Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

16

Colecciones de genéricos

Creo que ésta es la colección más extraordinaria de talento, de conocimiento humano, que se haya reunido en la Casa Blanca; con la posible excepción de cuando Thomas Jefferson comió solo.

—John F. Kennedy

Objetivos

En este capítulo aprenderá:

- Qué son las colecciones.
- A utilizar la clase `Arrays` para manipulaciones de arreglos.
- A usar las clases de envoltura de tipos que permiten a los programas procesar valores de datos primitivos como objetos.
- A utilizar las estructuras de datos genéricas prefabricadas del marco de trabajo de colecciones.
- A utilizar iteradores para "recorrer" una colección.
- A utilizar las tablas hash persistentes que se manipulan con objetos de la clase `Properties`.
- Cómo funcionan las envolturas de sincronización y las envolturas modificables.

Plan general

16.1 Introducción

En la sección 7.16 presentamos la colección de genéricos `ArrayList`, que es una estructura de datos tipo arreglo que puede ajustar dinámicamente su tamaño y almacenar referencias a objetos de un tipo que se especifica al momento de crear el objeto `ArrayList`. En este capítulo continuaremos nuestra explicación del **marco de trabajo de colecciones** de Java, que contiene muchas otras estructuras de datos genéricas *prefabricadas*.

Algunos ejemplos de colecciones son sus canciones favoritas almacenadas en su teléfono inteligente o reproductor de audio, su lista de contactos, las tarjetas que posee en un juego de cartas, los miembros de su equipo deportivo favorito y los cursos que tomó alguna vez en la escuela.

En este capítulo hablaremos de las interfaces del marco de trabajo de colecciones, las cuales declaran las posibilidades de cada tipo de colección; también hablaremos de varias clases que implementan a estas interfaces, de los métodos que procesan los objetos de las colecciones y de los **iteradores** que "recorren" las colecciones.

Java SE 8

Después de leer el capítulo 17, Lambdas y flujos de Java SE 8, podrá volver a implementar muchos de los ejemplos del capítulo 16 de una manera más concisa y elegante, y de modo tal que facilite la paralelización para mejorar el desempeño en los sistemas multinúcleo de la actualidad. En el capítulo 23 (en inglés, en el sitio web del libro), aprenderá a mejorar el desempeño en los sistemas multinúcleo mediante el uso de las *colecciones concurrentes* y las operaciones de *flujos paralelos* de Java.

16.2 Generalidades acerca de las colecciones

Una **colección** es una estructura de datos (o un objeto) que puede guardar referencias a otros objetos. Por lo general, las colecciones contienen referencias a cualquier tipo de objeto que tenga la relación "*es un*" con el tipo almacenado en la colección. Las interfaces del marco de trabajo de colecciones declaran las operaciones que se deben realizar en forma genérica en varios tipos de colecciones. La figura 16.1 lista algunas de las interfaces del marco de trabajo de colecciones. Varias implementaciones de estas interfaces se proporcionan dentro del marco de trabajo. Los programadores también pueden proporcionar implementaciones específicas para sus propios requerimientos.

Interfaz	Descripción
`Collection`	La interfaz raíz en la jerarquía de colecciones a partir de la cual se derivan las interfaces `Set`, `Queue` y `List`.
`Set`	Una colección que *no* contiene duplicados.
`List`	Una colección ordenada que *puede* contener elementos duplicados.
`Map`	Una colección que asocia claves con valores y *no puede* contener claves duplicadas. `Map` no se deriva de `Collection`.
`Queue`	Por lo general es una colección del tipo *primero en entrar, primero en salir*, que modela a una *línea de espera*; pueden especificarse otros órdenes.

Fig. 16.1 | Algunas interfaces del marco de trabajo de colecciones.

Colecciones basadas en `Object`

Las clases y las interfaces del marco de trabajo de colecciones son miembros del paquete `java.util`. En versiones anteriores de Java, las clases en el marco de trabajo de colecciones *solamente* almacenaban y manipulaban referencias `Object` (lo que nos permitía almacenar *cualquier* objeto en una colección), ya que todas las clases se derivan de manera directa o indirecta de la clase `Object`. Por lo general, los programas tienen la necesidad de procesar tipos *específicos* de objetos. Como resultado, las referencias `Object` que se obtienen de una colección necesitan *convertirse* en un tipo apropiado para permitir que el programa procese los objetos correctamente. Como vimos en el capítulo 10, por lo general debe evitarse la conversión descendente.

Colecciones de genéricos

Para eliminar este problema, el marco de trabajo de colecciones se mejoró con las herramientas de *genéricos* que presentamos con los objetos `ArrayList` genéricos en el capítulo 7 y que veremos con más detalle en el capítulo 20. Los genéricos nos permiten especificar el *tipo exacto* que se almacenará en una colección y nos dan los beneficios de la *comprobación de tipos en tiempo de ejecución*; el compilador emite mensajes de error si se usan tipos inapropiados en las colecciones. Una vez que especifique el tipo almacenado en una colección, cualquier referencia que obtenga de la colección tendrá ese tipo. Esto elimina la necesidad de conversiones de tipo explícitas que pueden lanzar excepciones `ClassCastException` cuando el objeto referenciado *no* es del tipo apropiado. Además, las colecciones de genéricos son *compatibles con versiones anteriores* de código Java que se haya escrito antes de que se introdujeran los genéricos.

Buena práctica de programación 16.1

Evite reinventar la rueda; en vez de crear sus propias estructuras de datos, use las interfaces y colecciones del marco de trabajo de colecciones de Java que se han probado y optimizado cuidadosamente para satisfacer los requerimientos de la mayoría de las aplicaciones.

Cómo elegir una colección

La documentación de cada colección describe sus requerimientos de memoria y las características de rendimiento de sus métodos para operaciones como agregar y eliminar elementos, buscar elementos, ordenar elementos y más. Antes de elegir una colección, consulte la documentación en línea para la categoría de colecciones que esté considerando (`Set`, `List`, `Map`, `Queue`, etc.) y luego seleccione la implementación que se adapte mejor a las necesidades de su aplicación. El capítulo 19, Búsqueda, ordenamiento y Big O, habla sobre un medio para describir qué tan duro trabaja un algoritmo para realizar

su tarea, con base en el número de elementos de datos que se vayan a procesar. Después de leer el capítulo 19 comprenderá mejor las características de rendimiento de cada colección, según lo descrito en la documentación en línea.

16.3 Clases de envoltura de tipos

Cada tipo primitivo (listado en el apéndice D) tiene su correspondiente **clase de envoltura de tipo** (en el paquete `java.lang`). Estas clases se llaman **`Boolean`**, **`Byte`**, **`Character`**, **`Double`**, **`Float`**, **`Integer`**, **`Long`** y **`Short`**; nos permiten manipular valores de tipos primitivos como objetos. Esto es importante, ya que las estructuras de datos que reutilizamos o desarrollamos en los capítulos 16 a 21 manipulan y comparten *objetos*, ya que no pueden manipular variables de tipos primitivos. Sin embargo, pueden manipular objetos de clases de envoltura de tipos, ya que cada clase se deriva en última instancia de `Object`.

Cada una de las clases de envoltura de tipos numéricos (`Byte`, `Short`, `Integer`, `Long`, `Float` y `Double`) extiende a la clase `Number`. Además, las clases de envoltura de tipos son clases `final`, de modo que no podemos extenderlas. Los tipos primitivos no tienen métodos, por lo que los métodos relacionados con un tipo primitivo se localizan en la clase de envoltura de tipos correspondiente (por ejemplo, el método `parseInt`, que convierte un objeto `String` en un valor `int`, se localiza en la clase `Integer`).

16.4 Autoboxing y autounboxing

Java cuenta con conversiones "boxing" y "unboxing", las cuales realizan conversiones automáticas entre los valores de tipo primitivo y los objetos de envoltura de tipos. Una **conversión boxing** convierte un valor de un tipo primitivo en un objeto de la correspondiente clase de envoltura de tipo. Una **conversión unboxing** convierte un objeto de una clase de envoltura de tipo en un valor del tipo primitivo correspondiente. Estas conversiones se realizan de manera automática: lo que se conoce como **autoboxing** y **autounboxing**. Considere las siguientes instrucciones:

```
Integer[] arregloEntero = new Integer[5]; // crea arregloEntero
arregloEntero[0] = 10; // asigna el Integer 10 a arregloEntero[0]
int valor = arregloEntero[0]; // obtiene el valor del Integer
```

En este caso, la conversión autoboxing ocurre al asignar un valor `int` (`10`) a `arregloEntero[0]`, ya que `arregloEntero` almacena referencias a objetos `Integer`, no valores `int`. La conversión auto-boxing ocurre al asignar `arregloEntero[0]` a la variable `int valor`, ya que esta variable almacena un valor `int`, no una referencia a un objeto `Integer`. Las conversiones boxing también ocurren en las condiciones, que se pueden evaluar como valores `boolean` primitivos u objetos `Boolean`. Muchos de los ejemplos en los capítulos 16 a 21 usan estas conversiones para almacenar valores primitivos en estructuras de datos y recuperarlos de éstas.

16.5 La interfaz Collection y la clase Collections

La interfaz `Collection` contiene **operaciones masivas** (es decir, operaciones que se llevan a cabo en *toda una* colección) para operaciones como *agregar*, *borrar* y *comparar* objetos (o elementos) en una colección. Un objeto `Collection` también puede convertirse en un arreglo. Además la interfaz `Collection` proporciona un método que devuelve un objeto **`Iterator`**, el cual permite a un programa recorrer toda la colección y eliminar elementos de la misma durante la iteración. En la sección 16.6.1 hablaremos sobre la clase `Iterator`. Otros métodos de la interfaz `Collection` permiten a un programa determinar el *tamaño* de una colección, y si está *vacía* o no.

Observación de ingeniería de software 16.1

`Collection` se utiliza comúnmente como un tipo de parámetro de métodos para permitir el procesamiento polimórfico de todos los objetos que implementen a la interfaz `Collection`.

Observación de ingeniería de software 16.2

La mayoría de las implementaciones de colecciones proporcionan un constructor que toma un argumento `Collection`, permitiendo así que se construya una nueva colección, la cual contiene los elementos de la colección especificada.

La clase **`Collections`** proporciona métodos `static` que *buscan*, *ordenan* y realizan otras operaciones sobre las colecciones. En la sección 16.7 hablaremos más acerca de los métodos de `Collections`. También cubriremos los **métodos de envoltura** de la clase `Collections`, los cuales nos permiten tratar a una colección como una *colección sincronizada* (sección 16.13) o una *colección no modificable* (sección 16.14). Las colecciones sincronizadas son para usarse con la tecnología multihilos (que veremos en el capítulo 23), la cual permite a los programas realizar operaciones *en paralelo*. Cuando dos o más hilos de un programa *comparten* una colección, podrían ocurrir problemas. Como una breve analogía, considere una intersección de tráfico. No podemos permitir que todos los automóviles accedan a una intersección al mismo tiempo; si lo hiciéramos, ocurrirían accidentes. Por esta razón, se colocan semáforos en las intersecciones para controlar el acceso a cada intersección. De manera similar, podemos *sincronizar* el acceso a una colección para asegurar que sólo *un* subproceso a la vez manipule la colección. Los métodos de envoltura de sincronización de la clase `Collections` devuelven las versiones sincronizadas de las colecciones que pueden compartirse entre los hilos en un programa. Las colecciones no modificables son útiles cuando un cliente de una clase necesita *ver* los elementos de una colección, pero *no* se le debe permitir que *modifique* la colección, agregando y eliminando elementos.

16.6 Listas

Un objeto `List` (conocido como **secuencia**) es un objeto `Collection` *ordenado* que puede contener elementos duplicados. Al igual que los índices de arreglos, los índices de objetos `List` empiezan desde cero (es decir, el índice del primer elemento es cero). Además de los métodos de interfaz heredados de `Collection`, `List` proporciona métodos para manipular elementos a través de sus índices, para manipular un rango especificado de elementos, para buscar elementos y para obtener un objeto **`ListIterator`** para acceder a los elementos.

La interfaz `List` es implementada por varias clases, incluyendo a **`ArrayList`**, **`LinkedList`** y **`Vector`**. La conversión autoboxing ocurre cuando se agregan valores de tipo primitivo a objetos de estas clases, ya que sólo almacenan referencias a objetos. Las clases `ArrayList` y `Vector` son implementaciones de un objeto `List` como arreglos que pueden modificar su tamaño. Insertar un elemento entre los elementos existentes de un objeto `ArrayList` o `Vector` es una operación *ineficiente*, ya que hay que quitar del camino a todos los elementos que van después del nuevo, lo cual podría ser una operación costosa en una colección con una gran cantidad de elementos. Un objeto `LinkedList` permite la inserción (o eliminación) *eficiente* de elementos a la mitad de una colección, pero es mucho menos eficiente que un objeto `ArrayList` para saltar a un elemento específico en la colección. En el capítulo 21 hablaremos sobre la arquitectura de las listas enlazadas.

`ArrayList` y `Vector` tienen comportamientos casi idénticos. Las operaciones en los objetos de la clase `Vector` están *sincronizadas* de manera predeterminada, mientras que las de los objetos `ArrayList` no. Además, la clase `Vector` es de Java 1.0, antes de que se agregara el marco de trabajo de colecciones a Java. Como tal, `Vector` tiene varios métodos que no forman parte de la interfaz `List` y que no se implementan en la clase `ArrayList`. Por ejemplo, los métodos `addElement` y `add` de `Vector` anexan un elemento a un objeto `Vector`, pero sólo el método `add` está especificado en la interfaz `List` y se implementa mediante `ArrayList`. *Las colecciones desincronizadas proporcionan un mejor rendimiento que las*

sincronizadas. Por esta razón, `ArrayList` se prefiere comúnmente a `Vector` en programas que no comparten una colección entre hilos. La API de colecciones de Java proporciona *envolturas de sincronización* independientes (sección 16.13) que pueden usarse para agregar sincronización a las colecciones desincronizadas; además hay disponibles varias colecciones sincronizadas poderosas en las API de concurrencia de Java.

Tip de rendimiento 16.1

Los objetos `ArrayList` se comportan igual que los objetos `Vector` desincronizados y por lo tanto se ejecutan con más rapidez que los objetos `Vector`, ya que los objetos `ArrayList` no tienen la sobrecarga que implica la sincronización de los subprocesos.

Observación de ingeniería de software 16.3

Los objetos `LinkedList` pueden usarse para crear pilas, colas, árboles y colas con dos partes finales (conocidas como "deques"). El marco de trabajo de colecciones proporciona implementaciones de algunas de estas estructuras de datos.

En las siguientes tres subsecciones se demuestran las herramientas de `List` y `Collection` con varios ejemplos. La sección 16.6.1 se enfoca en eliminar elementos de un objeto `ArrayList` mediante un objeto `Iterator`. La sección 16.6.2 se enfoca en `ListIterator` y varios métodos específicos de `List` y de `LinkedList`.

16.6.1 `ArrayList` e `Iterator`

En la figura 16.2 se utiliza un objeto `ArrayList` (que introdujimos en la sección 7.16) para demostrar varias herramientas de la interfaz `Collection`. El programa coloca dos arreglos `Color` en objetos `ArrayList` y utiliza un objeto `Iterator` para eliminar los elementos en la segunda colección `ArrayList` de la primera colección.

```
1  // Fig. 16.2: PruebaCollection.java
2  // Demostración de la interfaz Collection mediante un objeto ArrayList.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class PruebaCollection
9  {
10    public static void main(String[] args)
11    {
12       // agrega los elementos en el arreglo colores a la lista
13       String[] colors = {"MAGENTA", "ROJO", "BLANCO", "AZUL", "CYAN"};
14       List<String> lista = new ArrayList<String>();
15
16       for (String color : colores)
17          lista.add(color); // agrega el color al final de la lista
18
19       // agrega los elementos en el arreglo eliminarColores a eliminarLista
20       String[] eliminarColores = {"ROJO", "BLANCO", "AZUL"};
21       List<String> eliminarLista = new ArrayList<String>();
22
23       for (String color : eliminarColores)
24          eliminarLista.add(color);
25
```

Fig. 16.2 | Demostración de la interfaz `Collection` mediante un objeto `ArrayList` (parte 1 de 2).

```
26          // imprime en pantalla el contenido de la lista
27          System.out.println("ArrayList: ");
28
29          for (int cuenta = 0; cuenta < lista.size(); cuenta++)
30             System.out.printf("%s ", lista.get(cuenta));
31
32          // elimina de la lista los colores contenidos en eliminarLista
33          eliminarColores(lista, eliminarLista);
34
35          // imprime en pantalla el contenido de la lista
36          System.out.printf("%n%nArrayList despues de llamar a eliminarColores:%n");
37
38          for (String color : lista)
39             System.out.printf("%s ", color);
40       }
41
42       // elimina de coleccion1 los colores especificados en coleccion2
43       private static void eliminarColores(Collection<String> coleccion1,
44          Collection<String> coleccion2)
45       {
46          // obtiene el iterador
47          Iterator<String> iterador = coleccion1.iterator();
48
49          // itera mientras la colección tenga elementos
50          while (iterador.hasNext())
51          {
52             if (coleccion2.contains(iterador.next()))
53                iterador.remove(); // elimina el color actual
54          }
55       }
56    } // fin de la clase PruebaCollection
```

```
ArrayList:
MAGENTA ROJO BLANCO AZUL CYAN

ArrayList despues de llamar a eliminarColores:
MAGENTA CYAN
```

Fig. 16.2 | Demostración de la interfaz `Collection` mediante un objeto `ArrayList` (parte 2 de 2).

En las líneas 13 y 20 se declaran e inicializan los arreglos `String colores` y `eliminarColores`. En las líneas 14 y 21 se crean objetos `ArrayList<String>` y se asignan sus referencias a las variables `ArrayList<String> lista` y `eliminarLista`, respectivamente. Cabe mencionar que `ArrayList` es una clase *genérica*, por lo que podemos especificar un *argumento de tipo* (`String` en este caso) para indicar el tipo de los elementos en cada lista. Puesto que el tipo a almacenar en una colección se especifica en tiempo de compilación, las colecciones genéricas proporcionan una *seguridad de tipos* en tiempo de compilación que permite al compilador atrapar los intentos de usar tipos inválidos. Por ejemplo, no puede almacenar objetos `Empleado` en una colección de objetos `String`.

En las líneas 16 y 17 se llena `lista` con objetos `String` almacenados en el arreglo `colores`, y en las líneas 23 y 24 se llena `eliminarLista` con objetos `String` almacenados en el arreglo `eliminarColores`, usando el **método add de `List`**. En las líneas 29 y 30 se imprime en pantalla cada elemento de `lista`. En la línea 29 se llama al **método `size` de `List`** para obtener el número de elementos del objeto `ArrayList`. En la línea 30 se utiliza el **método `get` de `List`** para obtener valores de elementos individuales. En las líneas

29 y 30 también se pudo haber usado la instrucción `for` mejorada (que demostraremos con las colecciones en otros ejemplos).

En la línea 33 se hace una llamada al método `eliminarColores` (líneas 43 a 55), y se pasan `lista` y `eliminarLista` como argumentos. El método `eliminarColores` elimina los objetos `String` especificados en `eliminarLista` de los objetos `String` en `lista`. En las líneas 38 y 39 se imprimen en pantalla los elementos de `lista`, una vez que `eliminarColores` completa su tarea.

El método `eliminarColores` declara dos parámetros de tipo `Collection<String>` (líneas 43 y 44); pueden pasarse dos objetos `Collection` cualesquiera que contengan objetos `String` como argumentos. El método accede a los elementos del primer objeto `Collection` (`coleccion1`) mediante un objeto `Iterator`. En la línea 47 se llama al **método `iterator` de `Collection`**, el cual obtiene un objeto `Iterator` para el objeto `Collection`. Las interfaces `Collection` e `Iterator` son tipos genéricos. En la condición de continuación de ciclo (línea 50) se hace una llamada al **método `hasNext` de `Iterator`** para determinar si hay más elementos por los cuales iterar. El método `hasNext` devuelve `true` si otro elemento existe, y devuelve `false` en caso contrario.

La condición del `if` en la línea 52 llama al **método `next` de `Iterator`** para obtener una referencia al siguiente elemento, y después utiliza el método **`contains`** del segundo objeto `Collection` (`coleccion2`) para determinar si `coleccion2` contiene el elemento devuelto por `next`. De ser así, en la línea 53 se hace una llamada al **método `remove` de `Iterator`** para eliminar el elemento del objeto `coleccion1` de `Collection`.

Error común de programación 16.1

Si se modifica una colección mediante uno de sus métodos después de crear un iterador para esa colección, el iterador se vuelve inválido de manera inmediata; cualquier operación realizada con el iterador después de este punto lanza una excepción `ConcurrentModificationException`. Por esta razón, se dice que los iteradores son de "falla rápida". Estos iteradores ayudan a asegurar que una colección modificable no sea manipulada por dos o más hilos al mismo tiempo, lo que podría corromper la colección. En el capítulo 23 (en inglés, en el sitio web del libro) aprenderá sobre las colecciones concurrentes (paquete `java.util.concurrent`) que pueden manipularse de manera segura mediante varios hilos concurrentes.

Observación de ingeniería de software 16.4

Nos referimos a los objetos `ArrayList` en este ejemplo mediante variables `List`. Esto hace a nuestro código más flexible y fácil de modificar. Si posteriormente determinamos que serían más apropiados los objetos `LinkedList`, sólo habrá que modificar las líneas en donde creamos los objetos `ArrayList` (líneas 14 y 21). En general, al crear un objeto colección debe referirse a ese objeto con una variable del tipo de interfaz de la colección correspondiente.

Inferencia de tipos con la notación <>

Las líneas 14 y 21 especifican el tipo almacenado en el objeto `ArrayList` (es decir, `String`) en los lados izquierdo y derecho de las instrucciones de inicialización. En Java SE 7 se introdujo la *inferencia de tipos* con la notación <> (conocida como la **notación diamante**) en instrucciones que declaran y crean variables y objetos de tipo genérico. Por ejemplo, la línea 14 podría escribirse así:

```
List<String> lista = new ArrayList<>();
```

En este caso, Java usa el tipo en los paréntesis angulares del lado izquierdo de la declaración (es decir, `String`) mientras se crea el tipo almacenado en el objeto `ArrayList` del lado derecho de la declaración. Usaremos esta sintaxis para el resto de los ejemplos en el capítulo.

16.6.2 LinkedList

En la figura 16.3 se demuestran varias operaciones con objetos `LinkedList`. El programa crea dos objetos `LinkedList` que contienen objetos `String`. Los elementos de un objeto `List` se agregan al otro. Después, todos los objetos `String` se convierten a mayúsculas, y se elimina un rango de elementos.

```
1  // Fig. 16.3: PruebaList.java
2  // Uso de objetos List, LinkedList y ListIterator.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
6
7  public class PruebaList
8  {
9     public static void main(String[] args)
10    {
11       // agrega elementos de colores a lista1
12       String[] colores =
13          {"negro", "amarillo", "verde", "azul", "violeta", "plateado"};
14       List<String> lista1 = new LinkedList<>();
15
16       for (String color : colores)
17          lista1.add(color);
18
19       // agrega elementos de colores2 a lista2
20       String[] colores2 =
21          {"dorado", "blanco", "cafe", "azul", "gris", "plateado"};
22       List<String> lista2 = new LinkedList<>();
23
24       for (String color : colores2)
25          lista2.add(color);
26
27       lista1.addAll(lista2); // concatena las listas
28       lista2 = null; // libera los recursos
29       imprimirLista(lista1); // imprime los elementos de lista1
30
31       convertirCadenasAMayusculas(lista1); // convierte cadena a mayúsculas
32       imprimirLista(lista1); // imprime los elementos de lista1
33
34       System.out.printf("%nEliminando elementos 4 a 6...");
35       eliminarElementos(lista1, 4, 7); // elimina los elementos 4 a 6 de la lista
36       imprimirLista(lista1); // imprime los elementos de lista1
37       imprimirListaInversa(lista1); // imprime la lista en orden inverso
38    }
39
40    // imprime el contenido del objeto List
41    private static void imprimirLista(List<String> lista)
42    {
43       System.out.printf("%nlista:%n");
44
45       for (String color : lista)
46          System.out.printf("%s ", color);
47
48       System.out.println();
49    }
50
51    // localiza los objetos String y los convierte a mayúsculas
52    private static void convertirCadenasAMayusculas(List<String> lista)
53    {
```

Fig. 16.3 | Objetos `List`, `LinkedList` y `ListIterator` (parte 1 de 2).

```
54          ListIterator<String> iterador = lista.listIterator();
55
56          while (iterador.hasNext())
57          {
58             String color = iterador.next(); // obtiene elemento
59             iterador.set(color.toUpperCase()); // convierte a mayúsculas
60          }
61       }
62
63       // obtiene sublista y utiliza el método clear para eliminar los elementos de
            la misma
64       private static void eliminarElementos(List<String> lista,
65          int inicio, int fin)
66       {
67          lista.subList(inicio, fin).clear(); // elimina los elementos
68       }
69
70       // imprime la lista inversa
71       private static void imprimirListaInversa(List<String> lista)
72       {
73          ListIterator<String> iterador = lista.listIterator(lista.size());
74
75          System.out.printf("%nLista inversa:%n");
76
77          // imprime la lista en orden inverso
78          while (iterator.hasPrevious())
79             System.out.printf("%s ", iterador.previous());
80       }
81    } // fin de la clase PruebaList
```

```
lista:
negro amarillo verde azul violeta plateado dorado blanco cafe azul gris plateado

lista:
NEGRO AMARILLO VERDE AZUL VIOLETA PLATEADO DORADO BLANCO CAFE AZUL GRIS PLATEADO

Eliminando elementos 4 a 6...
lista:
NEGRO AMARILLO VERDE AZUL BLANCO CAFE AZUL GRIS PLATEADO

Lista inversa:
PLATEADO GRIS AZUL CAFE BLANCO AZUL VERDE AMARILLO NEGRO
```

Fig. 16.3 | Objetos `List`, `LinkedList` y `ListIterator` (parte 2 de 2).

En las líneas 14 y 22 se crean los objetos `LinkedList` llamados `lista1` y `lista2` de tipo `String`. `LinkedList` es una clase genérica que tiene un parámetro de tipo, para el cual especificamos el argumento de tipo `String` en este ejemplo. En las líneas 16 a 17 y 24 a 25 se hace una llamada al método `add` de `List` para *anexar* elementos de los arreglos `colores` y `colores2` al final de `lista1` y `lista2`, respectivamente.

En la línea 27 se hace una llamada al **método `addAll` de `List`** para *anexar todos los elementos* de `lista2` al final de `lista1`. En la línea 28 se establece `lista2` en `null`, ya que `lista2` no se necesita más. En la línea 29 se hace una llamada al método `imprimirLista` (líneas 41 a 49) para mostrar el contenido de `lista1`. En la línea 31 se hace una llamada al método `convertirCadenaAMayusculas` (líneas 52 a 61) para convertir cada elemento `String` a mayúsculas, y después en la línea 32 se hace una llamada nuevamente a `imprimirLista` para mostrar los objetos `String` modificados. En la línea 35 se hace una llamada

al método `eliminarElementos` (líneas 64 a 68) para eliminar el rango de elementos empezando desde el índice `4` hasta, pero sin incluir, el índice `7` de la lista. En la línea 37 se hace una llamada al método `imprimirListaInversa` (líneas 71 a 80) para imprimir la lista en orden inverso.

Método `convertirCadenasAMayusculas`

El método `convertirCadenasAMayusculas` (líneas 52 a 61) cambia los elementos `String` en minúsculas del argumento `List` por objetos `String` en mayúsculas. En la línea 54 se hace una llamada al **método `listIterator` de `List`** para obtener un **`iterador bidireccional`** (es decir, un iterador que pueda recorrer un objeto `Lista` *hacia delante* o *hacia atrás*) para el objeto `List`. `ListIterator` es también una clase genérica. En este ejemplo, el objeto `ListIterator` hace referencia a objetos `String`, ya que el método `listIterator` se llama en un objeto `List` que contiene objetos `String`. En la línea 56 se hace una llamada al método `hasNext` para determinar si el objeto `List` *contiene otro elemento*. En la línea 58 se obtiene el siguiente objeto `String` en el objeto `List`. En la línea 59 se hace una llamada al **método `toUpperCase` de `String`** para obtener una versión en mayúsculas del objeto `String` y se hace una llamada al **método `set` de `Iterator`** para reemplazar el objeto `String` actual al que hace referencia `iterador` con el objeto `String` devuelto por el método `toUpperCase`. Al igual que el método `toUpperCase`, el **método `toLowerCase` de `String`** devuelve una versión del objeto `String` en minúsculas.

Método `eliminarElementos`

El método `eliminarElementos` (líneas 64 a 68) *elimina un rango de elementos* de la lista. En la línea 67 se hace una llamada al **método `subList` de `List`** para obtener una porción del objeto `List` (lo que se conoce como **sublista**). A esto se le conoce como **método de vista de rango**, el cual permite al programa ver una parte de la lista. La sublista es simplemente otra vista hacia el interior del objeto `List` desde el que se hace la llamada a `subList`. El método `subList` recibe dos argumentos: el índice inicial para la sublista y el índice final. El índice final *no forma* parte del rango de la sublista. En este ejemplo, la línea 35 pasa el `4` para el índice inicial y `7` para el índice final a `subList`. La sublista devuelta es el conjunto de elementos con los índices `4` a `6`. A continuación, el programa hace una llamada al **método `clear` de `List`** en la sublista para eliminar los elementos que ésta contiene del objeto `List`. Cualquier cambio realizado a una sublista se hace en el objeto `List` original.

Método `imprimirListaInversa`

El método `imprimirListaInversa` (líneas 71 a 80) imprime la lista al revés. En la línea 73 se hace una llamada al método `listIterator` de `List` con un argumento que especifica la posición inicial (en nuestro caso, el último elemento en la lista) para obtener un *iterador bidireccional* para la lista. El **método `size` de `List`** devuelve el número de elementos en el objeto `List`. En la condición del ciclo `while` (línea 78) se hace una llamada al **método `hasPrevious` de `ListIterator`** para determinar si hay más elementos mientras se recorre la lista *hacia atrás*. En la línea 79 se hace una llamada al **método `previous` de `ListIterator`** para obtener el elemento anterior de la lista y se envía como salida al flujo de salida estándar.

Vistas en colecciones y el método `asList` de `Arrays`

La clase `Arrays` proporciona el método `static` **`asList`** para *ver* un arreglo (conocido algunas veces como el **arreglo de respaldo**) como una colección **`List`**. Una vista `List` permite al programador manipular el arreglo como si fuera una lista. Esto es útil para agregar los elementos de un arreglo a una colección y para ordenar los elementos del arreglo. En el siguiente ejemplo le demostraremos cómo crear un objeto `LinkedList` con una vista `List` de un arreglo, ya que no podemos pasar el arreglo a un constructor de `LinkedList`. En la figura 16.7 se demuestra cómo ordenar elementos de un arreglo con una vista `List`. Cualquier modificación realizada a través de la vista `List` cambia el arreglo, y cualquier modificación realizada al arreglo cambia la vista `List`. La única operación permitida en la vista devuelta por `asList` es *establecer*, la cual cambia el valor de la vista y del arreglo de soporte. Cualquier otro intento por cambiar la vista (como agregar o eliminar elementos) produce una excepción **`UnsupportedOperationException`**.

Ver arreglos como objetos `List` y convertir objetos `List` en arreglos

En la figura 16.4 se utiliza el método `asList` para ver un arreglo como una colección `List`, y el **método `toArray` de `List`** para obtener un arreglo de una colección `LinkedList`. El programa llama al método `asList` para crear una vista `List` de un arreglo, la cual se utiliza después para crear un objeto `LinkedList`; después agrega una serie de objetos `String` a un objeto `LinkedList` y llama al método `toArray` para obtener un arreglo que contiene referencias a esos objetos `String`.

```
1  // Fig. 16.4: UsoToArray.java
2  // Ver arreglos como objetos List y convertir objetos List en arreglos.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsoToArray
7  {
8     // el constructor crea un objeto LinkedList, le agrega elementos y lo
         convierte en arreglo
9     public static void main(String[] args)
10    {
11       String[] colors = {"negro", "azul", "amarillo"};
12       LinkedList<String> enlaces = new LinkedList<>(Arrays.asList(colores));
13
14       enlaces.addLast("rojo"); // lo agrega como último elemento
15       enlaces.add("rosa"); // lo agrega al final
16       enlaces.add(3, "verde"); // lo agrega en el 3er índice
17       enlaces.addFirst("cyan"); // lo agrega como primer elemento
18
19       // obtiene los elementos de LinkedList como un arreglo
20       colores = enlaces.toArray(new String[enlaces.size()]);
21
22       System.out.println("colores: ");
23
24       for (String color : colores)
25          System.out.println(color);
26    }
27 } // fin de la clase UsoToArray
```

```
colores:
cyan
negro
azul
amarillo
verde
rojo
rosa
```

Fig. 16.4 | Ver arreglos como objetos `List` y convertir objetos `List` en arreglos.

En la línea 12 se construye un objeto `LinkedList` de objetos `String`, el cual contiene los elementos del arreglo `colores`. El método `asList` de `Arrays` devuelve una vista del arreglo como un objeto `List`, y después la usa para inicializar el objeto `LinkedList` con un constructor que recibe un objeto `Collection` como argumento (un objeto `List` *es un* objeto `Collection`). En la línea 14 se hace una llamada al **método `addLast` de `LinkedList`** para agregar "`rojo`" al final de `enlaces`. En las líneas 15 y 16 se hace una llamada al **método `add` de `LinkedList`** para agregar "`rosa`" como el último elemento y "`verde`" como el elemento

en el índice 3 (es decir, el cuarto elemento). El método `addLast` (línea 14) es idéntico en función al método `add` (línea 15). En la línea 17 se hace una llamada al **método `addFirst` de `LinkedList`** para agregar "`cyan`" como el nuevo primer elemento en el objeto `LinkedList`. Las operaciones `add` están permitidas debido a que operan en el objeto `LinkedList`, no en la vista devuelta por `asList`. [*Nota*: cuando se agrega "`cyan`" como el primer elemento, "`verde`" se convierte en el quinto elemento en el objeto `LinkedList`].

En la línea 20 se hace una llamada al método `toArray` de la interfaz `List` para obtener un arreglo `String` de `enlaces`. El arreglo es una copia de los elementos de la lista, por lo que si se modifica el contenido del arreglo *no* se modifica la lista. El arreglo que se pasa al método `toArray` debe ser del mismo tipo que se desee que devuelva el método `toArray`. Si el número de elementos en el arreglo es mayor o igual que el número de elementos en el objeto `LinkedList`, `toArray` copia los elementos de la lista en su argumento tipo arreglo y devuelve ese arreglo. Si el objeto `LinkedList` tiene más elementos que el número de elementos en el arreglo que se pasa a `toArray`, este método *asigna un nuevo arreglo* del mismo tipo que recibe como argumento, *copia* los elementos de la lista en el nuevo arreglo y devuelve este nuevo arreglo.

Error común de programación 16.2

Pasar un arreglo que contenga datos al método `toArray` puede crear errores lógicos. Si el número de elementos en el arreglo es menor que el número de elementos en la lista en la que se llama a `toArray`, se asigna un nuevo arreglo para almacenar los elementos de la lista (sin preservar los elementos del argumento tipo arreglo). Si el número de elementos en el arreglo es mayor que el número de elementos en la lista, los elementos del arreglo (empezando en el índice cero) se sobrescriben con los elementos de la lista. Los elementos de arreglos que no se sobrescriben retienen sus valores.

16.7 Métodos de las colecciones

La clase `Collections` cuenta con varios algoritmos de alto rendimiento para manipular los elementos de una colección. Los algoritmos (figura 16.5) se implementan como métodos `static`. Los métodos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` y `copy` operan con objetos `List`. Los métodos `min`, `max`, `addAll`, `frequency` y `disjoint` operan con objetos `Collections`.

Método	Descripción
`sort`	Ordena los elementos de un objeto `List`.
`binarySearch`	Localiza un objeto en un objeto `List` mediante el algoritmo de búsqueda binario de alto rendimiento que introdujimos en la sección 7.15 y que describimos con detalle en la sección 19.4.
`reverse`	Invierte los elementos de un objeto `List`.
`shuffle`	Ordena al azar los elementos de un objeto `List`.
`fill`	Establece cada elemento de un objeto `List` para que haga referencia a un objeto especificado.
`copy`	Copia referencias de un objeto `List` a otro.
`min`	Devuelve el elemento más pequeño en un objeto `Collection`.
`max`	Devuelve el elemento más grande en un objeto `Collection`.
`addAll`	Anexa todos los elementos en un arreglo a un objeto `Collection`.
`frequency`	Calcula cuántos elementos en la colección son iguales al elemento especificado.
`disjoint`	Determina si dos colecciones no tienen elementos en común.

Fig. 16.5 | Métodos de `Collections`.

Observación de ingeniería de software 16.5

Los métodos del marco de trabajo de colecciones son polimórficos. Es decir, cada algoritmo puede operar en objetos que implementen interfaces específicas, sin importar sus implementaciones subyacentes.

16.7.1 El método sort

El **método sort** ordena los elementos de un objeto `List`, el cual debe implementar a la **interfaz** `Comparable`. El orden se determina con base en el orden natural del tipo de los elementos, según su implementación mediante el método `compareTo` de ese objeto. Por ejemplo, el orden natural para los valores numéricos es el ascendente, y el orden natural para los objetos `String` se basa en su orden lexicográfico (sección 14.3). El método `compareTo` está declarado en la interfaz `Comparable` y algunas veces se le conoce como el **método de comparación natural**. La llamada a `sort` puede especificar como segundo argumento un objeto `Comparator`, para determinar un ordenamiento alterno de los elementos.

Ordenamiento ascendente

En la figura 16.6 se utiliza el método `sort` de `Collections` para ordenar los elementos de un objeto `List` en forma *ascendente* (línea 17). El método `sort` realiza un ordenamiento de combinación iterativo (en la sección 19.8 demostramos un ordenamiento por combinación recursivo). En la línea 14 se crea `lista` como un objeto `List` de objetos `String`. En cada una de las líneas 15 y 18 se utiliza una llamada *implícita* al método `toString` de `lista` para imprimir el contenido de la lista en el formato que se muestra en los resultados.

```
1  // Fig. 16.6: Ordenamiento1.java
2  // Método sort de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Ordenamiento1
8  {
9     public static void main(String[] args)
10    {
11       String[] palos = {"Corazones", "Diamantes", "Bastos", "Espadas"};
12
13       // Crea y muestra una list que contiene los elementos del arreglo palos
14       List<String> lista = Arrays.asList(palos);
15       System.out.printf("Elementos del arreglo desordenados: %s%n", lista);
16
17       Collections.sort(lista); // ordena ArrayList
18       System.out.printf("Elementos del arreglo ordenados: %s%n", lista);
19    }
20 } // fin de la clase Ordenamiento1
```

```
Elementos del arreglo desordenados: [Corazones, Diamantes, Bastos, Espadas]
Elementos del arreglo ordenados: [Bastos, Corazones, Diamantes, Espadas]
```

Fig. 16.6 | El método `sort` de `Collections`.

Ordenamiento descendente

En la figura 16.7 se ordena la misma lista de cadenas utilizadas en la figura 16.6, en orden *descendente*. El ejemplo introduce la interfaz `Comparator`, la cual se utiliza para ordenar los elementos de un objeto

`Collection` en un orden distinto. En la línea 18 se hace una llamada al método `sort` de `Collections` para ordenar el objeto `List` en orden descendente. El **método `static` `reverseOrder` de `Collections`** devuelve un objeto `Comparator` que ordena los elementos de la colección en forma inversa.

```
1  // Fig. 16.7: Ordenamiento2.java
2  // Uso de un objeto Comparator con el algoritmo sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Ordenamiento2
8  {
9     public static void main(String[] args)
10    {
11       String[] palos = {"Corazones", "Diamantes", "Bastos", "Espadas"};
12
13       // Crea y muestra una lista que contiene los elementos del arreglo palos
14       List<String> lista = Arrays.asList(palos); // crea objeto List
15       System.out.printf("Elementos del arreglo desordenados: %s%n", lista);
16
17       // ordena en forma descendente, utilizando un comparador
18       Collections.sort(lista, Collections.reverseOrder());
19       System.out.printf("Elementos de lista ordenados: %s%n", lista);
20    }
21 } // fin de la clase Ordenamiento2
```

```
Elementos del arreglo desordenados: [Corazones, Diamantes, Bastos, Espadas]
Elementos de lista ordenados: [Espadas, Diamantes, Corazones, Bastos]
```

Fig. 16.7 | El método `sort` de `Collections` con un objeto `Comparator`.

Ordenamiento mediante un objeto `Comparator`

En la figura 16.8 se crea una clase `Comparator` personalizada, llamada `ComparadorTiempo`, la cual implementa a la interfaz `Comparator` para comparar dos objetos `Tiempo2`. La clase `Tiempo2` declarada en la figura 8.5, representa tiempos con horas, minutos y segundos.

```
1  // Fig. 16.8: ComparadorTiempo.java
2  // Clase Comparator personalizada que compara dos objetos Tiempo2.
3  import java.util.Comparator;
4
5  public class ComparadorTiempo implements Comparator<Tiempo2>
6  {
7     @Override
8     public int compare(Tiempo2 tiempo1, Tiempo2 tiempo2)
9     {
10       int diferenciaHora = tiempo1.obtenerHora() - tiempo2.obtenerHora();
11
12       if (diferenciaHora != 0) // evalúa primero la hora
13          return diferenciaHora;
```

Fig. 16.8 | Clase `Comparator` personalizada que compara dos objetos `Tiempo2` (parte 1 de 2).

```
14
15          int diferenciaMinuto = tiempo1.obtenerMinuto() - tiempo2.obtenerMinuto();
16
17          if (diferenciaMinuto != 0) // después evalúa el minuto
18             return diferenciaMinuto;
19
20          int diferenciaSegundo = tiempo1.obtenerSegundo() - tiempo2.obtenerSegundo();
21          return diferenciaSegundo;
22       }
23    } // fin de la clase ComparadorTiempo
```

Fig. 16.8 | Clase `Comparator` personalizada que compara dos objetos `Tiempo2` (parte 2 de 2).

La clase `ComparadorTiempo` implementa a la interfaz `Comparator`, un tipo genérico que recibe un argumento (en este caso, `Tiempo2`). Una clase que implementa a `Comparator` debe declarar un método `compare` que reciba dos argumentos y devuelva un entero *negativo* si el primer argumento es *menor que* el segundo, 0 si los argumentos son *iguales* o un entero *positivo* si el primer argumento es *mayor que* el segundo. El método `compare` (líneas 7 a 22) realiza comparaciones entre objetos `Tiempo2`. En la línea 10 se comparan las dos horas de los objetos `Tiempo2`. Si las horas son distintas (línea 12), entonces devolvemos este valor. Si el valor es *positivo*, entonces la primera hora es mayor que la segunda y el primer tiempo es mayor que el segundo. Si este valor es *negativo*, entonces la primera hora es menor que la segunda y el primer tiempo es menor que el segundo. Si este valor es cero, las horas son iguales y debemos evaluar los minutos (y tal vez los segundos) para determinar cuál tiempo es mayor.

En la figura 16.9 se ordena una lista mediante el uso de la clase `Comparator` personalizada, llamada `ComparadorTiempo`. En la línea 11 se crea un objeto `ArrayList` de objetos `Tiempo2`. Recuerde que `ArrayList` y `List` son tipos genéricos y aceptan un argumento de tipo que especifica el tipo de los elementos de la colección. En las líneas 13 a 17 se crean cinco objetos `Tiempo2` y se agregan a esta lista. En la línea 23 se hace una llamada al método `sort`, y le pasamos un objeto de nuestra clase `ComparadorTiempo` (figura 16.8).

```
1  // Fig. 16.9: Ordenamiento3.java
2  // Método sort de Collections con un objeto Comparator personalizado.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Ordenamiento3
8  {
9     public static void main(String[] args)
10    {
11       List<Tiempo2> lista = new ArrayList<>(); // crea objeto List
12
13       lista.add(new Tiempo2(6, 24, 34));
14       lista.add(new Tiempo2(18, 14, 58));
15       lista.add(new Tiempo2(6, 05, 34));
16       lista.add(new Tiempo2(12, 14, 58));
17       lista.add(new Tiempo2(6, 24, 22));
18
```

Fig. 16.9 | El método `sort` de `Collections` con un objeto `Comparator` personalizado (parte 1 de 2).

```
19         // imprime los elementos del objeto List
20         System.out.printf("Elementos del arreglo desordenados:%n%s%n", lista);
21
22         // ordena usando un comparador
23         Collections.sort(lista, new ComparadorTiempo());
24
25         // imprime los elementos del objeto List
26         System.out.printf("Elementos de la lista ordenados:%n%s%n", lista);
27      }
28   } // fin de la clase Ordenamiento3
```

```
Elementos del arreglo desordenados:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Elementos de la lista ordenados:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Fig. 16.9 | El método `sort` de `Collections` con un objeto `Comparator` personalizado (parte 2 de 2).

16.7.2 El método `shuffle`

El método **`shuffle`** ordena al azar los elementos de un objeto `List`. En el capítulo 7 presentamos una simulación para barajar y repartir cartas, en la que se utiliza un ciclo para barajar un mazo de cartas. En la figura 16.10 utilizamos el algoritmo `shuffle` para barajar un mazo de objetos `Carta` que podría usarse en un simulador de juego de cartas.

```
1  // Fig. 16.10: MazoDeCartas.java
2  // Barajar y repartir cartas con el método shuffle de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  // clase para representar un objeto Carta en un mazo de cartas
8  class Carta
9  {
10    public static enum Cara {As, Dos, Tres, Cuatro, Cinco, Seis,
11       Siete, Ocho, Nueve, Diez, Joker, Reina, Rey };
12    public static enum Palo {Bastos, Diamantes, Corazones, Espadas};
13
14    private final Cara cara;
15    private final Palo palo;
16
17    // constructor
18    public Carta(Cara cara, Palo palo)
19    {
20       this.cara = cara;
21       this.palo = palo;
22    }
23
24    // devuelve la cara de la carta
25    public Cara obtenerCara()
26    {
```

Fig. 16.10 | Barajar y repartir cartas con el método `shuffle` de `Collections` (parte 1 de 3).

```
27          return cara;
28       }
29
30       // devuelve el palo de la Carta
31       public Palo obtenerPalo()
32       {
33          return palo;
34       }
35
36       // devuelve la representación String de la Carta
37       public String toString()
38       {
39          return String.format("%s of %s", cara, palo);
40       }
41    } // fin de la clase Carta
42
43    // declaración de la clase MazoDeCartas
44    public class MazoDeCartas
45    {
46       private List<Carta> lista; // declara objeto List que almacenará los objetos Carta
47
48       // establece mazo de objetos Carta y baraja
49       public MazoDeCartas()
50       {
51          Carta[] mazo = new Carta[52];
52          int cuenta = 0; // número de cartas
53
54          // llena el mazo con objetos Carta
55          for (Carta.Palo palo: Carta.Palo.values())
56          {
57             for (Carta.Cara cara: Carta.Cara.values())
58             {
59                mazo[cuenta] = new Carta(cara, palo);
60                ++cuenta;
61             }
62          }
63
64          lista = Arrays.asList(mazo); // obtiene objeto List
65          Collections.shuffle(lista);  // baraja el mazo
66       } // fin del constructor de MazoDeCartas
67
68       // imprime el mazo
69       public void imprimirCartas()
70       {
71          // muestra las 52 cartas en dos columnas
72          for (int i = 0; i < lista.size(); i++)
73             System.out.printf("%-19s%s", lista.get(i),
74                ((i + 1) % 4 == 0) ? "%n" : "");
75       }
76
77       public static void main(String[] args)
78       {
```

Fig. 16.10 | Barajar y repartir cartas con el método `shuffle` de `Collections` (parte 2 de 3).

```
79          MazoDeCartas cartas = new MazoDeCartas();
80          cartas.imprimirCartas();
81       }
82    } // fin de la clase MazoDeCartas
```

```
Rey de Diamantes      Diez de Diamantes     Cinco de Espadas      Dos de Bastos
Dos de Diamantes      Dos de Corazones      Reina de Espadas      Tres de Diamantes
Siete de Espadas      Dos de Espadas        Siete de Diamantes    Cuatro de Espadas
Tres de Bastos        As de Diamantes       Cuatro de Diamantes   As de Bastos
Siete de Bastos       Ocho de Bastos        Reina de Corazones    As de Espadas
As de Corazones       Tres de Espadas       Cinco de Bastos       Ocho de Corazones
Cuatro de Corazones   Ocho de Diamantes     Nueve de Bastos       Ocho de Espadas
Joker de Bastos       Diez de Bastos        Siete de Corazones    Tres de Corazones
Rey de Espadas        Nueve de Corazones    Cinco de Corazones    Joker de Diamantes
Cuatro de Bastos      Seis de Bastos        Nueve de Diamantes    Reina de Diamantes
Rey de Corazones      Joker de Espadas      Diez de Espadas       Seis de Corazones
Rey de Bastos         Nueve de Espadas      Diez de Corazones     Seis de Diamantes
Seis de Espadas       Cinco de Diamantes    Joker de Corazones    Reina de Bastos
```

Fig. 16.10 | Barajar y repartir cartas con el método `shuffle` de `Collections` (parte 3 de 3).

La clase `Carta` (líneas 8 a 41) representa a una carta en un mazo de cartas. Cada `Carta` tiene una cara y un palo. Las líneas 10 a 12 declaran dos tipos `enum` (`Cara` y `Palo`) que representan la cara y el palo de la carta, respectivamente. El método `toString` (líneas 37 a 40) devuelve un objeto `String` que contiene la cara y el palo de la `Carta`, separados por la cadena " `de` ". Cuando una constante `enum` se convierte en una cadena, el identificador de la constante se utiliza como la representación de `String`. Por lo general, utilizamos letras mayúsculas para las constantes `enum`. En este ejemplo, optamos por usar letras mayúsculas sólo para la primera letra de cada constante `enum`, porque queremos que la carta se muestre con letras iniciales mayúsculas para la cara y el palo (por ejemplo, "`As de Bastos`").

En las líneas 55 a 62 se llena el arreglo `mazo` con cartas que tienen combinaciones únicas de cara y palo. Tanto `Cara` como `Palo` son tipos `public static enum` de la clase `Carta`. Para usar estos tipos `enum` fuera de la clase `Carta`, debe calificar el nombre de cada tipo `enum` con el nombre de la clase en la que reside (es decir, `Carta`) y un separador punto (`.`). Así, en las líneas 55 y 57 se utilizan `Carta.Palo` y `Carta.Cara` para declarar las variables de control de las instrucciones `for`. Recuerde que el método `values` de un tipo `enum` devuelve un arreglo que contiene todas las constantes del tipo `enum`. En las líneas 55 a 62 se utilizan instrucciones `for` mejoradas para construir 52 nuevos objetos `Carta`.

La acción de barajar las cartas ocurre en la línea 65, en la cual se hace una llamada al método `static` `shuffle` de la clase `Collections` para barajar los elementos del arreglo. El método `shuffle` requiere un argumento `List`, por lo que debemos obtener una vista `List` del arreglo antes de poder barajarlo. En la línea 64 se invoca el método `static asList` de la clase `Arrays` para obtener una vista `List` del arreglo `mazo`.

El método `imprimirCartas` (líneas 69 a 75) muestra el mazo de cartas en dos columnas. En cada iteración del ciclo (las líneas 73 y 74) se imprime una carta justificada a la izquierda, en un campo de 19 caracteres seguido de una nueva línea o de una cadena vacía, con base en el número de cartas mostradas hasta ese momento. Si el número de cartas es un múltiplo de 4, se imprime una nueva línea; en caso contrario, se imprime un tabulador.

16.7.3 Los métodos `reverse`, `fill`, `copy`, `max` y `min`

La clase `Collections` proporciona algoritmos para *invertir*, *llenar* y *copiar* objetos `List`. El **método `reverse` de `Collections`** invierte el orden de los elementos en un objeto `List` y el **método `fill`** *sobrescribe* los elementos en un objeto `List` con un valor especificado. La operación `fill` es útil para reinicia-

lizar un objeto `List`. El **método copy** recibe dos argumentos: un objeto `List` de destino y un objeto `List` de origen. Cada elemento del objeto `List` de origen se copia al objeto `List` de destino. El objeto `List` de destino debe tener cuando menos la misma longitud que el objeto `List` de origen; de lo contrario, se producirá una excepción `IndexOutOfBoundsException`. Si el objeto `List` de destino es más largo, los elementos que no se sobrescriban permanecerán sin cambio.

Cada uno de los métodos que hemos visto hasta ahora opera en objetos `List`. Los métodos **`min`** y **`max`** operan en cualquier objeto `Collection`. El método `min` devuelve el elemento más pequeño en un objeto `Collection` y el método `max` devuelve el elemento más grande en un objeto `Collection`. Ambos métodos pueden llamarse con un objeto `Comparator` como segundo argumento para realizar *comparaciones personalizadas* entre objetos, como el objeto `ComparadorTiempo` en la figura 16.9. En la figura 16.11 se demuestra el uso de los métodos `reverse`, `fill`, `copy`, `max` y `min`.

```
1  // Fig. 16.11: Algoritmos1.java
2  // Los métodos reverse, fill, copy, max y min de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Algoritmos1
8  {
9     public static void main(String[] args)
10    {
11       // crea y muestra un objeto List<Character>
12       Character[] letras = {'P', 'C', 'M'};
13       List<Character> lista = Arrays.asList(letras); // obtiene el objeto List
14       System.out.println("lista contiene: ");
15       imprimir(lista);
16
17       // invierte y muestra el objeto List<Character>
18       Collections.reverse(lista); // invierte el orden de los elementos
19       System.out.printf("%nDespues de llamar a reverse, lista contiene:%n");
20       imprimir(lista);
21
22       // crea copiaLista a partir de un arreglo de 3 objetos Character
23       Character[] letrasCopia = new Character[3];
24       List<Character> copiaLista = Arrays.asList(letrasCopia);
25
26       // copia el contenido de lista a copiaLista
27       Collections.copy(copiaLista, lista);
28       System.out.printf("%nDespues de copiar, copiaLista contiene:%n");
29       imprimir(copiaLista);
30
31       // llena la lista con letras R
32       Collections.fill(lista, 'R');
33       System.out.printf("%nDespues de llamar a fill, lista contiene:%n");
34       imprimir(lista);
35    }
36
37    // imprime la información del objeto List
38    private static void imprimir(List<Character> refLista)
39    {
```

Fig. 16.11 | Los métodos `reverse`, `fill`, `copy`, `max` y `min` de `Collections` (parte 1 de 2).

```
40         System.out.print("La lista es: ");
41
42         for (Character elemento : refLista)
43            System.out.printf("%s ", elemento);
44
45         System.out.printf("%nMax: %s", Collections.max(refLista));
46         System.out.printf("  Min: %s%n", Collections.min(refLista));
47      }
48   } // fin de la clase Algoritmos1
```

```
Lista inicial:
La lista es: P C M
Max: P  Min: C

Despues de llamar a reverse:
La lista es: M C P
Max: P  Min: C

Después de copy:
La lista es: M C P
Max: P  Min: C

Después de llamar a fill:
La lista es: R R R
Max: R  Min: R
```

Fig. 16.11 | Los métodos `reverse`, `fill`, `copy`, `max` y `min` de `Collections` (parte 2 de 2).

En la línea 13 se crea la variable `lista` de tipo `List<Character>` y se inicializa con una vista `List` del arreglo letras tipo `Character`. En las líneas 14 y 15 se imprime en pantalla el contenido actual del objeto `List`. En la línea 18 se hace una llamada al método `reverse` de `Collections` para invertir el orden de `lista`. El método `reverse` recibe un argumento `List`. Como `lista` es una vista `List` del arreglo `letras`, los elementos del arreglo están ahora en orden inverso. El contenido inverso se imprime en pantalla en las líneas 19 y 20. En la línea 27 se copian los elementos de `lista` en `copiaLista`, usando el método `copy` de `Collections`. Los cambios a `copiaLista` no cambian a `letras`, ya que `copiaLista` es un objeto `List` independiente que no es una vista `List` del arreglo `letras`. El método `copy` requiere dos argumentos `List`: el objeto `List` de destino y el de origen. En la línea 32 se hace una llamada al método `fill` de `Collections` para colocar el caracter 'R' en cada elemento de `lista`. Como `lista` es una vista `List` del arreglo `letras`, esta operación cambia cada elemento en `letras` a 'R'. El método `fill` requiere un objeto `List` como primer argumento, y un objeto `Object` como segundo argumento. En este caso, el objeto `Object` es la versión *embalada* del carácter 'R'. En las líneas 45 y 46 se hace una llamada a los métodos `max` y `min` de `Collections` para buscar el elemento más grande y más pequeño de la colección, respectivamente. Recuerde que la interfaz `List` extiende a la interfaz `Collection`, por lo que un objeto `List` *es un* objeto `Collection`.

16.7.4 El método `binarySearch`

El algoritmo de búsqueda binaria de alta velocidad (que veremos con detalle en la sección 19.4) está integrado al marco de trabajo de colecciones de Java como el **método `static` `binarySearch` de la clase `Collections`**. Este método localiza un objeto en un objeto `List` (es decir, un objeto `LinkedList` o `ArrayList`). Si se encuentra el objeto, se devuelve el índice de ese objeto. Si no se encuentra el objeto, `binarySearch` devuelve un valor negativo. El método `binarySearch` determina este valor negativo

calculando primero el punto de inserción y cambiando el signo del punto de inserción a negativo. Después, `binarySearch` resta 1 al punto de inserción para obtener el valor de retorno, el cual garantiza que el método `binarySearch` devolverá números positivos (>= 0), sí y sólo si se encuentra el objeto. Si varios elementos en la lista coinciden con la clave de búsqueda, no hay garantía de que uno se localice primero. En la figura 16.12 se utiliza el método `binarySearch` para buscar una serie de cadenas en un objeto `ArrayList`.

```
1  // Fig. 16.12: PruebaBusquedaBinaria.java
2  // El método binarySearch de Collections.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6  import java.util.ArrayList;
7
8  public class PruebaBusquedaBinaria
9  {
10    public static void main(String[] args)
11    {
12       // crea un ArrayList<String> a partir del contenido del arreglo colores
13       String[] colores = {"rojo", "blanco", "azul", "negro", "amarillo",
14          "purple", "tan", "pink"};
15       List<String> lista =
16          new ArrayList<>(Arrays.asList(colores));
17
18       Collections.sort(lista); // ordena el objeto ArrayList
19       System.out.printf("ArrayList ordenado: %s%n", lista);
20
21       // busca varios valores en la lista
22       imprimirResultadosBusqueda(lista, "negro"); // primer elemento
23       imprimirResultadosBusqueda(lista, "rojo"); // elemento medio
24       imprimirResultadosBusqueda(lista, "rosa"); // último elemento
25       imprimirResultadosBusqueda(lista, "aqua"); // debajo del menor
26       imprimirResultadosBusqueda(lista, "gris"); // no existe
27       imprimirResultadosBusqueda(lista, "verdeazulado"); // no existe
28    }
29
30    // realiza búsqueda y muestra el resultado
31    private static void imprimirResultadosBusqueda(
32       List<String> lista, String clave)
33    {
34       int resultado = 0;
35
36       System.out.printf("%nBuscando: %s%n", clave);
37       resultado = Collections.binarySearch(lista, clave);
38
39       if (resultado >= 0)
40          System.out.printf("Se encontro en el indice %d%n", resultado);
41       else
42          System.out.printf("No se encontro (%d)%n",resultado);
43    }
44 } // fin de la clase PruebaBusquedaBinaria
```

Fig. 16.12 | El método `binarySearch` de `Collections` (parte 1 de 2).

```
ArrayList ordenado: [amarillo, azul, blanco, carne, morado, negro, rojo, rosa]

Buscando: negro
Se encontro en el indice 5

Buscando: rojo
Se encontro en el indice 6

Buscando: rosa
Se encontro en el indice 7

Buscando: aqua
No se encontro (-2)

Buscando: gris
No se encontro (-5)

Buscando: verdeazulado
No se encontro (-9)
```

Fig. 16.12 | El método `binarySearch` de `Collections` (parte 2 de 2).

En las líneas 15 y 16 se inicializa `lista` con un `ArrayList` que contiene una copia de los elementos en el arreglo `colores`. El método `binarySearch` de `Collections` espera que los elementos del argumento `List` estén en orden *ascendente*, por lo que la línea 18 se usa el método `sort` de `Collections` para ordenar la lista. Si los elementos del argumento `List` *no* están ordenados, el resultado de usar `binarySearch` es *indefinido*. En la línea 19 se imprime la lista ordenada en la pantalla. En las líneas 22 al 27 se hacen llamadas al método `imprimirResultadosBusqueda` (líneas 31 a 43) para realizar la búsqueda e imprimir los resultados en pantalla. En la línea 37 se hace una llamada al método `binarySearch` de `Collections` para buscar en `lista` la `clave` especificada. El método `binarySearch` recibe un objeto `List` como primer argumento, y un objeto `Object` como segundo argumento. En las líneas 39 a 42 se imprimen en pantalla los resultados de la búsqueda. Una versión sobrecargada de `binarySearch` recibe un objeto `Comparator` como tercer argumento, el cual especifica la forma en que `binarySearch` debe comparar la clave de búsqueda con los elementos del objeto `List`.

16.7.5 Los métodos `addAll`, `frequency` y `disjoint`

La clase `Collections` también proporciona los métodos `addAll`, `frequency` y `disjoint`. El **método `addAll` de `Collections`** recibe dos argumentos: un objeto `Collection` en el que se van a *insertar* los nuevos elementos y un arreglo que proporciona los elementos a insertar. El **método `frequency` de `Collections`** recibe dos argumentos: un objeto `Collection` en el que se va a buscar y un objeto `Object` que se va a buscar en la colección. El método `frequency` devuelve el número de veces que aparece el segundo argumento en la colección. El **método `disjoint` de `Collections`** recibe dos objetos `Collections` y devuelve `true` si *no tienen elementos en común*. En la figura 16.13 se demuestra el uso de los métodos `addAll`, `frequency` y `disjoint`.

```
1  // Fig. 16.13: Algoritmos2.java
2  // Los métodos addAll, frequency y disjoint de Collections.
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Arrays;
6  import java.util.Collections;
```

Fig. 16.13 | Los métodos `addAll`, `frequency` y `disjoint` de `Collections` (parte 1 de 2).

```
 7
 8  public class Algoritmos2
 9  {
10     public static void main(String[] args)
11     {
12        // inicializa lista1 y lista2
13        String[] colores = {"rojo", "blanco", "amarillo", "azul"};
14        List<String> lista1 = Arrays.asList(colores);
15        ArrayList<String> lista2 = new ArrayList<>();
16
17        lista2.add("negro"); // agrega "negro" al final de lista2
18        lista2.add("rojo"); // agrega "rojo" al final de lista2
19        lista2.add("verde"); // agrega "verde" al final de lista2
20
21        System.out.print("Antes de addAll, lista2 contiene: ");
22
23        // muestra los elementos en lista2
24        for (String s : lista2)
25           System.out.printf("%s ", s);
26
27        Collections.addAll(lista2, colores); // agrega los objetos String de colores
                                               a lista2
28
29        System.out.printf("%nDespues de addAll, lista2 contiene: ");
30
31        // muestra los elementos en lista2
32        for (String s : lista2)
33           System.out.printf("%s ", s);
34
35        // obtiene la frecuencia de "rojo"
36        int frecuencia = Collections.frequency(lista2, "rojo");
37        System.out.printf(
38           "%nFrecuencia de rojo en lista2: %d%n", frecuencia);
39
40        // comprueba si lista1 y lista2 tienen elementos en común
41        boolean desunion = Collections.disjoint(lista1, lista2);
42
43        System.out.printf("lista1 y lista2 %s elementos en comun%n",
44           (desunion ? "no tienen" : "tienen"));
45     }
46  } // fin de la clase Algoritmos2
```

```
Antes de addAll, lista2 contiene: negro rojo verde
Despues de addAll, lista2 contiene: negro rojo verde rojo blanco amarillo azul
Frecuencia de rojo en lista2: 2
lista1 y lista2 tienen elementos en comun
```

Fig. 16.13 | Los métodos `addAll`, `frequency` y `disjoint` de `Collections` (parte 2 de 2).

En la línea 14 se inicializa `lista1` con los elementos en el arreglo `colores`, y en las líneas 17 a 19 se agregan los objetos `String` "`negro`", "`rojo`" y "`verde`" a `lista2`. En la línea 27 se invoca el método `addAll` para agregar los elementos en el arreglo `colores` a `lista2`. En la línea 36 se obtiene la frecuencia del objeto `String` "`rojo`" en `lista2`, usando el método `frequency`. En la línea 41 se invoca el método `disjoint` para evaluar si los objetos `Collections lista1` y `lista2` tienen elementos en común, lo cual es cierto en este ejemplo.

16.8 La clase Stack del paquete java.util

En la sección 6.6 presentamos el concepto de una *pila* al hablar sobre la pila de llamadas a métodos. En el capítulo 21 (en inglés, en el sitio web del libro) aprenderemos a construir estructuras de datos fundamentales, incluyendo *listas enlazadas*, *pilas*, *colas* y *árboles*. En un mundo de reutilización de software, en vez de construir las estructuras de datos conforme las necesitamos, podemos a menudo aprovechar las estructuras de datos existentes. En esta sección, investigaremos la clase **Stack** en el paquete de utilerías de Java (`java.util`).

La clase `Stack` extiende a la clase `Vector` para implementar una estructura de datos tipo pila. En la figura 16.14 se demuestran varios métodos de `Stack`. Para obtener los detalles de la clase `Stack`, visite el sitio Web `docs.oracle.com/javase/7/docs/api/java/util/Stack.html`.

```
1  // Fig. 16.14: PruebaStack.java
2  // La clase Stack del paquete java.util.
3  import java.util.Stack;
4  import java.util.EmptyStackException;
5
6  public class PruebaStack
7  {
8     public static void main(String[] args)
9     {
10       Stack<Number> pila = new Stack<>(); // crea un objeto Stack
11
12       // usa el método push
13       pila.push(12L); // mete el valor long 12L
14       System.out.println("Se metio 12L");
15       imprimirPila(pila);
16       pila.push(34567); // mete el valor int 34567
17       System.out.println("Se metio 34567");
18       imprimirPila(pila);
19       pila.push(1.0F); // mete el valor float 1.0F
20       System.out.println("Se metio 1.0F");
21       imprimirPila(pila);
22       pila.push(1234.5678); // mete el valor double 1234.5678
23       System.out.println("Se metio 1234.5678 ");
24       imprimirPila(pila);
25
26       // elimina los elementos de la pila
27       try
28       {
29          Number objetoEliminado = null;
30
31          // saca elementos de la pila
32          while (true)
33          {
34             objetoEliminado = pila.pop(); // usa el método pop
35             System.out.printf("Se saco %s%n", objetoEliminado);
36             imprimirPila(pila);
37          }
38       }
39       catch (EmptyStackException emptyStackException)
40       {
```

Fig. 16.14 | La clase `Stack` del paquete `java.util` (parte 1 de 2).

```
41            emptyStackException.printStackTrace();
42         }
43      }
44
45      // muestra el contenido de Pila
46      private static void imprimirPila(Stack<Number> pila)
47      {
48         if (pila.isEmpty())
49            System.out.printf("la pila esta vacia%n%n"); // la pila está vacía
50         else // la pila no está vacía
51            System.out.printf("la pila contiene: %s (cima)%n", pila);
52      }
53   } // fin de la clase PruebaStack
```

```
Se metio 12L
la pila contiene: [12] (cima)
Se metio 34567
la pila contiene: [12, 34567] (cima)
Se metio 1.0F
la pila contiene: [12, 34567, 1.0] (cima)
Se metio 1234.5678
la pila contiene: [12, 34567, 1.0, 1234.5678] (cima)
Se saco 1234.5678
la pila contiene: [12, 34567, 1.0] (cima)
Se saco 1.0
la pila contiene: [12, 34567] (cima)
Se saco 34567
la pila contiene: [12] (cima)
Se saco 12
La pila esta vacia

java.util.EmptyStackException
        at java.util.Stack.peek(Unknown Source)
        at java.util.Stack.pop(Unknown Source)
        at PruebaStack.main(PruebaStack.java:34)
```

Fig. 16.14 | La clase `Stack` del paquete `java.util` (parte 2 de 2).

Tip para prevenir errores 16.1

Como `Stack` extiende a `Vector`, todos los métodos `public` de `Vector` pueden llamarse en objetos `Stack`, aun si los métodos no representan operaciones de pila convencionales. Por ejemplo, el método `add` de `Vector` se puede utilizar para insertar un elemento en cualquier parte de una pila; una operación que podría "corromper" los datos de la pila. Al manipular un objeto `Stack`, sólo deben usarse los métodos `push` y `pop` para agregar y eliminar elementos de la pila, respectivamente. En la sección 21.5 creamos una clase `Stack` que usa la composición, de modo que `Stack` proporcione en su interfaz `public` sólo las herramientas que deban permitirse en una `Pila`.

En la línea 10 del constructor se crea un objeto `Stack` vacío de tipo `Number`. La clase `Number` (en el paquete `java.lang`) es la superclase de la mayoría de las clases de envoltura (como `Integer`, `Double`) para los tipos primitivos. Al crear un objeto `Stack` de objetos `Number`, se pueden meter en la pila objetos de cualquier clase que extienda a la clase `Number`. En cada una de las líneas 13, 16, 19 y 22 se hace una llamada al método **`push`** de `Stack` para agregar objetos `Number` a la *cima* de la pila. Observe las literales `12L` (línea 13) y `1.0F` (línea 19). Cualquier literal entera que tenga el **sufijo `L`** es un valor `long`. Cualquier literal

entera sin un sufijo es un valor `int`. De manera similar, cualquier literal de punto flotante que tenga el **sufijo `F`** es un valor `float`. Una literal de punto flotante sin un sufijo es un valor `double`. Puede aprender más acerca de las literales numéricas en la *Especificación del lenguaje Java*, en el sitio web `http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.8.1`.

Un ciclo infinito (líneas 32 a 37) llama al **método `pop` de `Stack`** para eliminar el elemento *superior* de la pila. El método devuelve una referencia `Number` al elemento eliminado. Si no hay elementos en el objeto `Stack`, el método `pop` lanza una excepción **`EmptyStackException`**, la cual termina el ciclo. La clase `Stack` también declara el **método `peek`**. Este método devuelve el elemento *superior* de la pila sin sacarlo.

El método `imprimirPila` (líneas 46 a 52) muestra el contenido de la pila. La *cima* actual de la pila (el último valor que se metió a ésta) es el *primer* valor que se imprime. En la línea 48 se hace una llamada al **método `isEmpty` de `Stack`** (heredado por `Stack` de la clase `Vector`) para determinar si la pila está vacía. Si está vacía, el método devuelve `true`; en caso contrario, devuelve `false`.

16.9 La clase `PriorityQueue` y la interfaz `Queue`

Recordemos que una cola es una colección que representa una fila de espera; por lo general, las *inserciones* se realizan en la parte posterior de una cola y las *eliminaciones* se realizan en la parte delantera. En la sección 21.6 veremos la estructura de datos tipo cola y crearemos nuestra propia implementación de ella. En esta sección investigaremos la interfaz **`Queue`** y la clase **`PriorityQueue`** del paquete `java.util`. La interfaz `Queue` extiende a la interfaz `Collection` y proporciona operaciones adicionales para *insertar*, *eliminar* e *inspeccionar* elementos en una cola. `PriorityQueue`, que implementa a la interfaz `Queue`, ordena los elementos con base en su orden natural, según lo especificado por el método `compareTo` de los elementos `Comparable`, o mediante un objeto `Comparator` que se suministra a través del constructor.

La clase `PriorityQueue` proporciona una funcionalidad que permite *inserciones en orden* en la estructura de datos subyacente, y *eliminaciones* de la parte *frontal* de la estructura de datos subyacente. Al agregar elementos a un objeto `PriorityQueue`, los elementos se insertan en orden de prioridad, de tal forma que el *elemento con mayor prioridad* (es decir, el valor más grande) será el primer elemento eliminado del objeto `PriorityQueue`.

Las operaciones comunes de `PriorityQueue` son: **`offer`** para *insertar* un elemento en la ubicación apropiada, con base en el orden de prioridad; **`poll`** para *eliminar* el elemento de mayor prioridad de la cola de prioridad (es decir, la parte inicial o cabeza de la cola); **`peek`** para obtener una referencia al elemento de mayor prioridad de la cola de prioridad (sin eliminar ese elemento); **`clear`** para *eliminar todos los elementos* en la cola de prioridad, y **`size`** para obtener el número de elementos en la cola de prioridad.

En la figura 16.15 se demuestra la clase `PriorityQueue`. En la línea 10 se crea un objeto `PriorityQueue` que almacena objetos `Double` con una *capacidad inicial* de 11 elementos, y se ordenan los elementos de acuerdo con el ordenamiento natural del objeto (los valores predeterminados para un objeto `PriorityQueue`). `PriorityQueue` es una clase genérica. En la línea 10 se crea una instancia de un objeto `PriorityQueue` con un argumento de tipo `Double`. La clase `PriorityQueue` proporciona cinco constructores adicionales. Uno de éstos recibe un `int` y un objeto `Comparator` para crear un objeto `PriorityQueue` con la *capacidad inicial* especificada por el valor `int` y el *ordenamiento* por el objeto `Comparator`. En las líneas 13 a 15 se utiliza el método `offer` para agregar elementos a la cola de prioridad. El método `offer` lanza una excepción `NullPointerException` si el programa trata de agregar un objeto `null` a la cola. El ciclo en las líneas 20 a 24 utiliza el método `size` para determinar si la cola de prioridad está *vacía* (línea 20). Mientras haya más elementos, en la línea 22 se utiliza el método `peek` de `PriorityQueue` para obtener el *elemento de mayor prioridad* en la cola, para imprimirlo en pantalla (*sin* eliminarlo de la cola). En la línea 23 se elimina el elemento de mayor prioridad en la cola, con el método `poll`, que devuelve el elemento eliminado.

```
1  // Fig. 16.15: PruebaPriorityQueue.java
2  // Programa de prueba de la clase PriorityQueue.
3  import java.util.PriorityQueue;
4
5  public class PruebaPriorityQueue
6  {
7     public static void main(String[] args)
8     {
9        // cola con capacidad de 11
10       PriorityQueue<Double> cola = new PriorityQueue<>();
11
12       // inserta elementos en la cola
13       cola.offer(3.2);
14       cola.offer(9.8);
15       cola.offer(5.4);
16
17       System.out.print("Sondeando de cola: ");
18
19       // muestra los elementos en la cola
20       while (cola.size() > 0)
21       {
22          System.out.printf("%.1f ", cola.peek()); // ve el elemento superior
23          cola.poll(); // elimina el elemento superior
24       }
25    }
26 } // fin de la clase PruebaPriorityQueue
```

```
Sondeando de cola: 3.2 5.4 9.8
```

Fig. 16.15 | Programa de prueba de la clase `PriorityQueue`.

16.10 Conjuntos

Un objeto **`Set`** es un objeto `Collection` *desordenado* que contiene elementos únicos (es decir, sin *elementos duplicados*). El marco de trabajo de colecciones contiene varias implementaciones de `Set`, incluyendo a **`HashSet`** y **`TreeSet`**. `HashSet` almacena sus elementos en una *tabla de hash*, y `TreeSet` almacena sus elementos en un árbol. El concepto de las tablas de hash se presenta en la sección 16.11. Hablaremos sobre los árboles en la sección 21.7.

En la figura 16.16 se utiliza un objeto `HashSet` para *eliminar las cadenas duplicadas* de un objeto `List`. Recuerde que tanto `List` como `Collection` son tipos genéricos, por lo que en la línea 16 se crea un objeto `List` que contiene objetos `String`, y en la línea 20 se pasa un objeto `Collection` de objetos `String` al método `imprimirSinDuplicados`. El método `imprimirSinDuplicados` (líneas 24 a 35) recibe un argumento `Collection`. En la línea 27 se crea un objeto `HashSet<String>` a partir del argumento `Collection<String>`. Por definición, los objetos `Set` *no* contienen valores duplicados, por lo que cuando se construye el objeto `HashSet`, éste *elimina cualquier valor duplicado* en el objeto `Collection`. En las líneas 31 y 32 se imprimen en pantalla los elementos en el objeto `Set`.

```
1  // Fig. 16.16: PruebaSet.java
2  // Uso de un objeto HashSet para eliminar valores duplicados de un arreglo de cadenas.
3  import java.util.List;
4  import java.util.Arrays;
```

Fig. 16.16 | Uso de un objeto `HashSet` para eliminar valores duplicados de un arreglo de cadenas (parte 1 de 2).

```
5  import java.util.HashSet;
6  import java.util.Set;
7  import java.util.Collection;
8
9  public class PruebaSet
10 {
11    public static void main(String[] args)
12    {
13       // crea y muestra un objeto List<String>
14       String[] colores = {"rojo", "blanco", "azul", "verde", "gris",
15          "naranja", "carne", "blanco", "cyan", "durazno", "gris", "naranja"};
16       List<String> lista = Arrays.asList(colores);
17       System.out.printf("List: %s%n", lista);
18
19       // elimina duplicados y luego imprime los valores únicos
20       imprimirSinDuplicados(lista);
21    }
22
23    // crea un objeto Set a partir de un objeto Collection para eliminar duplicados
24    private static void imprimirSinDuplicados(Collection<String> valores)
25    {
26       // crea un objeto HashSet
27       Set<String> conjunto = new HashSet<>(valores);
28
29       System.out.printf("%nLos valores sin duplicados son: ");
30
31       for (String valor : conjunto)
32          System.out.printf("%s ", valor);
33
34       System.out.println();
35    }
36 } // fin de la clase PruebaSet
```

```
List: [rojo, blanco, azul, verde, gris, naranja, carne, blanco, cyan, durazno, gris,
naranja]
Los valores sin duplicados son: durazno gris verde azul blanco rojo cyan carne naranja
```

Fig. 16.16 | Uso de un objeto `HashSet` para eliminar valores duplicados de un arreglo de cadenas (parte 2 de 2).

Conjuntos ordenados

El marco de trabajo de colecciones también incluye la **interfaz `SortedSet`** (que extiende a `Set`) para los conjuntos que mantengan a sus elementos *ordenados*; ya sea en el *orden natural de los elementos* (por ejemplo, los números se encuentran en orden *ascendente*) o en un orden especificado por un objeto `Comparator`. La clase `TreeSet` implementa a `SortedSet`. El programa de la figura 16.17 coloca objetos `String` en un objeto `TreeSet`. Estos objetos `String` se ordenan al ser agregadas al objeto `TreeSet`. Este ejemplo también demuestra los métodos de *vista de rango*, los cuales permiten a un programa ver una porción de una colección.

En la línea 14 se crea un objeto `TreeSet<String>` que contiene los elementos del arreglo `colores`, y luego se asigna el nuevo objeto `TreeSet<String>` a la variable `arbol Sorted<String>`. En la línea 17 se imprime en pantalla el conjunto inicial de cadenas, utilizando el método `imprimirConjunto` (líneas 33 a 39), sobre el cual hablaremos en breve. En la línea 31 se hace una llamada al **método `headSet` de `TreeSet`** para obtener un subconjunto del objeto `TreeSet`, en el que todos los elementos serán menores

que "naranja". La vista devuelta de headSet se imprime a continuación con imprimirConjunto. Si se hace algún cambio al subconjunto, éste se reflejará *también* en el objeto TreeSet original, debido a que el subconjunto devuelto por headSet es una vista del objeto TreeSet.

En la línea 25 se hace una llamada al **método tailSet de TreeSet** para obtener un subconjunto en el que cada elemento sea mayor o igual que "naranja", y después se imprime el resultado en pantalla. Cualquier cambio realizado a través de la vista tailSet se realiza también en el objeto TreeSet original. En las líneas 28 y 29 se hace una llamada a los **métodos first** y **last** de **SortedSet** para obtener el elemento más pequeño y más grande del conjunto, respectivamente.

El método imprimirConjunto (líneas 33 a 39) recibe un objeto SortedSet como argumento y lo imprime. En las líneas 35 y 36 se imprime en pantalla cada elemento del objeto SortedSet, usando la instrucción for mejorada.

```
1  // Fig. 16.17: PruebaSortedSet.java
2  // Uso de SortedSet y TreeSet.
3  import java.util.Arrays;
4  import java.util.SortedSet;
5  import java.util.TreeSet;
6
7  public class PruebaSortedSet
8  {
9     public static void main(String[] args)
10    {
11       // crea un TreeSet del arreglo colores
12       String[] colores = {"amarillo", "verde", "negro", "carne", "gris",
13          "blanco", "naranja", "rojo", "verde"};
14       SortedSet<String> arbol = new TreeSet<>(Arrays.asList(colores));
15
16       System.out.print("conjunto ordenado: ");
17       imprimirConjunto(arbol);
18
19       // obtiene subconjunto mediante headSet, con base en "naranja"
20       System.out.print("headSet (\"naranja\"):  ");
21       imprimirConjunto(arbol.headSet("naranja") );
22
23       // obtiene subconjunto mediante tailSet, con base en "naranja"
24       System.out.print("tailSet (\"naranja\"):  ");
25       imprimirConjunto(arbol.tailSet("naranja") );
26
27       // obtiene los elementos primero y último
28       System.out.printf("primero: %s%n", arbol.first());
29       System.out.printf("ultimo : %s%n", arbol.last());
30    }
31
32    // imprime SortedSet en pantalla mediante instrucción for mejorada
33    private static void imprimirConjunto(SortedSet<String> conjunto)
34    {
35       for (String s : conjunto)
36          System.out.printf("%s ", s);
37
38       System.out.println();
39    }
40 } // fin de la clase PruebaSortedSet
```

Fig. 16.17 | Uso de SortedSet y TreeSet (parte 1 de 2).

```
conjunto ordenado: amarillo blanco carne gris naranja negro rojo verde
headSet ("naranja"):  amarillo blanco carne gris
tailSet ("naranja"):  naranja negro rojo verde
primero: amarillo
ultimo : verde
```

Fig. 16.17 | Uso de `SortedSet` y `TreeSet` (parte 2 de 2).

16.11 Mapas

Los objetos **`Map`** asocian *claves* a *valores*. Las claves en un objeto `Map` deben ser *únicas*, pero los valores asociados no. Si un objeto `Map` contiene claves y valores únicos, se dice que implementa una **asociación de uno a uno**. Si sólo las claves son únicas, se dice que el objeto `Map` implementa una **asociación de varios a uno**; muchas claves pueden asociarse a un solo valor.

Los objetos `Map` difieren de los objetos `Set` en tanto que los primeros contienen claves y valores, mientras que los segundos contienen solamente valores. Tres de las muchas clases que implementan a la interfaz `Map` son **`Hashtable`**, **`HashMap`** y **`TreeMap`**. Los objetos `Hashtable` y `HashMap` almacenan elementos en tablas de hash, y los objetos `TreeMap` almacenan elementos en árboles. En esta sección veremos las tablas de hash y proporcionaremos un ejemplo en el que se utiliza un objeto `HashMap` para almacenar pares clave-valor. La **interfaz `SortedMap`** extiende a `Map` y mantiene sus claves en *orden*; ya sea el orden *natural* de los elementos o un orden especificado por un objeto `Comparator`. La clase `TreeMap` implementa a `SortedMap`.

Implementación de `Map` con tablas de hash

Cuando un programa crea objetos, es probable que necesite almacenarlos y recuperarlos con eficiencia. Los procesos de ordenar y obtener información con los arreglos son eficiente si cierto aspecto de los datos coincide directamente con un valor de clave numérico, y si las *claves son únicas* y están estrechamente empaquetadas. Si tenemos 100 empleados con números de seguro social de nueve dígitos, y deseamos almacenar y recuperar los datos de los empleados mediante el uso del número de seguro social como una clave, para ello requeriríamos un arreglo con más de 800 millones de elementos, ya que los números del seguro social de nueve dígitos deben comenzar con 001-899 (excluyendo 666) según el sitio Web de la Administración del Seguro Social:

```
http://www.socialsecurity.gov/employer/randomization.html
```

Esto es impráctico para casi todas las aplicaciones que utilizan números de seguro social como claves. Un programa que tuviera un arreglo de ese tamaño podría lograr un alto rendimiento para almacenar y recuperar registros de empleados con sólo usar el número de seguro social como índice del arreglo.

Hay muchas aplicaciones con este problema; entre otras, que las claves son del tipo incorrecto (por ejemplo, enteros no positivos que corresponden a los subíndices del arreglo) o que son del tipo correcto, pero se esparcen *escasamente* sobre un *enorme rango*. Lo que se necesita es un esquema de alta velocidad para convertir claves, como números de seguro social, números de piezas de inventario y demás, en índices únicos de arreglo. Así, cuando una aplicación necesite almacenar algo, el esquema podría convertir rápidamente la clave de la aplicación en un índice, y el registro podría almacenarse en esa posición del arreglo. Para recuperar datos se hace lo mismo: una vez que la aplicación tenga una clave para la que desee obtener un registro de datos, simplemente aplica la conversión a la clave; esto produce el índice del arreglo en el que se almacenan y obtienen los datos.

El esquema que describimos aquí es la base de una técnica conocida como **hashing**. ¿Por qué ese nombre? Al convertir una clave en un índice de arreglo, literalmente revolvemos los bits, formando un

tipo de número "desordenado". En realidad, el número no tiene un significado real más allá de su utilidad para almacenar y obtener un registro de datos específico.

Un fallo en este esquema se denomina **colisión**; esto ocurre cuando dos claves distintas se asocian a la misma celda (o elemento) en el arreglo. No podemos almacenar dos valores en el mismo espacio, por lo que necesitamos encontrar un hogar alternativo para todos los valores más allá del primero, que se asocie con un índice de arreglo específico. Hay muchos esquemas para hacer esto. Uno de ellos es "hacer hash de nuevo" (es decir, aplicar otra transformación de hashing a la clave para proporcionar la siguiente celda como candidato en el arreglo). El proceso de hashing está diseñado para *distribuir* los valores en toda la tabla, por lo que se asume que se encontrará una celda disponible con sólo unas cuantas transformaciones de hashing.

Otro esquema utiliza un hash para localizar la primera celda candidata. Si esa celda está ocupada, se buscan celdas sucesivas en orden, hasta que se encuentra una disponible. El proceso de recuperación de datos funciona de la misma forma: se aplica hash a la clave una vez para determinar la función inicial y comprobar si contiene los datos deseados. Si es así, la búsqueda termina. En caso contrario, se busca linealmente en las celdas sucesivas hasta encontrar los datos deseados.

La solución más popular a las colisiones en las tablas de hash es hacer que cada celda de la tabla sea una "cubeta" de hash, que por lo general viene siendo una lista enlazada de todos los pares clave/valor que se asocian con esa celda. Ésta es la solución que implementan las clases `Hashtable` y `HashMap` (del paquete `java.util`). Tanto `Hashtable` como `HashMap` implementan a la interfaz `Map`. Las principales diferencias entre ellas son que `HashMap` *no está sincronizada* (varios subprocesos no deben modificar un objeto `HashMap` en forma concurrente); además permite claves y valores `null`.

El **factor de carga** de una tabla de hash afecta al rendimiento de los esquemas de hashing. El factor de carga es la proporción del número de celdas ocupadas en la tabla de hash, con respecto al número total de celdas en la tabla. Entre más se acerque esta proporción a 1.0, mayor será la probabilidad de colisiones.

Tip de rendimiento 16.2

El factor de carga en una tabla de hash es un clásico ejemplo de una concesión entre espacio de memoria y tiempo de ejecución: al incrementar el factor de carga, obtenemos un mejor uso de la memoria, pero el programa se ejecuta con más lentitud, debido al incremento en las colisiones de hashing. Al reducir el factor de carga, obtenemos más velocidad en la ejecución del programa, debido a la reducción en las colisiones de hashing, pero obtenemos un uso más pobre de la memoria, debido a que una proporción más grande de la tabla de hash permanece vacía.

Los estudiantes de ciencias computacionales estudian los esquemas de hashing en cursos titulados "Estructuras de datos" y "Algoritmos". Las clases `Hashtable` y `HashMap` permiten a los programadores utilizar la técnica de hashing sin tener que implementar los mecanismos de las tablas de hash: un clásico ejemplo de reutilización. Este concepto es muy importante en nuestro estudio de la programación orientada a objetos. Como vimos en capítulos anteriores, las clases encapsulan y ocultan la complejidad (es decir, los detalles de implementación) y ofrecen interfaces amigables para el usuario. La construcción apropiada de clases para exhibir tal comportamiento es una de las habilidades más valiosas en el campo de la programación orientada a objetos. En la figura 16.18 se utiliza un objeto `HashMap` para contar el número de ocurrencias de cada palabra en una cadena.

```
1  // Fig. 16.18: ConteoTipoPalabras.java
2  // Programa que cuenta el número de ocurrencias de cada palabra en un objeto String
3  import java.util.Map;
4  import java.util.HashMap;
5  import java.util.Set;
6  import java.util.TreeSet;
```

Fig. 16.18 | Programa que cuenta el número de ocurrencias de cada palabra en un objeto `String` (parte 1 de 3).

```
7  import java.util.Scanner;
8
9  public class ConteoTipoPalabras
10 {
11    public static void main(String[] args)
12    {
13       // crea HashMap para almacenar claves String y valores Integer
14       Map<String, Integer> miMap = new HashMap<>();
15
16       crearMap(miMap); // crea un mapa con base en la entrada del usuario
17       mostrarMap(miMap); // muestra el contenido del mapa
18    }
19
20    // crea mapa a partir de la entrada del usuario
21    private static void crearMap(Map<String, Integer> mapa)
22    {
23       Scanner scanner = new Scanner(System.in); // crea scanner
24       System.out.println("Escriba una cadena:"); // pide la entrada del usuario
25       String entrada = scanner.nextLine();
26
27       // divide la entrada en tokens
28       String[] tokens = entrada.split(" ");
29
30       // procesamiento del texto de entrada
31       for (String token : tokens)
32       {
33          String palabra = token.toLowerCase(); // obtiene una palabra en minúsculas
34
35          // si el mapa contiene la palabra
36          if (mapa.containsKey(palabra)) // ¿está la palabra en el mapa?
37          {
38             int cuenta = mapa.get(palabra); // obtiene la cuenta actual
39             mapa.put(palabra, cuenta + 1); // incrementa la cuenta
40          }
41          else
42             mapa.put(palabra, 1); // agrega una nueva palabra con una cuenta de 1 al mapa
43       }
44    }
45
46    // muestra el contenido del mapa
47    private static void mostrarMap(Map<String, Integer> mapa)
48    {
49       Set<String> claves = mapa.keySet(); // obtiene las claves
50
51       // ordena las claves
52       TreeSet<String> clavesOrdenadas = new TreeSet<>(claves);
53
54       System.out.printf("%nEl mapa contiene:%nClave/t/tValor%n");
55
56       // genera la salida para cada clave en el mapa
57       for (String clave : clavesOrdenadas)
58          System.out.printf("%-10s%10s%n", clave, mapa.get(clave));
59
```

Fig. 16.18 | Programa que cuenta el número de ocurrencias de cada palabra en un objeto `String` (parte 2 de 3).

```
60          System.out.printf(
61             "%nsize: %d%nisEmpty: %b%n", mapa.size(), mapa.isEmpty());
62       }
63    } // fin de la clase ConteoTipoPalabras
```

```
Escriba una cadena:
Ser o no ser; esa es la pregunta Si es mas noble sufrir

El mapa contiene:
Clave             Valor
es                 2
esa                1
la                 1
mas                1
no                 1
noble              1
o                  1
pregunta           1
ser                1
si                 1
sufrir             1

size: 12
isEmpty: false
```

Fig. 16.18 | Programa que cuenta el número de ocurrencias de cada palabra en un objeto `String` (parte 3 de 3).

En la línea 14 se crea un objeto `HashMap` vacío con una *capacidad inicial predeterminada* (16 elementos) y un factor de carga predeterminado (0.75); estos valores predeterminados están integrados en la implementación de `HashMap`. Cuando el número de posiciones ocupadas en el objeto `HashMap` se vuelve mayor que la capacidad multiplicada por el factor de carga, la capacidad se duplica en forma automática. `HashMap` es una clase genérica que recibe dos argumentos: el tipo de clave (es decir, `String`) y el tipo de valor (es decir, `Integer`). Recuerde que los argumentos de tipo que se pasan a una clase genérica deben ser tipos de referencias, por lo cual el segundo argumento de tipo es `Integer`, no `int`.

En la línea 16 se hace una llamada al método `crearMap` (líneas 21 a 44), el cual usa un objeto `Map` para almacenar el número de ocurrencias de cada palabra en la oración. En la línea 25 se obtiene la entrada del usuario y en la línea 28 se descompone en tokens. En las líneas 31 a 43 se convierte el siguiente token en minúsculas (línea 33) y luego se hace una llamada al **método `containsKey` de `Map`** (línea 36) para determinar si la palabra está en el mapa (y por ende, que ha ocurrido antes en la cadena). Si el objeto `Map` *no* contiene la palabra, en la línea 42 se utiliza el **método `put`** de `Map` para crear una nueva entrada en el mapa, con la palabra como la clave y un objeto `Integer` que contiene `1` como valor. La conversión autoboxing ocurre cuando el programa pasa el entero `1` al método `put`, ya que el mapa almacena el número de ocurrencias de la palabra como un objeto `Integer`. Si la palabra no existe en el mapa, en la línea 38 se utiliza el **método `get`** de `Map` para obtener el valor asociado de la clave (la cuenta) en el mapa. En la línea 39 se incrementa ese valor y se utiliza `put` para reemplazar el valor asociado de la clave. El método `put` devuelve el valor anterior asociado con la clave, o `null` si la clave no estaba en el mapa.

Tip para prevenir errores 16.2

Use siempre claves inmutables con un objeto `Map`. La clave determina en dónde se coloca el valor correspondiente. Si la clave cambió desde la operación de inserción, cuando intente de manera subsiguiente recuperar ese valor, podría no encontrarlo. En los ejemplos de este capítulo, usamos objetos `String` como claves, ya que son inmutables.

El método `mostrarMap` (líneas 47 a 62) muestra todas las entradas en el mapa. Utiliza el **método `keySet` de `HashMap`** (línea 49) para obtener un conjunto de las claves. Estas claves tienen el tipo `String` en el `mapa`, por lo que el método `keySet` devuelve un tipo genérico `Set` con el parámetro de tipo especificado como `String`. En la línea 52 se crea un objeto `TreeSet` de las claves, en el cual se ordenan éstas. El ciclo en las líneas 57 a 58 accede a cada clave y a su valor en el mapa. En la línea 58 se muestra cada clave y su valor, usando el especificador de formato `%-10s` para *alinear cada clave a la izquierda*, y el especificador de formato `%10s` para *alinear cada valor a la derecha*. Las claves se muestran en orden *ascendente*. En la línea 61 se hace una llamada al **método `size` de `Map`** para obtener el número de pares clave-valor en el objeto `Map`. En la línea 61 se hace una llamada al **método `isEmpty` de `Map`**, el cual devuelve un valor `boolean` que indica si el objeto `Map` está vacío o no.

16.12 La clase `Properties`

Un objeto **`Properties`** es un objeto `Hashtable` *persistente*, que por lo general almacena *pares clave-valor* de cadenas; suponiendo que el programador utiliza los métodos **`setProperty`** y **`getProperty`** para manipular la tabla, en vez de los métodos `put` y `get` heredados de `Hashtable`. Al decir "persistente", significa que el objeto `Properties` se puede escribir en un flujo de salida (posiblemente un archivo) y se puede leer de vuelta, a través de un flujo de entrada. Un uso común de los objetos `Properties` en versiones anteriores de Java era mantener los datos de configuración de una aplicación, o las preferencias del usuario para las aplicaciones. [*Nota*: la **API Preferences** (paquete `java.util.prefs`) está diseñada para reemplazar este uso específico de la clase `Properties`, pero esto se encuentra más allá del alcance de este libro. Para aprender más, visite el sitio Web `http://bit.ly/JavaPreferences`]. La clase `Properties` extiende a la clase `Hashtable<Object, Object>`. En la figura 16.19 se demuestran varios métodos de la clase `Properties`.

En la línea 13 se crea un objeto `Properties` llamado `tabla` sin propiedades predeterminadas. La clase `Properties` también cuenta con un constructor sobrecargado, el cual recibe una referencia a un objeto `Properties` que contiene valores de propiedad predeterminados. En cada una de las líneas 16 y 17 se hace una llamada al método `setProperty` de `Properties` para almacenar un valor para la clave especificada. Si la clave no existe en la `tabla`, `setProperty` devuelve `null`; en caso contrario, devuelve el valor anterior para esa clave.

```
1  // Fig. 16.19: PruebaProperties.java
2  // Demuestra la clase Properties del paquete java.util.
3  import java.io.FileOutputStream;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.util.Properties;
7  import java.util.Set;
8
9  public class PruebaProperties
10 {
11    public static void main(String[] args)
12    {
13       Properties tabla = new Properties();
14
15       // establece las propiedades
16       tabla.setProperty("color", "azul");
17       tabla.setProperty("anchura", "200");
18
19       System.out.println("Despues de establecer propiedades");
20       listarPropiedades(tabla);
```

Fig. 16.19 | La clase `Properties` del paquete `java.util` (parte 1 de 3).

```
21
22          // reemplaza el valor de una propiedad
23          tabla.setProperty("color", "rojo");
24
25          System.out.println("Despues de reemplazar propiedades");
26          listarPropiedades(tabla);
27
28          guardarPropiedades(tabla);
29
30          tabla.clear(); // vacía la tabla
31
32          System.out.println("Despues de borrar propiedades");
33          listarPropiedades(tabla);
34
35          cargarPropiedades(tabla);
36
37          // obtiene el valor de la propiedad color
38          Object valor = tabla.getProperty("color");
39
40          // comprueba si el valor está en la tabla
41          if (valor != null)
42             System.out.printf("El valor de la propiedad color es %s%n", valor);
43          else
44             System.out.println("La propiedad color no está en la tabla");
45       }
46
47       // guarda las propiedades en un archivo
48       private static void guardarPropiedades(Properties props)
49       {
50          // guarda el contenido de la tabla
51          try
52          {
53             FileOutputStream salida = new FileOutputStream("props.dat");
54             props.store(salida, "Propiedades de ejemplo"); // guarda las propiedades
55             salida.close();
56             System.out.println("Despues de guardar las propiedades");
57             listarPropiedades(props);
58          }
59          catch (IOException ioException)
60          {
61             ioException.printStackTrace();
62          }
63       }
64
65       // carga las propiedades de un archivo
66       private static void cargarPropiedades(Properties props)
67       {
68          // carga el contenido de la tabla
69          try
70          {
71             FileInputStream entrada = new FileInputStream("props.dat");
72             props.load(entrada); // carga las propiedades
73             entrada.close();
```

Fig. 16.19 | La clase `Properties` del paquete `java.util` (parte 2 de 3).

```
74            System.out.println("Despues de cargar las propiedades");
75            listarPropiedades(props);
76         }
77         catch (IOException ioException)
78         {
79            ioException.printStackTrace();
80         }
81      }
82
83      // imprime los valores de las propiedades
84      private static void listarPropiedades(Properties props)
85      {
86         Set<Object> claves = props.keySet(); // obtiene los nombres de las propiedades
87
88         // imprime los pares nombre/valor
89         for (Object clave : claves)
90            System.out.printf(
91               "%s\t%s%n", clave, props.getProperty((String) clave));
92
93         System.out.println();
94      }
95   } // fin de la clase PruebaProperties
```

```
Despues de establecer propiedades
anchura 200
color   azul

Despues de reemplazar propiedades
anchura 200
color   rojo

Despues de guardar las propiedades
anchura 200
color   rojo

Despues de borrar propiedades

Después de cargar las propiedades
anchura 200
color   rojo

El valor de la propiedad color es rojo
```

Fig. 16.19 | La clase `Properties` del paquete `java.util` (parte 3 de 3).

En la línea 38 se llama al método `getProperty` de `Properties` para localizar el valor asociado con la clave especificada. Si la clave *no* se encuentra en este objeto `Properties`, `getProperty` devuelve `null`. Una versión sobrecargada de este método recibe un segundo argumento, el cual especifica el valor predeterminado a devolver si `getProperty` no puede localizar la clave.

En la línea 54 se hace una llamada al **método `store` de `Properties`** para guardar el contenido del objeto `Properties` en el objeto `OutputStream` especificado como el primer argumento (en este caso, un objeto `FileOutputStream`). El segundo argumento, un objeto `String`, es una descripción que se escribe en el archivo. El **método `list` de la clase `Properties`**, que recibe un argumento `PrintStream`, es útil para mostrar la lista de propiedades.

En la línea 72 se hace una llamada al **método `load` de `Properties`** para restaurar el contenido del objeto `Properties` a partir del objeto `InputStream` especificado como el primer argumento (en este caso, un objeto `FileInputStream`). En la línea 86 se hace una llamada al método `keySet` de `Properties` para obtener un objeto `Set` de los nombres de las propiedades. Puesto que la clase `Properties` almacena su contenido en forma de objetos `Object`, se devuelve un objeto `Set` de referencias `Object`. En la línea 91 se obtiene el valor de una propiedad, para lo cual se pasa una clave al método `getProperty`.

16.13 Colecciones sincronizadas

En el capítulo 23 (en inglés, en el sitio web del libro) hablaremos sobre la tecnología *multihilos*. Con la excepción de `Vector` y `Hashtable`, las colecciones en el marco de trabajo de colecciones están *desincronizadas* de manera predeterminada, por lo que pueden operar eficientemente cuando no se requiere el uso de multihilos. Sin embargo, debido a que están desincronizadas, el acceso concurrente a un objeto `Collection` por parte de varios hilos podría producir resultados indeterminados, o errores fatales, como demostraremos en el capítulo 23. Para evitar potenciales problemas de uso de hilos, se utilizan **envolturas de sincronización** para las colecciones que podrían ser utilizadas por varios hilos. Un objeto **envoltura** recibe llamadas a métodos, agrega la sincronización de hilos (para evitar un acceso concurrente a la colección) y *delega* las llamadas al objeto de la colección envuelto. La API `Collections` proporciona un conjunto de métodos `static` para envolver colecciones como versiones sincronizadas. En la figura 16.20 se listan los encabezados para las envolturas de sincronización. Los detalles acerca de estos métodos están disponibles en `http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`. Todos estos métodos reciben un tipo genérico como parámetro y devuelven una *vista sincronizada* del tipo genérico. Por ejemplo, el siguiente código crea un objeto `List` sincronizado (`lista2`) que almacena objetos `String`:

```
List<String> lista1 = new ArrayList<>();
List<String> lista2 = Collections.synchronizedList(lista1);
```

Encabezados de los métodos `public static`

```
<T> Collection<T> synchronizedCollection(Collection<T> c)

<T> List<T> synchronizedList(List<T> unaLista)

<T> Set<T> synchronizedSet(Set<T> s)

<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)

<K, V> Map<K, V> synchronizedMap(Map<K, V> m)

<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m)
```

Fig. 16.20 | Métodos de envoltura de sincronización.

16.14 Colecciones no modificables

La clase `Collections` proporciona un conjunto de métodos `static` que crean **envolturas no modificables** para las colecciones. Las envolturas no modificables lanzan excepciones `UnsupportedOperationException` cuando se producen intentos por modificar la colección. En una colección no modificable las referencias almacenadas en la colección no pueden modificarse, pero los objetos a los que hacen referencia *son modificables* a menos que pertenezcan a una clase inmutable como `String`. En la figura 16.21 se listan los encabezados para estos métodos. Los detalles acerca de estos métodos están disponibles en `http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`. Todos estos métodos reciben un tipo genérico como parámetro y devuelven una vista no modificable del tipo

genérico. Por ejemplo, el siguiente código crea un objeto `List` no modificable (`lista2`) que almacena objetos `String`:

```
List<String> lista1 = new ArrayList<>();
List<String> lista2 = Collections.unmodifiableList(lista1);
```

Observación de ingeniería de software 16.6

Puede utilizar una envoltura no modificable para crear una colección que ofrezca acceso de sólo lectura a otras personas, mientras que a usted le permita acceso de lectura/escritura. Para ello, simplemente dé a los demás una referencia a la envoltura no modificable, y usted conserve una referencia a la colección original.

Encabezados de los métodos `public static`
`<T> Collection<T> unmodifiableCollection(Collection<T> c)`
`<T> List<T> unmodifiableList(List<T> unaLista)`
`<T> Set<T> unmodifiableSet(Set<T> s)`
`<T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)`
`<K, V> Map<K, V> unmodifiableMap(Map<K, V> m)`
`<K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, V> m)`

Fig. 16.21 | Métodos de envolturas no modificables.

16.15 Implementaciones abstractas

El marco de trabajo de colecciones proporciona varias implementaciones abstractas de interfaces de `Collection`, a partir de las cuales el programador puede construir implementaciones completas. Estas implementaciones abstractas incluyen una implementación de `Collection` delgada, llamada **`AbstractCollection`**; una implementación de `List` que permite el *acceso tipo arreglo* a sus elementos y se le conoce como **`AbstractList`**; una implementación de `Map` delgada conocida como **`AbstractMap`**, una implementación de `List` que permite un *acceso secuencial* (de principio a fin) a sus elementos y se le conoce como **`AbstractSequentialList`**, una implementación de `Set` conocida como **`AbstractSet`** y una implementación de `Queue` conocida como **`AbstractQueue`**. Puede aprender más acerca de estas clases en `http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html`. Para escribir una implementación *personalizada*, puede extender la implementación abstracta que se adapte mejor a sus necesidades, implementar cada uno de los métodos `abstract` de la clase y sobrescribir los métodos concretos de la clase según sea necesario.

16.16 Conclusión

En este capítulo se presentó el marco de trabajo de colecciones de Java. Conoció la jerarquía de colecciones y aprendió a utilizar las interfaces del marco de trabajo de colecciones para programar con las colecciones mediante el polimorfismo. Usó las clases `ArrayList` y `LinkedList`, las cuales implementan a la interfaz `List`. Presentamos las interfaces y clases integradas de Java para manipular pilas y colas. Usó varios métodos predefinidos para manipular colecciones. Aprendió a usar la interfaz `Set` y la clase `HashSet` para manipular una colección desordenada de valores únicos. Continuamos nuestra presen-

tación de los conjuntos con la interfaz `SortedSet` y la clase `TreeSet` para manipular una colección ordenada de valores únicos. Luego aprendió sobre las interfaces y clases de Java para manipular pares clave-valor: `Map`, `SortedMap`, `Hashtable`, `HashMap` y `TreeMap`. Hablamos sobre la clase `Properties` especializada para manipular pares clave-valor de objetos `String` que pueden almacenarse en un archivo y recuperarse de un archivo. Por último, hablamos sobre los métodos `static` de la clase `Collections` para obtener vistas no modificables y sincronizadas de colecciones. Si desea información adicional sobre el marco de trabajo de colecciones, visite `http://docs.oracle.com/javase/7/docs/technotes/guides/collections`. En el capítulo 17, Lambdas y flujos de Java SE 8, usará las nuevas herramientas de programación funcional de Java SE 8 para simplificar las operaciones con colecciones. En el capítulo 23 (en inglés, en el sitio web del libro) aprenderá a mejorar el rendimiento en los sistemas multinúcleo mediante las colecciones concurrentes y las operaciones de flujos en paralelo de Java.

Resumen

Sección 16.1 Introducción

- El marco de trabajo de colecciones de Java proporciona acceso a las estructuras de datos prefabricadas, así como a los métodos para manipularlas.

Sección 16.2 Generalidades acerca de las colecciones

- Una colección es un objeto que puede contener referencias a otros objetos.
- Las clases y las interfaces del marco de trabajo de colecciones se encuentran en el paquete `java.util`.

Sección 16.3 Clases de envoltura de tipos

- Las clases de envoltura de tipos (como `Integer`, `Double`, `Boolean`) permiten a los programadores manipular los valores de tipos primitivos como objetos (pág. 687). Los objetos de estas clases pueden usarse en colecciones.

Sección 16.4 Autoboxing y Autounboxing

- La operación boxing (pág. 687) convierte un valor primitivo en un objeto de la clase de envoltura de tipo correspondiente. La operación unboxing (pág. 687) convierte un objeto de envoltura de tipo en el valor primitivo correspondiente.
- Java realiza conversiones boxing y conversiones unboxing de manera automática.

Sección 16.5 La interfaz `Collection` y la clase `Collections`

- Las interfaces `Set` y `List` extienden a `Collection` (pág. 686), que contiene operaciones para agregar, borrar, comparar y retener objetos en una colección, y el método `iterator` (pág. 691) para obtener el objeto `Iterator` de una colección (pág. 687).
- La clase `Collections` (pág. 688) proporciona métodos `static` para manipular colecciones.

Sección 16.6 Listas

- Un objeto `List` (pág. 694) es un objeto `Collection` ordenado, que puede contener elementos duplicados.
- La interfaz `List` se implementa mediante las clases `ArrayList`, `LinkedList` y `Vector`. La clase `ArrayList` (pág. 688) es una implementación tipo arreglo que puede cambiar su tamaño. Un objeto `LinkedList` (pág. 688) es una implementación tipo lista enlazada de un objeto `List`.
- Java SE 7 soporta la inferencia de tipos con la notación <> en instrucciones que declaran y crean variables y objetos de tipo genérico.
- El método `hasNext` de `Iterator` (pág. 691) determina si un objeto `Collection` contiene otro elemento. El método `next` devuelve una referencia al siguiente objeto en el objeto `Collection`, y avanza el objeto `Iterator`.
- El método `subList` (pág. 694) devuelve una vista de una porción de un objeto `List`. Cualquier modificación realizada en esta vista se realiza también en el objeto `List`.

- El método `clear` (pág. 694) elimina elementos de un objeto `List`.
- El método `toArray` (pág. 695) devuelve el contenido de una colección, en forma de un arreglo.

Sección 16.7 Métodos de las colecciones

- Los algoritmos `sort` (pág. 697), `binarySearch`, `reverse` (pág. 702), `shuffle` (pág. 700), `fill` (pág. 702), `copy`, `addAll` (pág. 693), `frequency` y `disjoint` operan en objetos `List`. Los algoritmos `min` y `max` (pág. 703) operan en objetos `Collection`.
- El algoritmo `addAll` anexa a una colección todos los elementos en un arreglo (pág. 706), el algoritmo `frequency` (pág. 706) calcula cuántos elementos en la colección son iguales al elemento especificado y `disjoint` (pág. 706) determina si dos colecciones tienen elementos en común.
- Los algoritmos `min` y `max` buscan los elementos mayor y menor en una colección.
- La interfaz `Comparator` (pág. 697) proporciona un medio para ordenar los elementos de un objeto `Collection` en un orden distinto a su orden natural.
- El método `reverseOrder` (pág. 698) de `Collections` devuelve un objeto `Comparator` que puede usarse con `sort` para ordenar elementos de una colección en forma inversa.
- El algoritmo `shuffle` (pág. 700) ordena al azar los elementos de un objeto `List`.
- El algoritmo `binarySearch` (pág. 704) localiza un objeto `Object` en un objeto `List` ordenado.

Sección 16.8 La clase `Stack` del paquete `java.util`

- La clase `Stack` (pág. 708) extiende a `Vector`. El método `push` de `Stack` (pág. 709) agrega su argumento a la parte superior de la pila. El método `pop` (pág. 710) elimina el elemento superior de la pila. El método `peek` devuelve una referencia al elemento superior sin eliminarlo. El método `empty` (pág. 710) de `Stack` determina si la pila está vacía o no.

Sección 16.9 La clase `PriorityQueue` y la interfaz `Queue`

- La interfaz `Queue` (pág. 710) extiende a la interfaz `Collection` y proporciona operaciones adicionales para insertar, eliminar e inspeccionar elementos en una cola.
- `PriorityQueue` (pág. 710) implementa a la interfaz `Queue` y ordena los elementos con base en su orden natural o mediante un objeto `Comparator` que se suministra a través del constructor.
- El método `offer` de `PriorityQueue` (pág. 710) inserta un elemento en la ubicación apropiada, con base en el orden de prioridad. El método `poll` (pág. 710) elimina el elemento de mayor prioridad de la cola de prioridad. El método `peek` obtiene una referencia al elemento de mayor prioridad de la cola de prioridad. El método `clear` (pág. 710) elimina todos los elementos de la cola de prioridad. El método `size` (pág. 710) obtiene el número de elementos en la cola de prioridad.

Sección 16.10 Conjuntos

- Un objeto `Set` (pág. 711) es un objeto `Collection` desordenado que no contiene elementos duplicados. `HashSet` (pág. 711) almacena sus elementos en una tabla de hash. `TreeSet` (pág. 711) almacena sus elementos en un árbol.
- La interfaz `SortedSet` (pág. 712) extiende a `Set` y representa un conjunto que mantiene sus elementos ordenados. La clase `TreeSet` implementa a `SortedSet`.
- El método `headSet` (pág. 712) de `TreeSet` obtiene una vista de un objeto `TreeSet` que es menor a un elemento especificado. El método `tailSet` (pág. 713) obtiene una vista de `TreeSet` con elementos que son mayores o iguales a un elemento especificado. Cualquier modificación realizada a estas vistas se realiza al objeto `TreeSet`.

Sección 16.11 Mapas

- Los objetos `Map` (pág. 714) asocian claves con valores y no pueden contener claves duplicadas. Los objetos `HashMap` y `Hashtable` (pág. 714) almacenan elementos en tablas de hash, y los objetos `TreeMap` (pág. 714) almacenan elementos en árboles.

- `HashMap` recibe dos argumentos de tipo: el tipo de la clave y el tipo de valor.
- El método `put` (pág. 717) de `HashMap` agrega par clave-valor a un objeto `HashMap`. El método `get` (pág. 717) localiza el valor asociado con la clave especificada. El método `isEmpty` (pág. 718) determina si el mapa está vacío.
- El método `keySet` (pág. 718) de `HashMap` devuelve un conjunto de las claves. El método `size` (pág. 718) de `Map` devuelve el número de pares clave-valor en el objeto `Map`.
- La interfaz `SortedMap` (pág. 714) extiende a `Map` y representa un mapa que mantiene sus claves en orden. La clase `TreeMap` implementa a `SortedMap`.

Sección 16.12 La clase `Properties`

- Un objeto `Properties` (pág. 718) es una subclase persistente de `Hashtable`.
- El constructor de `Properties` sin argumentos crea una tabla `Properties` vacía. Un constructor sobrecargado recibe un objeto `Properties` predeterminado que contiene valores de propiedades predeterminados.
- El método `setProperty` de `Properties` (pág. 718) especifica el valor asociado con el argumento tipo clave. El método `getProperty` (pág. 718) localiza el valor de la clave especificada como argumento. El método `store` (pág. 720) guarda el contenido de un objeto `Properties` en el objeto `OutputStream` especificado. El método `load` (pág. 721) restaura el contenido del objeto `Properties` del objeto `InputStream` especificado.

Sección 16.13 Colecciones sincronizadas

- Las colecciones del marco de trabajo de colecciones están desincronizadas. Las envolturas de sincronización (pág. 721) se proporcionan para las colecciones a las que pueden acceder varios subprocesos en forma simultánea.

Sección 16.14 Colecciones no modificables

- Las envolturas de colecciones no modificables (pág. 721) lanzan excepciones `UnsupportedOperationException` (pág. 694) si hay intentos de modificar la colección.

Sección 16.15 Implementaciones abstractas

- El marco de trabajo de colecciones proporciona varias implementaciones abstractas de las interfaces de colecciones, a partir de las cuales el programador puede crear rápidamente implementaciones personalizadas completas.

Ejercicios de autoevaluación

16.1 Complete las siguientes oraciones:

a) Un objeto _____ se utiliza para iterar a través de una colección y puede eliminar elementos de la colección durante la iteración.
b) Para acceder a un elemento en un objeto `List`, se utiliza el _____ del elemento.
c) Suponiendo que `miArreglo` contenga referencias a objetos `Double`, _____ ocurre cuando se ejecuta la instrucción "`miArreglo[0] = 1.25;`".
d) Las clases _____ y _____ de Java proporcionan las herramientas de estructuras de datos tipo arreglo, que pueden cambiar su tamaño en forma dinámica.
e) Si usted no especifica un incremento de capacidad, el sistema _____ el tamaño del objeto `Vector` cada vez que se requiere una capacidad adicional.
f) Puede utilizar un _____ para crear una colección que ofrezca acceso de sólo lectura a los demás, mientras que a usted le permita el acceso de lectura/escritura.
g) Suponiendo que `miArreglo` contenga referencias a objetos `Double`, _____ ocurre cuando se ejecuta la instrucción "`double numero = miArreglo[0];`".
h) El algoritmo _____ de `Collections` determina si dos colecciones tienen elementos en común.

16.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los valores de tipos primitivos pueden almacenarse directamente en una colección.
b) Un objeto `Set` puede contener valores duplicados.

c) Un objeto `Map` puede contener claves duplicadas.
d) Un objeto `LinkedList` puede contener valores duplicados.
e) `Collections` es una interfaz (`interface`).
f) Los objetos `Iterator` pueden eliminar elementos.
g) Con la técnica de hashing, a medida que se incrementa el factor de carga, disminuye la probabilidad de colisiones.
h) Un objeto `PriorityQueue` permite elementos `null`.

Respuestas a los ejercicios de autoevaluación

16.1 a) `Iterator`. b) índice. c) autoboxing. d) `ArrayList`, `Vector`. e) duplicará. f) envoltura no modificable. g) autounboxing. h) `disjoint`.

16.2 a) Falso. La conversión autoboxing ocurre cuando se agrega un tipo primitivo a una colección, lo cual significa que el tipo primitivo se convierte en su clase de envoltura de tipo correspondiente.
b) Falso. Un objeto `Set` no puede contener valores duplicados.
c) Falso. Un objeto `Map` no puede contener claves duplicadas.
d) Verdadero.
e) Falso. `Collections` es una clase; `Collection` es una interfaz (`interface`).
f) Verdadero.
g) Falso. A medida que aumenta el factor de carga, hay menos posiciones disponibles, relativas al número total de posiciones, por lo que la probabilidad de una colisión se incrementa.
h) Falso. Al tratar de insertar un elemento `null` se produce una excepción `NullPointerException`.

Ejercicios

16.3 Defina cada uno de los siguientes términos:
a) `Collection`
b) Colecciones
c) `Comparator`
d) `List`
e) factor de carga
f) colisión
g) concesión entre espacio y tiempo en hashing
h) `HashMap`

16.4 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `Vector`:
a) `add`
b) `set`
c) `remove`
d) `removeAllElements`
e) `removeElementAt`
f) `firstElement`
g) `lastElement`
h) `contains`
i) `indexOf`
j) `size`
k) `capacity`

16.5 Explique por qué la operación de insertar elementos adicionales en un objeto `Vector`, cuyo tamaño actual sea menor que su capacidad, es una operación relativamente rápida, y por qué la inserción de elementos adicionales en un objeto `Vector`, cuyo tamaño actual sea igual a la capacidad, es una operación relativamente baja.

16.6 Al extender la clase `Vector`, los diseñadores de Java pudieron crear rápidamente la clase `Stack`. ¿Cuáles son los aspectos negativos de este uso de la herencia, en especial para la clase `Stack`?

16.7 Responda brevemente a las siguientes preguntas:
a) ¿Cuál es la principal diferencia entre un objeto `Set` y un objeto `Map`?
b) ¿Qué ocurre cuando agregamos un valor de tipo primitivo (por ejemplo, `double`) a una colección?
c) ¿Podemos imprimir todos los elementos en una colección sin utilizar un objeto `Iterator`? Si es así, explique cómo.

16.8 Explique brevemente la operación de cada uno de los siguientes métodos relacionados con `Iterator`:
a) `iterator`
b) `hasNext`
c) `next`

16.9 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `HashMap`:
a) `put`
b) `get`
c) `isEmpty`
d) `containsKey`
e) `keySet`

16.10 Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Los elementos en un objeto `Collection` deben almacenarse en orden ascendente, antes de poder realizar una búsqueda binaria mediante `binarySearch`.
b) El método `first` obtiene el primer elemento en un objeto `TreeSet`.
c) Un objeto `List` creado con el método `asList` de `Arrays` puede cambiar su tamaño.

16.11 Explique la operación de cada uno de los siguientes métodos de la clase `Properties`:
a) `load`
b) `store`
c) `getProperty`
d) `list`

16.12 Vuelva a escribir las líneas 16 a 25 en la figura 16.3 para que sean más concisas; utilice el método `asList` y el constructor de `LinkedList` que recibe un argumento `Collection`.

16.13 ***(Eliminación de duplicados)*** Escriba un programa que lea una serie de nombres de pila y elimine duplicados almacenándolos en un objeto `Set`. Permita al usuario buscar un nombre de pila.

16.14 ***(Conteo de letras)*** Modifique el programa de la figura 16.18 para contar el número de ocurrencias de cada letra, en vez de cada palabra. Por ejemplo, la cadena "`HOLA A TODOS`" contiene una `H`, tres `O`, una `L`, dos `A`, una `T`, una `D` y una `S`. Muestre los resultados.

16.15 ***(Selector de colores)*** Use un objeto `HashMap` para crear una clase reutilizable y elegir uno de los 13 colores predefinidos en la clase `Color`. Los nombres de los colores deben usarse como claves, y los objetos `Color` predefinidos deben usarse como valores. Coloque esta clase en un paquete que pueda importarse en cualquier programa en Java. Use su nueva clase en una aplicación que permita al usuario seleccionar un color y dibujar una figura en ese color.

16.16 ***(Conteo de palabras duplicadas)*** Escriba un programa que determine e imprima el número de palabras duplicadas en un enunciado. Trate a las letras mayúsculas y minúsculas de igual forma. Ignore los signos de puntuación.

16.17 ***(Insertar elementos en un objeto `LinkedList` en orden)*** Escriba un programa que inserte 25 enteros aleatorios de 0 a 100 en orden, en un objeto `LinkedList`. El programa debe ordenar los elementos, para luego calcular la suma de éstos y su promedio de punto flotante.

16.18 ***(Copiar e invertir objetos `LinkedList`)*** Escriba un programa que cree un objeto `LinkedList` de 10 caracteres; después el programa debe crear un segundo objeto `LinkedList` que contenga una copia de la primera lista, pero en orden inverso.

16.19 ***(Números primos y factores primos)*** Escriba un programa que reciba una entrada tipo número entero de un usuario, y que determine si es primo. Si el número no es primo, muestre sus factores primos únicos. Recuerde que los

factores de un número primo son sólo 1 y el mismo número primo. Todo número que no sea primo tiene una factorización prima única. Por ejemplo, considere el número 54. Los factores primos de 54 son 2, 3, 3 y 3. Cuando los valores se multiplican entre sí, el resultado es 54. Para el número 54, los factores primos a imprimir deben ser 2 y 3. Use objetos `Set` como parte de su solución.

16.20 ***(Ordenar palabras con un objeto `TreeSet`)*** Escriba un programa que utilice el método `split` de `String` para dividir en tokens una línea de texto introducida por el usuario, y que coloque cada token en un objeto `TreeSet`. Imprima los elementos del objeto `TreeSet`. [*Nota*: esto debe hacer que se impriman los elementos en orden ascendente].

16.21 ***(Cambiar el orden de un objeto `PriorityQueue`)*** Los resultados de la figura 16.15 muestran que `PriorityQueue` ordena elementos `Double` en orden ascendente. Vuelva a escribir la figura 16.15, de manera que ordene los elementos `Double` en forma descendente (es decir, `9.8` debe ser el elemento de mayor prioridad, en vez de `3.2`).