Si se sustituye `equals` hay que sustituir también el método `hashCode`, porque sino el `hashSet` no funcionará.

pistas idénticas. La especificación de `hashCode` dice que si dos objetos son declarados iguales por el método `equals`, entonces el método `hashCode` debe devolver el mismo valor para ellos, y si se viola esta especificación, el `HashSet` no podrá encontrar los objetos, aun cuando `equals` declare que hay una correspondencia. Si `equals` declara que los objetos no son iguales, el método `hashCode` debería devolver un valor distinto para ellos, aunque esto no es obligatorio. Sin embargo, aunque no sea obligatorio sí que es muy beneficioso para el rendimiento de `HashSet` que `hashCode` solo proporcione resultados idénticos para objetos desiguales en raras ocasiones. En el Capítulo 20 explicaremos cómo interactúan `hashCode` y `HashSet`.

La Figura 6.35 ilustra una clase `SimpleStudent` en la que dos objetos `SimpleStudent` son iguales si ambos tienen el mismo nombre (y ambos son de tipo `SimpleStudent`). Esto podría sustituirse utilizando las técnicas de la Figura 6.34 de la forma necesaria, o bien declarando este método como `final`. Si se le declara `final`, entonces la comprobación utilizada solo permitirá declarar como iguales a dos objetos `SimpleStudent` que sean de tipo idéntico. Si, teniendo un `equals` final, sustituimos la comprobación de la línea 40 por una comprobación `instanceof`, entonces cualesquiera dos objetos de la jerarquía podrán ser declarados iguales si sus nombres se corresponden.

El método `hashCode` de las líneas 47 y 48 simplemente utiliza el `hashCode` del campo `name`. Así, si dos objetos `SimpleStudent` tienen el mismo nombre (tal como lo declare `equals`), tendrán también el mismo `hashCode`, ya que, presumiblemente, los implementadores de `String` habrán respetado la especificación de `hashCode`.

El programa de prueba de acompañamiento forma parte de una prueba de mayor tamaño que ilustra todos los contenedores básicos. Observe que si el `hashCode` no se ha implementado, se añadirán al `HashSet` los tres objetos `SimpleStudent`, porque no se detectará el duplicado.

Resulta que, como promedio, las operaciones `HashSet` puede realizarse en un tiempo constante. Esto parece un resultado bastante sorprendente, porque implica que el coste de una única operación `HashSet` no depende de si el `HashSet` contiene 10 elementos o 10.000. La teoría que subyace al concepto de `HashSet` es fascinante y se describe en el Capítulo 20.

6.8 Mapas

Un `Map` se utiliza para almacenar una colección de entradas compuestas de *claves* y sus correspondientes *valores*. El mapa asigna claves a valores.

Un `Map` se utiliza para almacenar una colección de entradas formadas por sus *claves* y sus *valores*. El `Map` asigna claves a valores. Las claves deben ser diferentes, pero pueden asignarse varias claves a un mismo valor. Por tanto, los que no necesitan ser diferentes son los valores. Existe una interfaz `SortedMap` que mantiene el mapa ordenado, desde el punto de vista lógico, según las claves.

No es sorprendente que existan dos implementaciones: `HashMap` y `TreeMap`. `HashMap` no mantiene las claves en orden, mientras que `TreeMap` sí que lo hace. Por simplicidad, no vamos a implementar la interfaz `SortedMap`, pero sí que implementaremos `HashMap` y `TreeMap`.

El `Map` puede implementarse como un `Set` instanciado con un *par* (véase la Sección 3.9), cuyo comparador o implementación `equals`/`hashCode` solo hace referencia a la clave. La interfaz `Map` no amplía `Collection`; por el contrario, es independiente. En las Figuras 6.36 y 6.37 se muestra una interfaz de ejemplo que contiene los métodos más importantes.

```
1  package weiss.util;
2
3  /**
4   * Interfaz Map.
5   * Un mapa almacena parejas clave/valor.
6   * En nuestra implementación, no están permitidas las claves duplicadas.
7   */
8  public interface Map<KeyType,ValueType> extends java.io.Serializable
9  {
10    /**
11     * Devuelve el número de claves en este mapa.
12     */
13    int size( );
14
15    /**
16     * Comprueba si este mapa está vacío.
17     */
18    boolean isEmpty( );
19
20    /**
21     * Comprueba si este mapa contiene una clave especificada.
22     */
23    boolean containsKey( KeyType key );
24
25    /**
26     * Devuelve el valor que se corresponde con la clave o null
27     * si no se encuentra la clave. Dado que están permitidos los valores
28     * null, comprobar si el valor de retorno es null puede no ser una forma
29     * segura de determinar si la clave está presente en el mapa.
30     */
31    ValueType get( KeyType key );
32
33    /**
34     * Añade la pareja clave/valor al mapa, sustituyendo el valor
35     * original en caso de que la clave ya estuviera presente.
36     * Devuelve el antiguo valor asociado con la clave o
37     * null si la clave  no estaba presente antes de esta llamada.
38 */
39 ValueType put( KeyType key, ValueType value );
40
41    /**
42     * Elimina la clave y su valor del mapa.
43     * Devuelve el valor previamente asociado con la clave
44     * o null si la clave no estaba presente antes de esta llamada.
45     */
46 ValueType remove( KeyType key );
```

Figura 6.36 Una interfaz `Map` de ejemplo (parte 1).

```
47    /**
48     * Elimina todas las parejas clave/valor del mapa.
49     */
50    void clear( );
51
52    /**
53     * Devuelve las claves del mapa.
54     */
55    Set<KeyType> keySet( );
56
57    /**
58     * Devuelve los valores del mapa. Puede haber duplicados.
59     */
60    Collection<ValueType> values( );
61
62    /**
63     * Devuelve un conjunto de objetos Map.Entry correspondientes
64     * a las parejas clave/valor contenidas en el mapa.
65     */
66    Set<Entry<KeyType,ValueType>> entrySet( );
67
68    /**
69     * Interfaz utilizada para acceder a las parejas clave/valor de un mapa.
70     * A partir de un mapa, utilice entrySet().iterator para obtener un
71     * iterador sobre un Set de parejas. El método next() de este iterador
72     * proporciona objetos de tipo Map.Entry<KeyType,ValueType>.
73     */
74    public interface Entry<KeyType,ValueType> extends java.io.Serializable
75    {
76       /**
77        * Devuelve la clave de esta pareja.
78        */
79       KeyType getKey( );
80
81       /**
82        * Devuelve el valor de esta pareja.
83     */
84       ValueType getValue( );
85
86       /**
87        * Cambia el valor de esta pareja.
88        * @return el valor antiguo asociado con esta pareja.
89        */
90       ValueType setValue( ValueType newValue );
91    }
92 }
```

Figura 6.37 Una interfaz `Map` de ejemplo (parte 2).

La mayoría de los métodos tienen un semántica intuitiva. El método `put` se utiliza para añadir un pareja clave/valor, `remove` se emplea para eliminar una pareja clave/valor (solo se especifica la clave) y `get` devuelve el valor asociado con una clave. Están permitidos valores `null`, lo que complica las cosas para `get`, ya que el valor de retorno proporcionado por `get` no permitirá distinguir entre una búsqueda fallida y una búsqueda que haya tenido éxito y que devuelva `null` como valor. Si se sabe que existen valores `null` en el mapa, puede emplearse `containsKey`.

La interfaz `Map` no proporciona un método `iterator` ni una clase iteradora. En lugar de ello, devuelve una `Collection` que puede utilizarse para visualizar los contenidos del mapa.

El método `keySet` proporciona una `Collection` que contiene todas las claves. Puesto que no están permitidas las claves duplicadas, el resultado de `keySet` es un `Set`, para el cual podemos obtener un iterador. Si el `Map` es un `SortedMap`, el `Set` es un `SortedSet`.

De forma similar, el método `values` devuelve una `Collection` que contiene todos los valores. En este caso se trata realmente de una `Collection`, ya que los valores duplicados están permitidos.

`Map.Entry` abstrae la noción de una pareja dentro del mapa.

Finalmente, el método `entrySet` devuelve una `Collection` de parejas clave/valor. De nuevo, se trata de un `Set`, porque las parejas deben tener claves distintas. Los objetos del `Set` devueltos por el `entrySet` son parejas; debe haber un tipo que represente a esas parejas clave/valor. Este tipo es especificado por la interfaz `Entry` anidada dentro de la interfaz `Map`. Por tanto, el tipo de objeto almacenado en el `entrySet` es `Map.Entry`.

La Figura 6.38 ilustra el uso del `Map` con `TreeMap`. En la línea 23 se crea un mapa vacío y luego se rellena con una serie de llamadas a `put` en las líneas 25 a 29. La última llamada a `put` simplemente sustituye un valor con `"unlisted"`. Las líneas 31 y 32 imprimen el resultado de una llamada a `get`, que se utiliza para obtener el valor correspondiente a la clave `"Jane Doe"`. Más interesante es la rutina `printMap` que abarca las líneas 8 a 19.

En `printMap`, en la línea 12, obtenemos un `Set` que contiene parejas `Map.Entry`. A partir del `Set`, podemos utilizar un bucle `for` avanzado para visualizar las entradas `Map.Entry` y podemos obtener la información de clave y valor utilizando `getKey` y `getValue`, como se muestra en las líneas 16 y 17.

`keySet`, `values` y `entrySet` devuelven vistas.

Volviendo a `main`, vemos que `keySet` devuelve un conjunto de claves (en la línea 37) que pueden imprimirse en la línea 38 llamando a `printCollection` (en la Figura 6.11); de forma similar, en las líneas 41 y 42, `values` devuelve una colección de valores que se puede imprimir. Más interesante resulta el que el conjunto de claves y la colección de valores son *vistas* del mapa, por lo que los cambios en el mapa se ven inmediatamente reflejados en el conjunto de claves y la colección de valores, y las eliminaciones que efectuemos en el conjunto de claves o en el de valores se convierten en eliminaciones en el mapa subyacente. Por tanto, la línea 44 no solo elimina la clave del conjunto de claves, sino también la entrada asociada en el mapa. De forma similar, la línea 45 elimina una entrada del mapa. Por tanto, la operación de impresión en la línea 49 refleja un mapa en el que se han eliminado dos entradas.

Las vistas en sí mismas constituyen un concepto interesante y explicaremos los detalles específicos acerca de cómo se implementan más adelante, cuando implementemos las clases de mapas. En la Sección 6.10 se exponen algunos ejemplos adicionales de vistas.

La Figura 6.39 ilustra otro uso del mapa, en un método que devuelve los elementos de una lista que aparecen más de una vez. En este código, se está utilizando internamente un mapa para agrupar

```
1  import java.util.Map;
2  import java.util.TreeMap;
3  import java.util.Set;
4  import java.util.Collection;
5
6  public class MapDemo
7  {
8     public static <KeyType,ValueType>
9     void printMap( String msg, Map<KeyType,ValueType> m )
10    {
11       System.out.println( msg + ":" );
12       Set<Map.Entry<KeyType,ValueType>> entries = m.entrySet( );
13
14       for( Map.Entry<KeyType,ValueType> thisPair : entries )
15       {
16          System.out.print( thisPair.getKey( ) + ": " );
17       System.out.println( thisPair.getValue( ) );
18       }
19    }
20
21    public static void main( String [ ] args )
22    {
23       Map<String,String> phone1 = new TreeMap<String,String>( );
24
25       phone1.put( "John Doe", "212-555-1212" );
26       phone1.put( "Jane Doe", "312-555-1212" );
27       phone1.put( "Holly Doe", "213-555-1212" );
28       phone1.put( "Susan Doe", "617-555-1212" );
29       phone1.put( "Jane Doe", "unlisted" );
30
31       System.out.println( "phone1.get(\"Jane Doe\"): " +
32                           phone1.get( "Jane Doe" ) );
33       System.out.println( "\nThe map is: " );
34       printMap( "phone1", phone1 );
35
36       System.out.println( "\nThe keys are: " );
37       Set<String> keys = phone1.keySet( );
38       printCollection( keys );
39
40       System.out.println( "\nThe values are: " );
41       Collection<String> values = phone1.values( );
42       printCollection( values );
43
44       keys.remove( "John Doe" );
45       values.remove( "unlisted" );
46
47       System.out.println( "After John Doe and 1 unlisted are removed" );
48       System.out.println( "\nThe map is: " );
49       printMap( "phone1", phone1 );
50    }
51 }
```

Figura 6.38 Una ilustración de cómo se utiliza la interfaz `Map`.

```
1  public static List<String> listDuplicates( List<String> coll )
2  {
3     Map<String,Integer> count = new TreeMap<String,Integer>( );
4     List<String> result = new ArrayList<String>( );
5
6     for( String word : coll )
7     {
8        Integer occurs = count.get( word );
9        if( occurs == null )
10          count.put( word, 1 );
11       else
12          count.put( word, occurs + 1 );
13    }
14
15    for( Map.Entry<String,Integer> e : count.entrySet( ) )
16       if( e.getValue( ) >= 2 )
17          result.add( e.getKey( ) );
18
19    return result;
20 }
```

Figura 6.39 Un uso típico de un mapa.

los duplicados: la clave del mapa es un elemento y el valor es el número de veces que el elemento ha aparecido. Las líneas 8-12 ilustran la idea típica que podemos ver a la hora de construir un mapa de esta forma. Si el elemento nunca ha sido insertado en el mapa, lo hacemos con un recuento igual a 1. En caso contrario, actualizamos el recuento. Observe el juicioso uso de los mecanismos de *autoboxing* y *unboxing*. Después, en las líneas 15-17 utilizamos un iterador para recorrer el conjunto de entradas, obteniendo las claves que aparezcan con un recuento mayor o igual que dos en el mapa.

6.9 Colas con prioridad

La *cola con prioridad* solo permite acceder al elemento mínimo.

Aunque los trabajos de impresión enviados a una impresora suelen colocarse en una cola, esa política puede no ser siempre la más adecuada. Por ejemplo, un trabajo de impresión podría ser particularmente importante, por lo que desearíamos poder permitir que ese trabajo se imprima en cuanto la impresora esté disponible. A la inversa, cuando la impresora finalice un trabajo y haya varios trabajos de 1 página y uno de 100 páginas esperando, puede ser razonable imprimir el último trabajo al final, aun cuando no sea el último trabajo enviado. (Lamentablemente, la mayoría de los sistemas no hacen esto, lo cual puede resultar particularmente molesto en ciertas ocasiones.)

De forma similar, en un entorno multiusuario, el planificador del sistema operativo debe decidir cuál de entre varios procesos ejecutar. En general, a cada proceso solo se le permite ejecutarse durante un periodo de tiempo fijo. Un algoritmo no muy adecuado para implementar este tipo de