el arreglo se maneja de manera circular.
 * Es decir, la siguiente casilla después de la última es la primera.
*/
public class ColaA<T> implements ColaADT<T>{
      private T cola[ ];
      privateint frente, fin;
      private final int MAX = 10;

  /* Se construye un arreglo de Object y se lo convierte explícitamente a tipo T.
   * Inicialmente la cola está vacía, lo cual se indica con -1 en frente y fin.
   */
  public ColaA( ) {
```

```
        cola = (T[ ]) new Object[MAX];
        frente = -1;
        fin = -1;
    }

    /* Se construye un arreglo de objetos y se lo convierte explícitamente a tipo T.
     * El espacio máximo reservado queda determinado por el parámetro max.
     * Inicialmente la cola está vacía, lo que se indica con -1 en frente y fin.
     */
    public ColaA(int max) {
        cola = (T[ ]) new Object[max];
        frente = -1;
        fin = -1;
    }

/* Agrega un dato al final de la cola, actualizando el puntero correspondiente.
 * Si la cola estuviera vacía, debe también actualizar el frente. Si la cola está llena,
 * primero se crea un arreglo de mayor tamaño y luego se agrega el nuevo elemento.
 */
    public void agrega(T dato) {
        if ((fin + 1) % cola.length  == frente) // Cola llena
            aumentaCapacidad( );
        else
            if (estáVacía( ))
                frente = 0;
        fin = (fin + 1) % cola.length;
        cola[fin] = dato;
}

    /* Método auxiliar que construye un arreglo de mayor tamaño y copia en él todos
     * los elementos de la cola, asignando al arreglo cola la referencia del nuevo
     * arreglo. Además, actualiza los punteros al frente y al final de la cola.
     */
    private void aumentaCapacidad( ){
        T nuevo[ ] = (T[ ]) (new Object[cola.length * 2]);
        int j;
```

```
    j = 0;
    while (!estáVacía( )){
            nuevo[j] = quita( );
            j++;
    }
    cola = nuevo;
    frente = 0;
    fin = j - 1;
    }

    // Regresa true si la cola no tiene elementos almacenados.
    public boolean estáVacía( ) {
          return frente == -1;
    }

    /* Regresa el dato que está en el frente de la cola. Si la cola está vacía
     * lanza una excepción.
     */
    public T primero( ) {
          if (estáVacía( ))
             throw new ExcepciónColecciónVacía("Cola vacía");
          else
             return cola[frente];
    }

    /* Elimina y regresa el elemento que está en el frente de la cola.
     * Si la cola está vacía, lanza una excepción.
     */
    public T quita( ) {
          if (estáVacía( ))
             throw new ExcepciónColecciónVacía("Cola vacía");
          else {
             T resul = cola[frente];
             if (frente == fin){  // Tiene un único elemento.
                frente = -1;
                fin = -1;
```

```
    }
    else
        frente = (frente + 1) % cola.length;
    return resul;
        }
      }
    }
```

Programa 7.8 **ColaE.java**

```
package cap7;
import cap6.EE;
import cap6.ExcepciónColecciónVacía;
/*
 * @author Silvia Guardati
* Programa 7.8
 * Implementación de la estructura Cola por medio de una estructura enlazada.
*/
public class ColaE<T> implements ColaADT<T>{
      private EE <T> cola;

// Se construye un objeto tipo estructura enlazada (EE).
  publicColaE( ) {
       cola = new EE( );
}

// Se agrega el dato al final de la cola.
public void agrega(T dato) {
     cola.agregaFinal(dato);
}

 // Regresa true si la cola está vacía.
public boolean estáVacía( ) {
     return cola.estáVacía( );
}
```

```
/* Regresa el dato que está en el frente de la cola.
 * Si la cola está vacía lanza una excepción.
 */
public T primero( ) {
    if (estáVacía( ))
      throw new ExcepciónColecciónVacía("Cola vacía");
    else
      return cola.getElemento(1);
}

/* Elimina y regresa el dato que está en el frente de la cola.
 * Si la cola está vacía lanza una excepción.
 */
public T quita( ) {
    if (estáVacía( ))
      throw new ExcepciónColecciónVacía("Cola vacía");
    else
      return cola.quitaPrimero( );
  }
}
```

Muy importante

- En los constructores –implementación con arreglo– el *frente* y el *fin* se inicializan con -1, indicando cola vacía.
- En el constructor –implementación con estructura enlazada– se debe construir un objeto de tipo *EE*.
- El *frente* apunta al primer elemento insertado en la cola y, por lo tanto, será el siguiente elemento que puede quitarse.
- El *fin* apunta al último elemento insertado en la cola.
- En una cola implementada por medio de un arreglo los espacios se manejan circularmente. Es decir, el siguiente elemento del último es el primero.

7.3.3 Aplicaciones de colas

El concepto de cola se utiliza mucho en diversas áreas. Uno de sus usos más comunes es para organizar la atención, según el orden de llegada, de pacientes, clientes, objetos a ser reparados, etc. En el área de la computación una aplicación muy conocida es la cola de impresión, la cual se genera cuando existe una impresora atendiendo a varios usuarios y lo hace de acuerdo con el orden en que se solicitaron los servicios. A continuación se presenta un ejemplo muy simple de una aplicación en que se simula una cola de impresión.

Se crea la clase *Archivo* para representar, de manera simplificada, un archivo por medio del nombre, tipo y tamaño. Además, se define la clase *Impresora* que tiene tres métodos y, entre sus atributos, una cola de archivos. El primero de los métodos es para almacenar los archivos que deben ser impresos, el segundo es para atender la impresión de los mismos de acuerdo con el orden en que se recibieron, y el último quita todos los archivos que estén en la cola de impresión y que sean del tipo dado, regresando la cantidad de archivos eliminados. La figura 7.16 presenta el diagrama UML de las clases descritas. Los programas 7.9 y 7.10, correspondientes a estas dos clases, pueden consultarse en el paquete cap7.

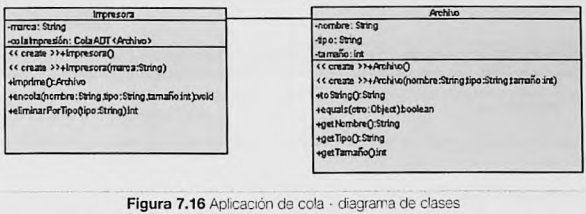


Figura 7.16 Aplicación de cola - diagrama de clases

En el programa 7.11 (usa impresora) que se encuentra en el paquete cap 7 se presenta un ejemplo de aplicación muy simple de las clases precedentes. En esta aplicación se genera aleatoriamente un número entero comprendido entre 0 y 3 inclusive. El 0 indica que ingresa una solicitud de impresión de un archivo. Se pide el nombre, tipo y tamaño y se agrega a la cola de impresión de la impresora construida. El 1 indica que la impresora está disponible, por lo tanto se saca un archivo de la cola y se imprime. Si la cola está vacía (no hay archivos pendientes de impresión) se despliega un mensaje adecuado. El 2 representa la tercera funcionalidad de la impresora, que es eliminar todos los archivos que sean de un cierto tipo. Para ello se le pide al usuario que ingrese el tipo y se invoca al método que elimina por tipo, dando como resultado el total de archivos eliminados. Finalmente, el 3 se usa para terminar con la ejecución. Es importante tener en cuenta que como los números se generan aleatoriamente puede suceder que en una corrida el primer valor generado sea el 3 y, por lo tanto, el programa termine sin ejecutar ninguna de las otras operaciones. Se sugiere al lector revisar y probar el código.

7.3.4 Doble cola

Una doble cola es una generalización de la estructura tipo cola. Se distingue porque las operaciones de inserción y eliminación pueden hacerse por cualquiera de los extremos. Por lo tanto, se puede insertar y eliminar por el final y por el frente. En consecuencia, si se inserta por un extremo y luego se elimina por ese mismo extremo, al menos en ese caso la cola se estaría comportando como una pila. Gráficamente una doble cola se representa como lo muestra la figura 7.17.

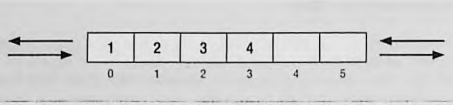


Figura 7.17 Doble cola

Este tipo de estructura también es abstracta, por lo tanto para su implementación se usan los arreglos o las estructuras enlazadas. En las clases correspondientes se tendrán dos versiones de los métodos de inserción y eliminación para que ambas operaciones puedan aplicarse por los dos extremos. En el caso de los arreglos, es recomendable que los mismos se traten circularmente para optimizar el espacio usado. A continuación se presenta la interface *ColaDobleADT*. Observe que la interface es genérica y contiene los mismos métodos que la interface *ColaADT* adaptados para cumplir con las características de este tipo de estructura de datos. Queda a cargo del lector la implementación de esta interface por medio de dos clases, la primera usando un arreglo circular y la segunda una estructura enlazada.

Programa 7.12 **ColaDobleADT.java**

```
package cap7;

/**
 * @author Silvia Guardati
 * Programa 7.12
 * Definición de la interface correspondiente a una estructura tipo Cola Doble.
 */
public interface ColaDobleADT<T>{
    public voidagregaFrente(T dato); // Agrega un elemento al inicio de la cola.
    public voidagregaFin(T dato); // Agrega un elemento al final de la cola.
    public T quitaFrente( ); // Quita y regresa el elemento que está en el frente de la cola.
    public T quitaFin( ); // Quita y regresa el elemento que está al final de la cola.
    public boolean estáVacía( ); // Regresa true si no hay elementos
    public T primeroFrente( ); // Regresa el elemento que está en el frente, sin quitarlo.
    public T primeroFin( ); // Regresa el elemento que está al final, sin quitarlo.
}
```

A su vez, las dobles colas pueden ser:

- Con entrada restringida
- Con salida restringida

La primera de ellas sólo permite que la inserción se lleve a cabo por el final (como lo establece una estructura tipo cola), mientras que la eliminación puede hacerse por cualquiera de los dos extremos. La segunda variante mantiene la eliminación restringida al frente, mientras que la inserción puede llevarse a cabo tanto por el frente como por el final.

La implementación de estas dos variantes es una simplificación de la implementación de una doble cola. Eliminando el método *agregaFrente()* se genera la versión correspondiente a la entrada restringida, y sacando el método *quitaFin()* se obtiene la de salida restringida.

7.4 RESUMEN

En este capítulo se presentaron las estructuras de datos abstractas pilas y colas con el enfoque del paradigma de la programación orientada a objetos. En ambos casos, por ser abstractas, se presentó su implementación por medio de arreglos y de estructuras enlazadas. Asimismo, se estudiaron las principales operaciones que pueden realizarse sobre estas estructuras.

Los temas cubiertos en este capítulo se complementaron con un ejemplo de aplicación que favorece la comprensión de los mismos.

7.5 EJERCICIOS

7.1 Escriba un método estático que reciba un objeto tipo *PilaA* y lo modifique de tal manera que la pila quede con los elementos en orden inverso. Es decir, si la pila original contiene 1,2,3 en ese orden, luego de aplicado el método deberá quedar como 3,2,1. ¿Qué estructura de datos auxiliar puede usar?

7.2 Escriba un método estático que elimine todos los elementos repetidos de un objeto tipo *PilaE*. Suponga que si existen elementos repetidos, estos se encuentran en posiciones consecutivas. ¿Qué estructura de datos auxiliar puede usar?

7.3 Escriba un método estático booleano que reciba como parámetro dos objetos tipo *PilaADT* y regrese true si las pilas son iguales. En caso contrario deberá regresar false. Cuide la eficiencia. Dos pilas son iguales si tienen los mismos elementos, en contenido y orden. Al finalizar el método, los objetos recibidos deben permanecer sin cambios.

7.4 Escriba un método estático entero que reciba como parámetro un objeto tipo *PilaADT* y regrese como resultado el total de elementos que contiene dicha pila. Observe que el método no debe modificar al objeto; es decir, una vez ejecutado, la pila debe quedar como estaba originalmente.

7.5 Escriba un método estático que reciba un objeto tipo *ColaA* y lo modifique de tal manera que la cola quede con los elementos en orden inverso. Es decir, si la cola original contiene 1,2,3 en ese orden, luego de aplicado el método la cola deberá quedar como 3,2,1. ¿Qué estructura de datos auxiliar puede usar?

7.6 Escriba un método estático que elimine todos los elementos repetidos de un objeto tipo *ColaE*. Suponga que si existen elementos repetidos, estos se encuentran en posiciones consecutivas. ¿Qué estructura de datos auxiliar puede usar?

7.7 Escriba un método estático booleano que reciba como parámetro dos objetos tipo *ColaADT* y regrese true si las colas son iguales. En caso contrario, deberá regresar false. Cuide la eficiencia. Dos colas son iguales si tienen los mismos elementos, en contenido y orden. Al finalizar el método, los objetos recibidos deben permanecer sin cambios.

7.8 Escriba un método estático booleano que reciba como parámetros dos objetos tipo *ColaE* (c1 y c2) y regrese true si c1 contiene a c2. Una cola contiene a otra si tiene, al menos, todos los elementos que tiene la segunda cola. Cuide la eficiencia. Al finalizar el método, los objetos recibidos deben permanecer sin cambios.

7.9 Escriba un método estático que reciba un objeto tipo *ColaA* y un dato y que elimine todas las ocurrencias de dicho dato. Es decir, si la cola contiene 1 o más veces al dato, se deberán eliminar todos dejando los demás elementos en el orden original.

7.10 Defina las clases *Auto* y *CentroVerificContaminación* de acuerdo con el diagrama de la figura 7.18. Observe que la clase *CentroVerificContaminación* tiene, entre sus atributos, una cola de autos que deberán pasar el control de contaminación. Esta clase debe tener, entre sus funcionalidades, métodos que permitan agregar autos a la cola a medida que lleguen al centro de verificación, así como sacar autos de la cola en cuanto pasen el control. Posteriormente escriba una aplicación para probar su solución.

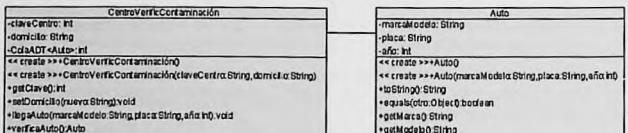


Figura 7.18 Diagrama de clases: *CentroVerificContaminación* y *Auto*

8 RECURSIÓN

Contenido

Competencias

- Explicar el concepto de recursión.
- Presentar los principios de una solución recursiva correctamente planteada.
- Presentar las ventajas y desventajas de las soluciones recursivas.

8.1 INTRODUCCIÓN

Este capítulo está dedicado al estudio de la recursión y de su uso para la solución de cierto tipo de problemas. La recursión se define como la propiedad de un problema por la cual la solución de dicho problema se plantea en términos del mismo problema. En el contexto de la programación, un método es recursivo cuando se llama a sí mismo.

Si bien la recursión es una herramienta poderosa para resolver problemas, también es cierto que en algunos casos consume muchos recursos durante su ejecución. Por tanto, es necesario ser cautelosos y usarla sólo en aquellas situaciones en que los beneficios superen el costo. Para la explicación del concepto de recursión se usarán algunos ejemplos que son muy didácticos; sin embargo, no son soluciones eficientes. Es decir, su solución iterativa es computacionalmente más eficiente, como se explicará más adelante.

8.2 PROBLEMAS RECURSIVOS

Existe cierto tipo de problemas cuya solución se expresa en términos del mismo problema; por ello, se dice que son problemas naturalmente recursivos. Sin embargo, esto no implica necesariamente que la solución debe ser recursiva. A continuación se explica el concepto por medio de un problema fácil de entender y que, por tanto, resulta muy didáctico para introducir el tema.

Considere la suma de los primeros *n* números enteros positivos. Si *n* es 1, entonces la suma(*n*) = 1; si *n* es 2, entonces la suma(*n*) = 2 + suma (1) y, así sucesivamente. En general, la suma(n) = n + suma(n-1). Por lo tanto, la solución queda expresada de la siguiente manera:

suma(n) = n + suma(n – 1)
suma(n – 1) = (n – 1) + suma (n – 2)
...
suma(2) = 2 + suma(1)
suma(1) = 1

Para n = 3 queda:
suma(3) = 3 + suma(2)
suma(2) = 2 + suma(1)
suma(1) = 1
Una vez resuelto suma(1) se pueden calcular las sumas anteriores, obteniendo:
suma(2) = 2 + suma(1) = 2 + 1 = 3
Y trasladando el resultado de suma(2) al cálculo de suma(3), se obtiene:
suma(3) = 3 + suma(2) = 3 + 3 = 6

En el problema anterior se puede identificar los elementos que deben estar presentes en toda solución recursiva correctamente planteada, los cuales se listan a continuación:

- **Estado base:** aquel en el cual se conoce la solución o su valor se puede obtener de manera directa. Al estado base también se le conoce como *condición de parada*, ya que es el que determina que se interrumpa la ejecución.
- **Estado recursivo:** aquel en el que la solución se expresa en función del problema, lo que implica que debe continuar la ejecución.
- **Acercamiento al estado base:** son una o más instrucciones (o expresiones) que permiten que la entrada del método cambie hasta llegar al estado base. Estas instrucciones son las que garantizan que el problema alcance la solución.

El problema presentado más arriba tiene una solución recursiva correcta, ya que:

- **Estado base:** cuando *n* es 1, ya que suma(1) = 1 –la recursión se interrumpe–.
- **Estado recursivo:** cuando n > 1, ya que suma(*n*) = n + suma(*n-1*) –el problema aparece en la solución del mismo, está en el lado derecho de la expresión–.
- **Acercamiento al estado base:** con la resta (*n-1*), ya que siendo *n* un valor positivo, seguramente alcanzará el valor de 1

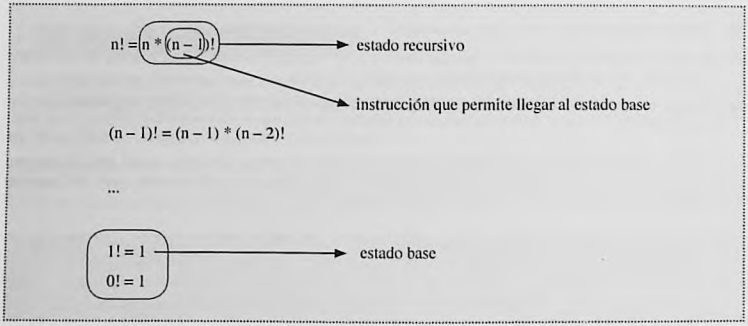
El código correspondiente a este método es el siguiente. El código completo, junto con un método main( ) para probarlo, se encuentra en el programa 8.1, PruebaRecursión, del paquete cap8 del proyecto EstructurasDatosBásicas de Netbeans que complementa este libro.

```
/* Calcula y regresa la suma de los primeros n enteros positivos, siendo n el
* valor que se recibe como parámetro.
*/
   public static int sumaPrimerosEnteros(int n){
      if (n == 1) { // Estado base
         return n;
      }
      else {
         // Estado recursivo y acercamiento al estado base
         return n + sumaPrimerosEnteros(n - 1);
      }
   }
```

Otro ejemplo muy simple y muy ilustrativo es el cálculo del factorial de un número, el cual se define de la siguiente manera:

$$n! = n * (n - 1)!, \text{ si } n > 1$$
$$n! = 1, \text{ si } n = 1 \text{ o } n = 0$$

Observe que en la primera expresión, el problema a resolver (el factorial) aparece también como parte de la solución, siendo así una solución recursiva. Sin embargo, el problema se reduce, ya que la *n* se decrementa y, por tanto, se acerca al estado base.



Como se mencionó, el problema (calcular el factorial de *n*) aparece en el lado derecho de la expresión, como parte de la solución; en este caso, calcular el factorial de (n – 1). Seguramente la solución llega a un valor final conocido debido a que n – 1, siendo n > 0, es un valor que se va acercando a 1, que es el estado base. Como en el caso anterior, se identifica claramente los estados base y recursivo y la instrucción que permite llegar al estado base. A continuación se muestra el método correspondiente para calcular el factorial de un entero positivo. El código puede probarse en el programa 8.1, PruebaRecursión.java, del paquete cap8 del proyecto de Netbeans que complementa este libro.

```
/* Calcula y regresa el factorial de un número entero. El resultado se declara
* de tipo double para tener mayor capacidad de almacenamiento.
*/
   public static double factorial(int n){
     if (n==0 || n== 1) { // Estado base
```

```
        return 1;
    }
    else {
        return n * factorial(n-1); // Estado recursivo y acercamiento al estado base
    }
}
```

A continuación se presenta un tercer ejemplo usando un arreglo de enteros. El método suma recursivamente todos los números almacenados en el arreglo. ¿Cuál es el estado base? Que el arreglo no tenga elementos, en cuyo caso el resultado es 0. Si tuviera un solo elemento, la suma es dicho elemento. Si tuviera 2, la suma sería el valor que está en la segunda casilla, más la suma del contenido de la primera casilla. Si el arreglo tuviera n elementos, entonces la suma es el valor que está en la casilla (n-1), más la suma de las (n-1) casillas anteriores. Es decir, como el arreglo se va recortando, en algún punto alcanzará el estado base.

```
suma(arreglo, 0) = 0   // Estado base
...
suma(arreglo, n) = arreglo[n-1] + suma(arreglo, n-1)  // Estado recursivo
```

El código correspondiente a este método se muestra más abajo. El mismo se encuentra en el programa 8.1, PruebaRecursión.java, del paquete cap8.

```
// Calcula y regresa la suma de los elementos de un arreglo.
public static double sumaArre(double arre[], int n) {
    if (n == 0) {  // Estado base
        return 0;
    }
    else {
        // Estado recursivo y acercamiento al estado base
        return arre[n-1] + sumaArre(arre, n-1);
    }
}
```

Muy importante

- Una solución recursiva debe tener:
  - Un estado base, en el cual se interrumpe la recursión.
  - Un estado recursivo, en el cual el problema aparece como parte de la solución.
  - Al menos una instrucción que permita llegar al estado base.

8.3 REPRESENTACIÓN GRÁFICA DE LA PILA INTERNA DE LA RECURSIÓN

En ciertas soluciones recursivas, quizás en la mayoría, se tiene un gran consumo de memoria debido a la manera en que se ejecutan estos métodos. Retomemos el caso del cálculo del factorial de un número y supongamos que se desea calcular el factorial de 4. Por lo tanto, aplicando la expresión vista, queda como se muestra a continuación:

$$4! = 4 * 3!$$
$$3! = 3 * 2!$$
$$2! = 2 * 1!$$
$$1! = 1$$

La entrada del método será el 4; sin embargo, no es posible realizar la multiplicación hasta que se calcule el factorial de 3. Al calcular el factorial de 3 pasa algo similar: se debe multiplicar 3 por el factorial de 2, el cual aún no se conoce. El proceso se repite hasta llegar al estado base. ¿Dónde queda lo que está pendiente de ejecutarse? La respuesta es que todas las operaciones pendientes de ejecución se guardan en una pila interna hasta que se puedan ejecutar y se disponga del valor requerido para ello. Observe la figura 8.1, en ella aparece un mapa de memoria y una representación gráfica de la pila interna. En la columna etiquetada con *n* se presenta el valor que va tomando el parámetro cada vez que se ejecuta el método. En la otra columna se explica lo que pasa internamente.

n	Comentario
4	Se guarda en la pila 4 * lo que resulte de calcular el factorial de 3 (elemento 0 de la pila (a)). Se hace la llamada recursiva con n = 3.
3	Se guarda en la pila 3 * lo que resulte de calcular el factorial de 2 (elemento 1 de la pila (a)). Se hace la llamada recursiva con n = 2.
2	Se guarda en la pila 2 * lo que resulte de calcular el factorial de 1 (elemento 2 de la pila (a)). Se hace la llamada recursiva con n = 1.

continúa...

1	El método regresa 1 como resultado (estado base) y comienzan a quitarse y a ejecutarse las operaciones guardadas en la pila. Primero se saca y se evalúa la operación que está en la posición 2 de la pila (b), ya que el fact(1) tiene solución. El resultado de esta multiplicación es 2 y pasa a ser el resultado de fact(2), que estaba pendiente en la posición 1 de la pila (b). Se quita esta operación y se multiplica para obtener 6, el cual es el resultado de fact(3). Con este valor se puede ejecutar la última instrucción pendiente (a su vez la primera guardada en la pila), que es la de la posición 0. Al evaluarse, se obtiene el 24 que será el resultado final del método.

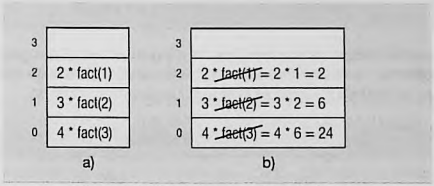


Figura 8.1 Mapa de memoria y pila interna

Como se puede apreciar en la figura 8.1, se utiliza la pila para guardar todas aquellas instrucciones que no pueden ejecutarse hasta llegar al estado base. Una vez alcanzado éste, se van sacando y ejecutando dichas instrucciones. El orden en el cual se ejecutan está determinado por el uso de la pila. Por tanto, se ejecuta desde la última (primera en ejecutarse) hasta la primera almacenada (última en ejecutarse).

Si bien en el ejemplo anterior se ve claramente el uso de la pila interna, hay otros ejemplos en los cuales es más evidente el consumo de memoria al implementar métodos recursivos. Considere la serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21,... en la cual se definen los dos primeros términos y los sucesivos se definen como la suma de los dos anteriores, tal como se muestra a continuación:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = fibonacci(1) + fibonacci(0) = 1 + 0 = 1
...
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

El código en Java resulta simple y, como puede observarse, es casi una traducción directa de la expresión que aparece más arriba. Este método también se encuentra en el programa 8.1, Prueba Recursión.java.

```
// Calcula y regresa el fibonacci de un número entero.
public static int fibonacci(int n) {
    if (n == 0 || n== 1) { // Estado base
```

```
        return n;
    }
    else {
        // Estado recursivo y acercamiento al estado base
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

En la figura 8.2 se presenta el mapa de memoria para n = 4 y una representación gráfica de la pila interna. En la columna etiquetada con *n* se va mostrando el valor que va tomando el parámetro cada vez que se ejecuta el método, mientras que en la segunda columna se explica el funcionamiento interno.

n	Comentario
4	Se guarda en la pila fibo(3) + fibo(2) –elemento 0 de la pila (a)–, ejecutándose la primera de las llamadas y quedando pendiente la segunda llamada y la suma.
3	Se guarda en la pila fibo(2) + fibo(1) –elemento 1 de la pila (a)–, ejecutándose la primera de las llamadas y quedando pendiente la segunda llamada y la suma.
2	Se guarda en la pila fibo(1) + fibo(0) –elemento 2 de la pila (a)–, ejecutándose la primera de las llamadas y quedando pendiente la segunda llamada y la suma.
1	El método regresa *1* como resultado (estado base) y comienza a quitar y a ejecutar las operaciones guardadas en la pila. Primero se saca y se evalúa la llamada que está en la posición 2 de la pila, que es fibo(0).
0	El método regresa *0* como resultado (estado base), y por tanto, puede realizarse la suma. Observe la posición 2 de la pila en (b). El resultado de la suma es fibo(2), por tanto, ahora se quita de la pila la segunda llamada de la posición 1 –fibo(1)–.
1	El método regresa *1* (estado base), y por tanto, puede realizarse la suma. Observe la posición 1 de la pila en (b). El resultado de la suma es fibo(3), por tanto, ahora se quita de la pila la segunda llamada de la posición 0 –fibo(2)–.
2	Se guarda en la pila fibo(1) + fibo(0) –elemento 0 de la pila en (c)–, ejecutándose la primera de las llamadas y quedando pendiente la segunda llamada y la suma.
1	El método regresa *1* como resultado (estado base) y comienza a quitar y a ejecutar las operaciones guardadas en la pila. Primero se saca y se evalúa la llamada que está en la posición 0 de la pila (c), que es fibo(0).
0	El método regresa *0* como resultado (estado base), y por tanto, puede realizarse la suma. El valor obtenido, 1, es el resultado de fibo(2), que está en la posición 0 de (b). Por tanto, ahora se hace la última suma pendiente –posición 0 de la pila (b)– y se obtiene el resultado final, que es 3.

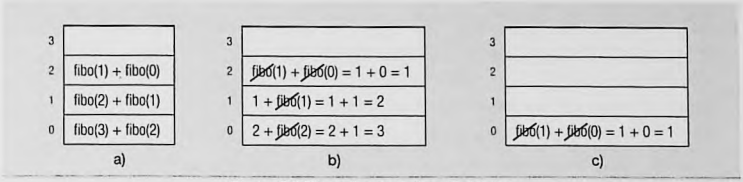
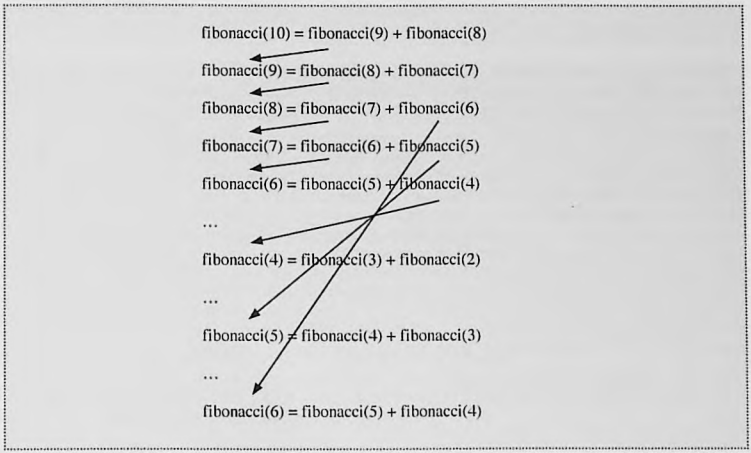


Figura 8.2 Mapa de memoria y pila interna

Por razones de claridad, se dibujaron tres pilas; sin embargo ,es importante notar que internamente se usa una sola, en la cual se va agregando o quitando elementos a medida que se ejecuta el método.

En el caso del ejemplo anterior, además del espacio de memoria que se consume con el uso de la pila interna, existe un costo adicional, debido a que el método se ejecuta varias veces con la misma entrada. Para *n* igual a 4, el fibonacci(2) se ejecutó dos veces, lo mismo para el de 1 y 0. Si se hubiera probado con una *n* mayor, se tendrían más repeticiones. Analice el siguiente ejemplo, tomando n = 10.

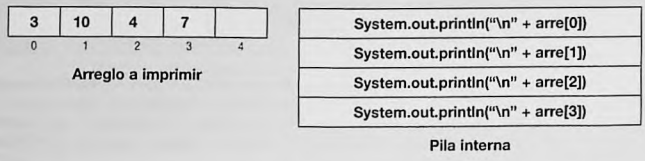


El manejo de la pila interna no depende del desarrollador; sin embargo, es muy importante conocer el funcionamiento interno para aprovecharlo en la solución de cierto tipo de problemas. Supongamos que se quiere imprimir el contenido de un arreglo usando sólo dos parámetros. A continuación se presentan dos alternativas de métodos para llevar a cabo esta actividad. El código puede consultarse y probarse en el programa PruebaRecursión.java del cap8.

```
//Imprime recursivamente un arreglo, de derecha a izquierda.
public static void impArreDI(double arre[], int n) {
   if (n>0) {
      System.out.println("\n" + arre[n-1]);
      impArreDI(arre, n-1);
   }
}

   //Imprime recursivamente un arreglo, de izquierda a derecha.
public static void impArreID(double arre[], int n) {
   if (n>0) {
      impArreID(arre, n-1);
      System.out.println("\n" + arre[n-1]);
   }
}
```

Observe que en la primera solución no se usa la pila interna, ya que se imprime el elemento de la posición n-1 y se regresa a ejecutar el método sin dejar nada pendiente de ejecución. El resultado de este método es la impresión de todos los elementos comenzando por el de la posición n-1 y terminando con el de la posición 0.

En el caso de la segunda implementación, es justamente el uso de la pila lo que permite la impresión de izquierda a derecha, sin el uso de parámetros auxiliares. Cuando *n* es mayor que 0 (hay elementos para imprimir), se regresa al método, quedando en la pila interna la impresión del elemento n-1. Por ser una pila, una vez que se llegue al estado base, los elementos se irán imprimiendo en el orden inverso en que se guardaron. Es decir, el último elemento almacenado en la pila (el valor del arreglo que está en la posición 0) será el primero que se imprima, mientras que el elemento del arreglo de la posición n-1 será el último en imprimirse. Consecuentemente, el resultado de este método es la impresión de todos los elementos, comenzando por el de la posición 0 y terminando con el de la posición n-1. Considere el ejemplo de la figura 8.3. Se tiene un arreglo de 4 elementos que deben imprimirse de izquierda a derecha, es decir, deben desplegarse así: 3, 10, 4 y 7. En la figura se muestra una representación gráfica de la pila interna, en la cual se van a ir guardando las impresiones pendientes de ejecución, así como un mapa de memoria en que se presentan los valores que va tomando el parámetro, junto a un comentario en el cual se explica lo que sucede en el método.



n	Comentario
4	Se ejecuta la llamada recursiva con n-1 (3) y queda en la pila la impresión de la posición n-1(3) del arreglo.
3	Se ejecuta la llamada recursiva con n-1 (2) y queda en la pila la impresión de la posición n-1 (2) del arreglo.
2	Se ejecuta la llamada recursiva con n-1 (1) y queda en la pila la impresión de la posición n-1 (1) del arreglo.
1	Se ejecuta la llamada recursiva con n-1 (0) y queda en la pila la impresión de la posición n-1 (0) del arreglo.
0	No se cumple la condición que controla que el método se ejecute. Consecuentemente, termina. Sin embargo, en la pila interna hay instrucciones pendientes de ejecución, por tanto, se van sacando de la pila y se van ejecutando. En este caso, se imprimirá arre[0], arre[1], arre[2] y, finalmente, arre[3], en ese orden.

Figura 8.3 Mapa de memoria. Uso de la pila interna para resolver un problema

Es importante mencionar que se puede obtener el mismo resultado usando un parámetro extra, como se muestra más abajo. En esta solución, usando el tercer parámetro, se evita la pila interna, ya que los elementos se van imprimiendo a medida que se va ejecutando el método. Por tanto, esta versión del método es más eficiente en cuanto al uso de memoria; no obstante, es menos elegante al demandar un parámetro adicional. El código puede probarse en el programa 8.1,PruebaRecursión.java.

```
/* Sobrecarga del método para que, usando un parámetro extra, imprima recursivamente
* un arreglo de izquierda a derecha, sin emplear la pila interna.
 * El parámetro indica la posición que debe imprimirse. La primera vez pos es 0.
 */
   public static void impArreIDSinPila(double arre[], int n) {
      impArreIDSinPila(arre, n, 0);
   }

   private static void impArreIDSinPila (double arre[], int n, intpos) {
      if (pos< n) {
         System.out.println("\n" + arre[pos]);
         impArreIDSinPila(arre, n, pos + 1);
      }
}
```

En las tres versiones presentadas del método que imprime el contenido de un arreglo se puede apreciar que el estado base está implícito en el código. Se pregunta por el caso contrario, es decir, cuando hay tareas para ejecutar quedando, de manera implícita, el estado base, que es cuando se interrumpe la recursión.

En el programa 8.1, PruebaRecursión.java, se presentan otros ejemplos de problemas resueltos en forma recursiva. Además, se incluye un método *main* muy simple para probar todos los métodos desarrollados en dicho programa.

8.4 ¿RECURSIÓN O ITERACIÓN?

En las secciones previas se presentaron ejemplos de problemas que pueden resolverse fácilmente tanto recursiva como iterativamente y, en algunos casos, se vio que la solución recursiva implica un costo adicional por el manejo de la pila interna. En este grupo de problemas podemos mencionar el factorial, la serie Fibonacci, la impresión de un arreglo, entre otros.

La mayoría de los problemas iterativos pueden expresarse en términos recursivos, aunque esto no sea lo óptimo. Asimismo, casi cualquier problema recursivo puede formularse iterativamente, aunque resulte en una solución compleja y extensa.

En general, se puede decir que para evaluar la conveniencia o no de escribir una solución recursiva se debe tener en cuenta diversos aspectos de la misma, entre los que cabe mencionar: la legibilidad, eficiencia y complejidad. Es decir, vale la pena cuestionarse: ¿resulta natural la solución escrita recursivamente?, ¿es fácil de entender?, ¿es fácil de mantener o corregir?, ¿consume mucho espacio la pila interna?, ¿el método se ejecuta más de una vez con la misma entrada?

En aquellos problemas en los cuales un mismo subproblema se calcula varias veces, es sumamente recomendable diseñar una solución iterativa. Ejemplo de este tipo de problemas es Fibonacci, tal como se vio en la sección previa. En el caso del cálculo del factorial, la suma de los elementos de un arreglo y problemas similares, también la solución iterativa es más eficiente, ya que no usa la pila interna. Además, estos algoritmos iterativos son muy comprensibles.

Hay otros tipos de problemas cuya solución recursiva es notoriamente más simple y comprensible que su equivalente iterativa. Tal es el caso de las torres de Hanoi y del método de ordenación QuickSort. Es decir, en ambos casos es posible implementar una solución iterativa con la ayuda de pilas auxiliares; sin embargo, la complejidad del algoritmo resultante difícilmente compensa la eficiencia ganada. Estos dos problemas se analizarán en otra sección.

Por otra parte, hay que considerar la existencia de estructuras de datos que son naturalmente recursivas y, por tanto, los algoritmos diseñados para su manipulación resultan mucho más fáciles si se implementan también recursivamente. Ejemplos de estas estructuras son las listas y los árboles. De las estructuras ya estudiadas, las estructuras enlazadas (*EE*) son también un claro ejemplo de estructuras recursivas. Recuerde que un nodo está formado por un dato y un apuntador a un nodo, el cual, a su vez, está formado por un dato y un apuntador a un nodo, y así hasta el último nodo (estado base), el cual tiene un dato y el apuntador tiene el valor *null*. Para ilustrar esta idea se presentan algunos de los métodos de la clase *EE* implementados de manera recursiva. La clase completa se puede consultar en el programa 8.2, EE.java, en el cual se incluyó un *main* muy simple para probar los métodos escritos recursivamente.

```
/* Regresa en forma de cadena la información almacenada en los nodos.
* Implementación recursiva.
*/
public String toString( ){
    Iterator <T> it = iterator( );
    return toString(it);
}

// Sobrecarga del método. Utiliza un iterador para recorrer toda la estructura.
private String toString(Iterator <T> it){
    if (it.hasNext( ))
        return it.next( ) + " " + toString(it);
    else
        return "";
}


/* Elimina el último nodo de la estructura y regresa el dato que almacena.
* Si la estructura está vacía lanza una excepción.
 * Implementación recursiva.
 */
public T quitaUltimo( ){
    if (estáVacía( ))
        throw new ExcepciónColecciónVacía("No hay elementos");
    else
        return quitaUltimo(primero, primero);
}

// Sobrecarga del método.
private T quitaUltimo(Nodo<T> anterior, Nodo<T> actual){
    if (actual.getSig( ) != null)
        return quitaUltimo(actual, actual.getSig( ));
    else{
        T dato = actual.getDato( );
        if (primero == actual)
            primero = null;
```

```
            else
        anterior.setSig(null);
            return dato;
    }
}

    // Busca el dato en la estructura enlazada. Implementación recursiva.
    public boolean busca(T dato){
            return busca(iterator( ), dato);
    }

    // Sobrecarga del método. Regresa true si lo encuentra y false en caso contrario.
    private boolean busca(Iterator<T> it, T dato){
            if (!it.hasNext( ))
                return false;
            else {
                T elemento = it.next( );
                if (elemento.equals(dato))
                    return true;
                else
                    return busca(it, dato);
                }
    }
```

En los métodos presentados puede observarse que la recursión remplazó los ciclos usados previamente. En el método *toString( )* se usó un iterador para recorrer la estructura, mientras que en *quitaUltimo( )* se emplearon nodos auxiliares.

Muy importante

- ☑ Se recomienda una solución recursiva cuando su equivalente iterativa sea compleja y/o extensa.
- ☑ Se recomienda una solución iterativa cuando la solución diseñada se mantenga simple y legible.
- ☑ Cuando se tengan múltiples llamadas y repetición de llamadas con las mismas entradas (caso Fibonacci), se sugiere evitar las soluciones recursivas.
- ☑ En aquellos casos en los cuales no se usa la pila interna la decisión debe ser sólo por legibilidad y estilo.

8.5 APLICACIÓN DE LA RECURSIÓN EN LA SOLUCIÓN DE PROBLEMAS

De acuerdo con lo que se presentó en la sección anterior, algunos problemas, por su naturaleza, tienen una solución claramente recursiva. Si bien es cierto que en la mayoría de los casos se puede reemplazar la recursión usando explícitamente pilas, también es cierto que la solución generada puede resultar larga y compleja de entender. A continuación se presentan algunos problemas cuya solución recursiva es muy simple y concisa, pero que si la reescribiéramos usando pilas auxiliares perderían estas características.

8.5.1 Torres de Hanoi

Consiste en un juego en el cual hay una serie de discos ordenados de manera creciente en una torre, los que se deben pasar a otra torre siguiendo algunas reglas:

- Se debe mover un disco a la vez.
- Sólo se puede mover el disco que está arriba de todos.
- No puede quedar un disco sobre un disco de menor tamaño que él.
- Se puede usar una tercera torre como auxiliar.

En la figura 8.4 se presenta un esquema de los elementos que participan en el problema. Como se puede observar, hay tres torres, la primera de ellas es donde se encuentran inicialmente los discos, la segunda es donde se pretende dejar los discos (respetando las reglas enunciadas) y la tercera funciona como auxiliar. En la torre original se encuentran *n* (en el ejemplo n = 3) discos que deben pasarse a la torre destino. En la figura 8.5 se muestra la secuencia de pasos que deben realizarse para alcanzar el objetivo.

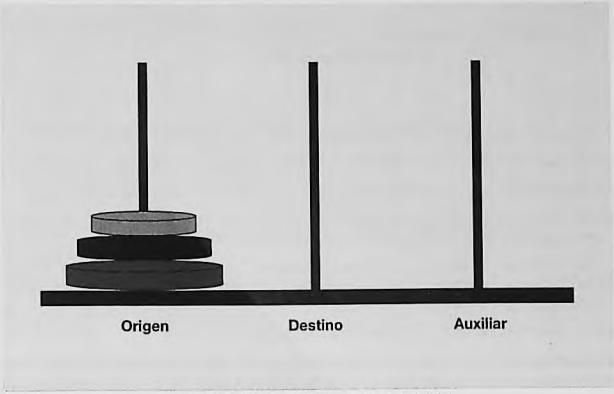


Figura 8.4 Torres de Hanoi – Estado inicial

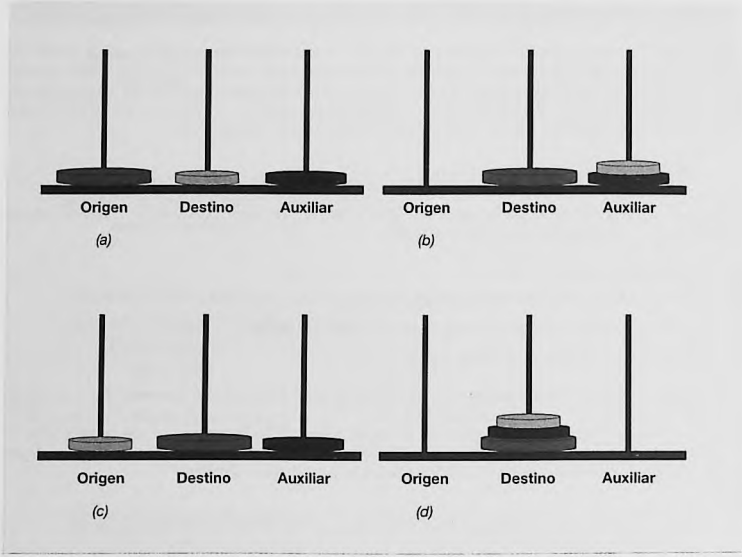


Figura 8.5 Torres de Hanoi – Solución

- Se pasa el disco más pequeño (el que estaba arriba de la torre origen) a la torre destino.
- Se pasa el disco mediano (el que había quedado arriba de la torre origen) a la torre auxiliar.
- Luego de estos dos pasos, los discos quedan ubicados según lo muestra la figura 8.5 *(a)*.
- Se pasa el disco más pequeño a la torre auxiliar. Observe que hasta este punto se pasaron (n-1) discos de la torre origen a la torre auxiliar, usando (como auxiliar) la torre destino.
- Se pasa el disco más grande a la torre destino, quedando como se muestra en *(b)*. En *(b)* se presenta el estado del problema con *n* reducida en 1, ya que ahora hay que pasar (n-1) discos de la torre auxiliar a la torre destino usando la origen como apoyo.
- Se pasa el disco más pequeño a la torre origen, figura *(c)*.
- Se pasa el disco mediano de la torre auxiliar a la torre destino y, por último, se pasa el disco más pequeño de la torre origen a la torre destino, figura *(d)*, alcanzando el objetivo. Es importante destacar que en todos los pasos se respetaron las reglas dadas.

A continuación se presenta el código correspondiente a este problema. El método imprime los movimientos de discos entre las distintas torres y, además, regresa como resultado el total de movimientos realizados. Los parámetros indican el total de discos a mover, el nombre de la torre origen (donde están inicialmente los discos), el nombre de la torre destino (torre en la que deben quedar los discos al finalizar el algoritmo) y el nombre de la torre auxiliar (la que sirve para colocar temporalmente algunos discos). Observe que si se quitara el código dedicado a contar los movimientos, el método sería aún más reducido. Se sugiere al lector consultar y probar el programa 8.3, TorresHanoi, donde se presenta una clase que incluye el método para realizar los movimientos, así como un método *main* para probarlo.

```
/* Método que imprime los movimientos necesarios para mover n discos
 * de la torre Origen a la torre Destino según las reglas señaladas.
 * El método, además de imprimir los movimientos, calcula y regresa como resultado
 * el total de movimientos realizados.
 */
public int mueveTorresDeHanoi( ){
    return mueveTorresDeHanoi(numDiscos, torreO, torreD, torreA);
}

// Sobrecarga del método usado para mover los discos
private int mueveTorresDeHanoi(int n, String tO, String tD, String tA){
    if (n == 1){ // Estado base: hay 1 solo disco
        System.out.println("Mueve de " + tO + " a "+ tD );
        return 1;
    }
else {
    int resp;
    resp = mueveTorresDeHanoi(n-1, tO, tA, tD); // Estado recursivo

    System.out.println("Mueve de " + tO + " a "+ tD);

    resp = resp + 1 + mueveTorresDeHanoi(n-1, tA, tD, tO); // Estado recursivo
    return resp;
    }
}
```

Este problema también puede resolverse sin usar recursión. En ese caso, la función que cumplen las pilas internas debe llevarse a cabo por pilas definidas con ese propósito. La solución que se obtiene es mucho más compleja y extensa.

8.5.2 Método de ordenación Quicksort

Este método se utiliza para ordenar un arreglo. Él mismo está basado en una técnica de programación llamada *divide y vencerás*, que consiste en ir dividiendo el problema en subproblemas cada vez más simples. El proceso se repite hasta llegar, eventualmente, a problemas de fácil solución. Una vez alcanzado este punto, las soluciones de cada uno de los subproblemas se combinan para generar la solución del problema original.

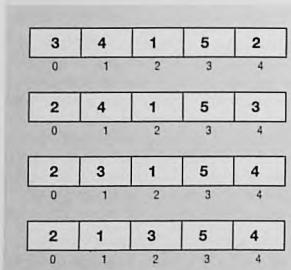
En el capítulo 5 de este libro se presentó el método de ordenación por selección directa, como una alternativa para dejar un arreglo ordenado de manera creciente o decreciente. El método que nos ocupa en esta sección es más eficiente que el anterior, aunque también es más complejo. Básicamente, opera de la siguiente manera: se debe elegir un valor (entre los elementos del arreglo) de tal modo que todos los datos que estén a su izquierda sean menores o iguales a él y los que estén a su derecha sean mayores o iguales. A este elemento se le llama, generalmente, *pivote*. El proceso se repite para todos los datos del arreglo que estén a la derecha y a la izquierda del pivote. De esta manera se tiene una solución naturalmente recursiva, ya que en cada parte del arreglo (la que quedó a la izquierda y la quedó a la derecha) se vuelve a aplicar los pasos arriba mencionados. Como el espacio se va reduciendo, en algún momento todos los elementos del arreglo estarán en la posición adecuada, según su valor, quedando consecuentemente el arreglo ordenado.

Se retoma la clase *AGO (Arreglo Genérico Ordenado)*, en la cual se incluye un método para ordenar de acuerdo con el algoritmo descrito. El código del método *Quicksort* se presenta a continuación. Se sugiere al lector analizar el programa 8.4, AGO.java, en el cual se incluyó el método y un *main* sencillo para probarlo. Además, en este programa se reescribieron algunos de los métodos de la clase, reemplazando ciclos por recursión (métodos de búsqueda secuencial y binaria, *toString* y ordenación por selección directa). Se sugiere al lector revisar estas implementaciones.

```
// Ordena aplicando el método Quicksort.
public void ordenaQuickSort( ){
    ordenaQuickSort(0, total - 1);
}

// Sobrecarga del método para usar la recursión.
private void ordenaQuickSort(int inicio, int fin){
    int izq, der, pivote;
    boolean bandera;

    izq = inicio;
    der = fin;
    pivote = inicio;
    bandera = true;
```

```
        while (bandera){
            bandera = false;
            while (pivote != der &&colec[pivote].compareTo(colec[der]) <= 0)
                der--;
            if (pivote != der){
                intercambia(pivote, der);
                pivote = der;
                while (pivote != izq&&colec[pivote].compareTo(colec[izq]) >= 0)
                    izq++;
                if (pivote != izq){
                    bandera = true;
                    intercambia(pivote, izq);
                    pivote = izq;
                }
            }
        }
        if (pivote -1 > inicio)
                ordenaQuickSort(inicio, pivote - 1);
        if (pivote + 1 < fin)
                ordenaQuickSort(pivote + 1, fin);
    }

    /* Método auxiliar que se utiliza para intercambiar el contenido de dos casillas
     * del arreglo.
     */
    private void intercambia(int pos1, int pos2){
        T temporal = colec[pos1];
        colec[pos1] = colec[pos2];
        colec[pos2] = temporal;
    }
```

Analice el siguiente ejemplo. Se tiene un arreglo de cinco elementos, tal como se muestra en *(a)*, de la figura 8.6. En *(b)*, *(c)* y *(d)* se observa cómo va quedando el arreglo a medida que se aplican intercambios entre sus elementos, según lo indicado por el algoritmo del Quicksort. En la figura 8.7 se presenta el mapa de memoria correspondiente al seguimiento del algoritmo para el arreglo *(a)*.



a. Estado inicial.

b. Luego de intercambiar el contenido de las posiciones 0 y 4.

c. Luego de intercambiar el contenido de las posiciones 1 y 4.

d. Luego de intercambiar el contenido de las posiciones 1 y 2. El valor 3 quedó ubicado de tal manera que todos los elementos que están a su izquierda son menores que él y los que están a su derecha son mayores.

Figura 8.6 Diferentes estados del arreglo a medida que se ejecuta el método Quicksort

inicio	fin	izq	der	pivote	bandera	Comentario
0	4	0	4	0	true	Se compara arreglo[0]<arreglo[4], no entra al *while* y hace el intercambio *(b)*.
		1		4	false	Mientras arreglo[4] > arreglo[izq] se incrementa izq. Cuando izq = 1 se sale del *while*, hace el intercambio, redefine el valor de pivote y cambia el valor de la bandera *(c)*.
			3	1		Mientras arreglo[1] < arreglo[der] se decrementa der.
			2	2		Cuando der = 2 se sale del *while*, se intercambia y se redefine el valor de pivote *(d)*.
		2			false	Mientras arreglo[2] > arreglo[izq] se incrementa izq. Cuando izq = pivote se sale del *while*. No hay intercambio.
						Como der = pivote no entra al ciclo. Como bandera = false se sale del ciclo. El pivote (3) quedó en la posición correcta del arreglo, asegurando que todos a su izquierda son menores y a su derecha son mayores.
						Se evalúa si pivote-1> inicio. En este caso se cumple, por lo que llama recursivamente al método con (0, 1). Queda pendiente en la pila interna evaluar si pivote+1 < fin. Cuando se evalúe será verdadero; por tanto, se hará la llamada recursiva con (3, 4). Queda a cargo del lector el seguimiento de las llamadas recursivas.

Figura 8.7 Mapa de memoria para el método de ordenación Quicksort

En el mapa de memoria de la figura 8.7 se presentó el seguimiento del algoritmo para la primera entrada al método. Como resultado de esta ejecución quedó el primer pivote ubicado en el lugar que le corresponde. Posteriormente, se evalúa el espacio que quedó a su izquierda y el que quedó a su derecha. En cada caso, si todavía hay elementos por ordenar, vuelve a llamar recursivamente al método redefiniendo el inicio y el fin del espacio a ordenar. Se sugiere al lector realizar estos seguimientos para afianzar el funcionamiento del Quicksort.

8.6 TIPOS DE RECURSIÓN

La recursión se clasifica en distintos tipos, de acuerdo con diversos criterios. La recursión *lineal* es aquella en que la solución recursiva genera, máximo, otra llamada recursiva. Un ejemplo de problema cuya solución es de este tipo es el factorial.

Otro tipo de recursión es la *múltiple*; es aquella en la cual una llamada recursiva genera más de una llamada al método. Un ejemplo de este tipo, ya estudiado, es el de la serie de Fibonacci. Recuerde que en la llamada recursiva se invoca al método con n-1 y con n-2.

La recursión *mutua* es cuando dos o más métodos se llaman mutuamente. Un ejemplo muy conocido de este tipo de recursión es determinar si un número es par o no por medio de dos funciones. En el código que aparece más abajo se definieron dos funciones, una para números impares y otra para pares. Sin embargo, cada una de ellas llama a la otra, aplicando así la recursión mutua. Estos métodos pueden probarse en el programa 8.1, PruebaRecursión.java, del paquete cap8.

```
/* Método que determina si un número es impar. Para ello se apoya en otro método:
 * esPar( ), el cual, a su vez, puede volver a llamar a éste.
 * Estos dos métodos son un ejemplo de recursión mutua.
 */
  public static boolean esImpar (int numero){
     if (numero == 0)
        return false;
     else
        return esPar(numero-1);
  }

  /* Método que determina si un número es par, apoyándose en el método esImpar( ) y
   * usando recursión mutua.
   */
public static boolean esPar (int numero){
  if (numero == 0)
     return true;
  else
     return esImpar(numero-1);
  }
```

La recursión *directa* es cuando un método se llama a sí mismo, como en el caso del factorial, las torres de Hanoi, entre otros. Mientras que la recursión *indirecta* es cuando el método llama a otro, el que a su vez llama al primero. Ejemplo de este último es el que se presentó más arriba, el cual consiste en determinar la paridad de un número.

La recursión *de cola* se presenta cuando al hacer la llamada recursiva no se deja ninguna instrucción pendiente de ejecución. Los algoritmos presentados para imprimir un arreglo de derecha a izquierda y el correspondiente a la búsqueda de un elemento en una estructura enlazada son ejemplos de este tipo de recursión.

8.7 RESUMEN

En este capítulo se presentó la recursión como una nueva manera de resolver problemas, explicando el funcionamiento interno de la misma para comprender el costo de los recursos computacionales que se consumen. También se presentaron las ventajas y desventajas que tienen las soluciones recursivas, de acuerdo con el tipo de problemas en los que se aplican. Como complemento, en todos los temas se mostraron ejemplos que ayudan a una mayor comprensión de los conceptos estudiados.

8.8 EJERCICIOS

8.1 En los problemas que se dan a continuación, la descripción del problema es naturalmente recursiva. A pesar de que se pueden plantear soluciones iterativas, con el propósito de practicar el concepto expuesto se pide que diseñe, implemente y pruebe algoritmos recursivos.

a. Calcular el máximo común divisor, MCD (x, y), de acuerdo con la siguiente expresión:

MCD (m, 0) = m

MCD (m, n) = MCD (n, m % n), si $n \neq 0$

b. Calcular los coeficientes binomiales, C (m, n), de acuerdo con la siguiente expresión:

C (m, 0) = 1, si $m \geq 0$

C (m, n) = 0, si $n > m$

C (m, n) = C (m-1, n-1) + C (m-1, n), en otro caso

c. Calcular la función de Ackermman, Ack (m, n), de acuerdo con la siguiente expresión:

Ack (0, n) = n + 1

Ack (m, 0) = Ack (m-1, 1)

Ack (m, n) = Ack (m-1, Ack (m, n-1)), si m, $n > 0$

8.2 Escriba un método estático recursivo que reciba un objeto tipo *PilaADT* y la vacíe. Es decir, debe quitar todos los elementos de la pila.

8.3 Escriba un método estático recursivo que calcule y regrese como resultado el total de elementos de una estructura enlazada.

8.4 Escriba un método estático recursivo que reciba como parámetros dos objetos tipo *PilaADT* y regrese *true* si las pilas son iguales. En caso contrario, debe regresar *false*. Dos pilas son iguales si tienen los mismos objetos, en la misma cantidad y en el mismo orden. Asegúrese de que, una vez ejecutado el método, las pilas permanecen en su estado original.

8.5 Escriba un método estático recursivo que reciba como parámetros dos objetos tipo *ColaADT* y regrese *true* si las colas son iguales. En caso contrario, debe regresar *false*. Dos colas son iguales si tienen los mismos objetos, en la misma cantidad y en el mismo orden. Asegúrese de que, una vez ejecutado el método, las colas permanecen en su estado original.

8.6 Defina su propia clase *EE*. Incluya un método recursivo que reciba como parámetro un dato tipo *T* y regrese *true* si dicho dato está en la estructura enlazada. En caso contrario, deberá regresar *false*. Cuide la eficiencia. Asegúrese de no alterar la estructura.

8.7 En su clase *EE* agregue un método recursivo que reciba como parámetro un dato tipo *T* y regrese el total de veces que dicho dato está en la estructura enlazada. Asegúrese de no alterar la estructura.

8.8 En la clase *AGO* agregue un método recursivo que reciba como parámetro un objeto de tipo *AGO* y regrese *true* si los arreglos genéricos son iguales o *false* en caso contrario. Dos arreglos son iguales si tienen los mismos valores en las mismas posiciones.

8.9 Defina su propia clase *AGO*. Incluya un método de búsqueda secuencial, implementado de manera recursiva.

8.10 En su clase *AGO* escriba un método de búsqueda binaria, implementado de manera recursiva.

8.11 En su clase *AGO* escriba un método de ordenación por selección directa, implementado de manera recursiva.

ÍNDICE ANALÍTICO

Estructuras de Datos Básicas

PROGRAMACIÓN ORIENTADA A OBJETOS CON Java

En esta obra se presentan los principales conceptos de la programación orienta a objetos (POO) y de las estructuras de datos lineales implementadas usando este paradigma de programación, por tanto está dirigida a quien requiere aprender a programar en un entorno de POO y a programar en Java, también a quien necesita aplicar el paradigma orientado a objetos en el diseño e implementación de soluciones basadas en la programación y a quien busca aplicar estructuras de datos lineales en la solución de problemas.

Aprenda:	**A programar en un entorno de programación orientada a objetos.**
Conozca:	**Los conceptos necesarios para programar en Java.**
Desarrolle:	**Aplicaciones de estructuras de datos lineales en la solución de problemas.**

Silvia Guardati Buemo. Profesora del Departamento Académico de Computación, del Instituto Tecnológico Autónomo de México (ITAM), desde 1988 a la fecha. Es egresada de la Universidad Tecnológica Nacional, Argentina (1984), y del CINVESTAV-IPN, México (1987). Sus áreas de interés son las estructuras de datos, la programación orientada a objetos y la ingeniería de software.

www.alfaomega.com.mx

ÁREA: Computación — SUBÁREA: Lenguajes y Técnicas de Programación

Apoyo en la web

ISBN 978-958-778-073-4

"Te acerca al conocimiento"

Alfaomega Grupo Editor