Tema 6

Colecciones

Los aficionados a la filatelia, a la numismática o a las colecciones de cromos, usan habitualmente álbumes para guardar sus sellos, monedas, o cromos. Análogamente, los desarrolladores de aplicaciones se ven a menudo en la necesidad de almacenar en la memoria del computador colecciones de elementos. Actualmente se conocen múltiples **estructuras de datos** o formas de almacenar los elementos de una colección en la memoria de un computador. Esas estructuras tienen una naturaleza dinámica en el sentido de que, durante la ejecución del programa que las crea, no es raro que se incorporen nuevos elementos o que se retiren algunos de los que estaban. *java* pone a nuestra disposición una familia de clases que nos permiten beneficiarnos de ese conocimiento, sin tener que conocer y codificar los detalles de las estructuras de datos utilizadas con más frecuencia. A continuación veremos las principales características de algunas de esas clases.

6.1. Conjuntos

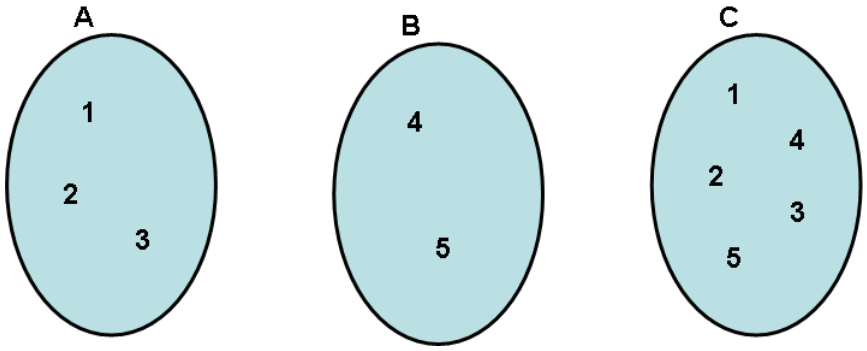
En programación es relativamente frecuente encontrarse en la necesidad de almacenar en la memoria del computador **conjuntos** de elementos. Para ello es posible optar entre diferentes estructuras de datos, que son una opción razonable cuando se van a usar, sobre todo, para saber si un elemento forma parte del conjunto. La elección de la palabra *conjunto* no es casual. Un aspecto a tener en cuenta de este tipo de colecciones es que, por definición, sus elementos no se repiten. Si fuera así, estaríamos hablando de *multiconjuntos* o de otro tipo de colecciones que se verán después.

Por lo que se refiere a *java*, y dicho informalmente, un objeto de la clase `TreeSet<Integer>` almacena un conjunto de números enteros. Aunque de recién creado, un `TreeSet<Integer>` está vacío y no contiene ningún elemento, usando los métodos de esa clase podemos añadirle tantos elementos como deseemos. Por ejemplo, con las instrucciones siguientes:

```
TreeSet<Integer> a = new TreeSet<Integer>();
TreeSet<Integer> b = new TreeSet<Integer>();
TreeSet<Integer> c = new TreeSet<Integer>();
/* a, b, c estan vacios */
a.add(1); /* se anyade el 1 al conjunto a */
a.add(2); /* se anyade el 2 al conjunto a */
a.add(3); /* se anyade el 3 al conjunto a */
```

```
b.add(4); /* se anyade el 4 al conjunto b */
b.add(5); /* se anyade el 5 al conjunto b */
c.add(1); /* se anyade el 1 al conjunto c */
c.add(2); /* se anyade el 2 al conjunto c */
c.add(3); /* se anyade el 3 al conjunto c */
c.add(4); /* se anyade el 4 al conjunto c */
c.add(5); /* se anyade el 5 al conjunto c */
```

se crearán tres objetos y se les añadirán sucesivos elementos. Al completarse esas instrucciones, los objetos `a`, `b` y `c` que se han creado representan los conjuntos de la figura siguiente:



También se pueden manipular conjuntos cuyos elementos sean valores de otros tipos; las clases correspondientes se llaman así:

Clase	*tipo de elementos*
`TreeSet<Double>`	Valores en coma flotante
`TreeSet<Boolean>`	Valores lógicos (`true`, `false`)
`TreeSet<Character>`	caracteres (Códigos Unicode)
`TreeSet<String>`	ristras de caracteres

Todas las clases de conjuntos mencionadas incluyen métodos parecidos. Básicamente, podemos añadir un elemento a un conjunto, retirar un elemento de un conjunto, o determinar si un elemento pertenece o no a un conjunto. Las cabeceras de esos métodos se incluyen en la figura 6.1. Por brevedad, no se repite la declaración de esos métodos para los diferentes tipos de elementos; basta con sustituir en la figura 6.1 el símbolo **T** por el nombre del tipo adecuado a cada caso: `Integer`, `Double`, `Boolean`, `Character`, `String`.

6.1.1. Un ejemplo: repetición de valores en orden creciente

El programa 20 muestra un ejemplo muy sencillo de uso de conjuntos. Ese programa permite al usuario introducir tantos valores enteros como desee y, acto seguido, escribe en orden creciente todos los elementos distintos de la secuencia de valores introducida por el usuario. Para almacenar los números que dicte el usuario se crea un objeto `TreeSet<Integer>`, en la línea 16. Obsérvese que cada número introducido por el usuario es añadido a un conjunto mediante el método `add`, en la línea 22. Lo que ocurre es que si un número ya estaba en el conjunto, porque el usuario lo introdujo antes, el objeto `conjtoValoresDistintos` permanece inalterado. Por último, en la línea 26 se usa el método `toString`, que no se nombró en

public boolean `add(T s)`
si `s` no pertenece al conjunto, se añade el elemento `s`; en caso contrario, el conjunto no se modifica
devuelve `true` si `s` no pertenecía al conjunto

public boolean `remove(T s)`
si `s` pertenece al conjunto, se retira el elemento `s`; en caso contrario, el conjunto no se modifica
devuelve `true` si `s` pertenecía al conjunto

public boolean `contains(T s)`
devuelve `true` si `s` pertenece al conjunto; `false` en caso contrario

public void `clear()`
se eliminan todos los elementos del conjunto

public int `size()`
devuelve el número de elementos del conjunto

public T `first()`
devuelve el menor elemento del conjunto.

Figura 6.1: Algunos de los métodos de las clases `TreeSet<T>`

la figura 6.1. El envío de ese mensaje a un conjunto sirve para solicitarle un texto con los elementos del conjunto. Como el formato del texto construido por el conjunto raramente se ajusta a las necesidades de cada problema particular, la utilidad de ese método es algo limitada.

6.1.1.1. ¿`TreeSet<Integer>` o `java.util.TreeSet<Integer>`?

En el ejemplo anterior, en lugar de usar el nombre `TreeSet<Integer>` se ha usado uno algo más raro:

`java.util.TreeSet<Integer>`

En *java* se considera que todas las clases existentes están agrupadas en *paquetes*. Un *paquete* es simplemente una agrupación de clases con un nombre, formado por una o varias palabras separadas por un punto. Por ejemplo, tanto la clase `TreeSet<Integer>` como las demás clases de colecciones que veremos en lo que sigue pertenecen al paquete:

`java.util`

Una clase pertenece al paquete cuyo nombre aparece después de la palabra **package** en el texto de esa clase:

package `nombre.de.paquete;`

Programa 20 Repetición de valores en orden creciente

```
1  package ehu.student.collections;
2
3  /**
4   * Repetir los numeros que diga el usuario,
5   * eliminando repeticiones y en orden creciente.
6   */
7  public class RepetirSecuenciaEnOrden {
8
9      public static void main(String[] args){
10         acm.io.IODialog dialog = new acm.io.IODialog();
11
12         /*
13          * Conjunto donde se se almacenaran los valores leidos
14          */
15         java.util.TreeSet<Integer> conjtoValoresDistintos =
16             new java.util.TreeSet<Integer>();
17
18         while(dialog.readBoolean("Continuar?")){
19             int s = dialog.readInt("Siguiente?");
20
21             /* "s" se añade al conjunto, si no estaba ya */
22             conjtoValoresDistintos.add(s);
23         }
24
25         /* Construir un texto con los valores del conjunto... */
26         String texto = conjtoValoresDistintos.toString();
27
28         System.out.println(texto); /* ... y escribirlo*/
29
30         System.exit(0);
31     }
32 }
```

Cuando usamos el nombre de una clase, para crear una instancia, para declarar una variable, etc, normalmente escribiremos simplemente su nombre. No obstante, también cabe la posibilidad de escribir el nombre del paquete al cual pertenece esa clase seguido por su nombre, como en:

```
java.util.TreeSet<Integer> set = new java.util.TreeSet<Integer>();
```

En el primer caso, estamos usando el *nombre simple* de la clase, y en el segundo lo que se conoce como *nombre cualificado*; podríamos decir que éste es el nombre completo y el primero la forma abreviada. Ahora bien, para elegir entre las dos opciones referidas hay que tener en cuenta que:

- para que podamos usar en nuestro programa el nombre simple de otra clase, como `TreeSet<Integer>`, hay que incluir una línea como la siguiente

```
import java.util.TreeSet;
```

en la cabecera de nuestro programa. En el ejemplo anterior esa línea debería insertarse en la línea 3.

- no siempre se puede usar el nombre simple

Si se usa un entorno de programación como *Eclipse*, no merece la pena entrar a comentar detalles relacionados con el uso de nombres simples o de nombres cualificados. Por un lado, aunque no sea el estilo habitual, podemos optar por escribir siempre el nombre cualificado, como se ha hecho en el ejemplo anterior, ya que con *Eclipse* eso no supone ningún esfuerzo. Por otro lado, podemos escribir el nombre completo en la primera instrucción en la que usemos el nombre de una clase; en el ejemplo, eso ocurre en las líneas 17 y 18. Después de completar esa instrucción, podemos cambiarlo por el nombre simple; *Eclipse* nos ayudará a hacer las correcciones oportunas, y en instrucciones posteriores bastará con escribir directamente el nombre simple.

6.1.2. Instrucciones de iteración y conjuntos

Muchas de las tareas relacionadas con una colección de elementos obligan a *recorrer* la estructura de datos correspondiente, para someter cada elemento de la colección a algún tipo de procesamiento, pero sin alterar el contenido de la colección. Por ejemplo, en el programa 20 las líneas 26 y 28 se usan para escribir los valores contenidos en el conjunto llamado `conjtoValoresDistintos`. Si no nos gusta la forma en que ese programa presenta sus resultados, y, por ejemplo, queremos que cada valor del conjunto aparezca en una línea distinta, entonces hay que hacer algunos cambios. Básicamente, habrá que sustituir las líneas referidas por algo que nos permita escribir cada valor en una línea distinta; está claro que, en este caso, el tratamiento que hay que aplicar a cada elemento es muy simple.

En *java* la manera más adecuada de procesar cada elemento de un conjunto de enteros es mediante una instrucción **for** con la estructura siguiente:

```
for (Integer v: S){
    /*
     * bloque de instrucciones
     */
}
```

donde **v** y **S** deben sustituirse por nombres de variables; **v** debe ser un nombre de variable que *no* haya sido ya declarada, y **S** debe ser una variable ya declarada de tipo

```
TreeSet<Integer>
```

que indica cuál es el conjunto de elementos a recorrer. Una instrucción así se incluye en el programa siguiente:

```
1 package ehu.student.collections;
2
3 import java.util.TreeSet;
4
```

```
5  /**
6   * Uso de instruccion for para procesar los elementos de un conjunto
7   */
8  public class RecorridoDeConjuntoTriv {
9
10     public static void main(String[] args)
11     {
12         TreeSet<Integer> conjunto = new TreeSet<Integer>();
13         /* añadir elementos varios */
14         conjunto.add(+1);
15         conjunto.add(-1);
16
17         /* Para cada elemento del conjunto... */
18         for (@SuppressWarnings("unused")
19                 Integer var: conjunto){
20             System.out.println("Hola");
21         }
22     }
23 }
```

Este programa crea un conjunto de enteros con un par de elementos, aunque evidentemente podrían ser más. La instrucción **for** que se incluye en él, escribe una vez el mensaje `Hola` por cada uno de los elementos del conjunto recorrido.

Una instrucción **for** con la estructura indicada arriba se ejecuta repitiendo su bloque de instrucciones,

una vez por cada elemento del conjunto designado por **S**

En cada iteración, a la variable **v**, se le asigna automáticamente un elemento del conjunto, de menor a mayor. Por lo que se refiere al tipo de esa variable, omitiremos los detalles por ahora, y considereremos que `Integer` y `int` son equivalentes; luego veremos que entre ambos términos hay algunas diferencias. El uso de la variable **v** en las instrucciones del bloque (de hecho, no puede usarse fuera de él), proporciona el mecanismo para describir el tratamiento a realizar con cada elemento. Así, en el programa siguiente, muy parecido al anterior, el valor de cada elemento del conjunto es sumado a la variable `suma`, cuyo valor acabará siendo igual a la suma de los elementos del conjunto creado.

```
1  package ehu.student.collections;
2
3  import java.util.TreeSet;
4
5  /**
6   * Uso de instruccion for para procesar los elementos de un conjunto
7   */
8  public class RecorridoDeConjuntoParaSumarTodos {
9
10     public static void main(String[] args){
11
12         TreeSet<Integer> conjunto = new TreeSet<Integer>();
13         /* añadir elementos varios */
14         conjunto.add(+1);
```

```
15          conjunto.add(-1);
16
17          int suma = 0;  /* ... de los elementos */
18
19          /* Para cada elemento del conjunto... */
20          for (Integer var: conjunto){
21              suma = suma+var; /* sumarselo a suma */
22          }
23          System.out.println("Suma = " + suma);
24      }
25  }
```

De manera similar podría modificarse el programa 20, de manera que se escriba en una línea distinta cada elemento del conjunto `conjtoValoresDistintos`. Bastaría con sustituir las líneas 26 y 28 de ese programa por:

```
for (Integer v: conjtoValoresDistintos){
  System.out.println(v);
}
```

Este tipo de instrucciones puede usarse también cuando los elementos del conjunto recorrido son de otros tipos. En general, tienen el formato siguiente:

```
for (T v: S){
    /*
     * bloque de instrucciones
     */
}
```

donde **v** y **S** deben sustituirse por nombres de variables, conforme a lo dicho antes, y **T** debe sustituirse por `Integer`, `Double`, `Boolean`, `Character`, `String`, según qué tipo de elementos formen el conjunto en cuestión.

6.1.3. Un ejemplo: cálculo de los primeros elementos de un conjunto

Imaginemos que dada una secuencia de palabras introducida por el usuario, se deben seleccionar las primeros palabras de esa secuencia, en orden alfabético, sin tener en cuenta eventuales repeticiones. Es decir, una vez leidas tantas palabras como desee el usuario, el programa deberá presentar las primeras en orden alfabético de todas las introducidas, seleccionando tantas como indique el usuario. Por ejemplo, supongamos que la secuencia dada por el usuario es:

beta alfa omega alfa omega alfa

Si, a continuación, el usuario introduce el número dos, el programa mostrará las dos palabras siguientes:

alfa beta

Si por el contrario el usuario introduce el número tres, el programa mostrará:

alfa beta omega

Programa 21 Seleccionando las primeras palabras de un conjunto

```
1  package ehu.student.collections;
2
3  import java.util.TreeSet;
4
5  /**
6   * Calculo de las primeras palabras de una secuencia dada.
7   *
8   * El usuario introducira tantas palabras como desee.
9   * A continuacion, para cada valor K que indique el usuario,
10  * se escribiran las K primeras palabras distintas de todas ellas.
11  */
12 public class SeleccionSimpleDePrimerasPalabras {
13
14     public static void main(String[] args){
15         acm.io.IODialog dialog = new acm.io.IODialog();
16
17         /* las palabras que diga el usuario se añaden a un conjunto */
18         TreeSet<String> palabras = new TreeSet<String>();
19         while(dialog.readBoolean("Continuar añadiendo palabras?")){
20             String s=dialog.readLine("Siguiente?");
21             palabras.add(s); /* añadir "s" al conjunto de palabras leidas */
22         }
23
24         while(dialog.readBoolean("Continuar seleccionando?")){
25             int K = dialog.readInt("K?");
26
27             /*
28              * Presentar las primeras K palabras
29              */
30             int contadorDePalabrasEscritas = 0; //escribir solo las K primeras!
31             for (String s: palabras){ //Para cada palabra...
32                 System.out.println(s);//... se escribe
33                 contadorDePalabrasEscritas++;
34                 if (contadorDePalabrasEscritas == K){
35                     /* Ya se han escrito las K primeras */
36                     break;
37                 }
38             }
39         }
40         System.exit(0);
41     }
42 }
```

Usando un conjunto de palabras el problema no tiene apenas dificultad; el programa 21 muestra una solución. Cada palabra que el usuario introduce se añade al conjunto de palabras leidas; por definición, ese conjunto no contiene palabras repetidas. A continuación, para cada entero `K` que seleccione el usuario, se escriben las primeras K palabras del conjunto referido. La instrucción **for** incluida en la línea 31 permite someter cada palabra al tratamiento adecuado: recuérdese que esa instrucción ejecuta el bloque de instrucciones una vez por cada palabra,

empezando por la primera en órden alfabético. Por otro lado, el tratamiento referido se reduce, en este caso, a escribir la palabra e incrementar un contador, ya que una vez escrita la cantidad de palabras elegida se puede dar por terminado el recorrido del conjunto de palabras.

6.1.3.1. Métodos y conjuntos

Al definir una clase, los parámetros de sus métodos pueden ser también referencias de objetos, al igual que el resultado producido. Aunque quizás no sea muy realista, podríamos pensar en reorganizar el programa anterior, definiendo las clases de objetos adecuadas, de manera que algunas de las partes de ese programa se puedan reutilizar. Por ejemplo, el bucle que aparece en la línea 19, se usa para añadir las palabras que indique el usuario a un conjunto. Definiendo una clase de objetos que sean especialistas en crear conjuntos de palabras, ese bucle podría re-emplazarse por algo como lo siguiente:

```
TreeSet<String> palabras = lector.nuevoConjuntoDeString();
```

La clase referida podría definirse como sigue:

```
1  package ehu.student.collections;
2
3  import java.util.TreeSet;
4
5  /**
6   * Clase de conveniencia para crear conjuntos de elementos
7   * con los valores que introduzca el usuario.
8   *
9   * Se usa un <code>acm.io.IODialog</code> para la lectura.
10  */
11 public class LectorDeConjuntosDeString
12 {
13     private acm.io.IODialog io = new acm.io.IODialog();
14     private String prompt =
15         "Continuar introduciendo palabras en el conjunto?";
16
17     /**
18      * Devuelve un nuevo conjunto con las palabras introducidas por el usuario.
19      */
20     public TreeSet<String> nuevoConjuntoDeString()
21     {
22         TreeSet<String> set = new TreeSet<String>();
23         while(io.readBoolean(prompt)){
24             String tmp = io.readLine("Siguiente palabra? ");
25             set.add(tmp);
26         }
27         return set;
28     }
29 }
```

De esta manera, cuando una instancia de esa clase reciba el mensaje `nuevoConjuntoDeString`, procederá a construir un nuevo conjunto con las palabras que le indique el usuario, y ese conjunto (o mejor dicho, la dirección de memoria donde está ese conjunto) será la respuesta devuelta.

Otro aspecto que podríamos cambiar en el programa 21 se refiere al bucle que aparece en la línea 31, que se utiliza para seleccionar y mostrar las primeras palabras del conjunto de palabras introducidas por el usuario. Definiendo una clase de objetos que sean especialistas en seleccionar las primeras palabras de un conjunto de palabras, ese bucle podría cambiarse por:

```
TreeSet<String> seleccion =
        seleccionador.crearConjuntoConMenores(K, palabras);
System.out.println(seleccion.toString());
```

Teniendo en cuenta todo esto, a continuación se muestra la definición de la clase referida:

```
1  package ehu.student.collections;
2
3  import java.util.TreeSet;
4  /**
5   * Calculo de los primeros(menores) y ultimos(mayores) elementos de un conjunto.
6   */
7  public class SeleccionadorDeElementosDeUnConjunto
8  {
9      /**
10      * Devuelve un nuevo conjunto con los K PRIMEROS elementos de "conjunto".
11      * Si no tiene K elementos, el resultado contiene todos sus elementos.
12      */
13     public TreeSet<String> crearConjuntoConMenores(int K,
14                                                    TreeSet<String> conjunto)
15     {
16         TreeSet<String> resultado = new TreeSet<String>();
17         /* Para cada elemento del conjunto dado...*/
18         for (String x: conjunto){
19             resultado.add(x); /* ese elemento se añade al resultado */
20             if (resultado.size() == K){
21                 /* ... ya tiene los K menores del conjunto original */
22                 break; // fin del bucle!
23             }
24         }
25         return resultado;
26     }
27
28     /**
29      * Devuelve un nuevo conjunto con los K ULTIMOS elementos de "conjunto".
30      * Si no tiene K elementos, el resultado contiene todos sus elementos.
31      */
32     public TreeSet<String> crearConjuntoConMayores(int K,
33                                                    TreeSet<String> conjunto)
34     {
35         TreeSet<String> resultado = new TreeSet<String>();
36         /* Para cada elemento del conjunto dado...*/
37         for (String x: conjunto){
38             resultado.add(x); /* ese elemento se añade al resultado */
39             if (resultado.size() > K){
40                 /* ... ya tiene K+1 elementos: hay que eliminar el menor */
41                 String v = resultado.first();// v es la primera(menor)
42                 resultado.remove(v);      // eliminar el elemento v de resultado
43             }
44         }
45         return resultado;
46     }
47 }
```

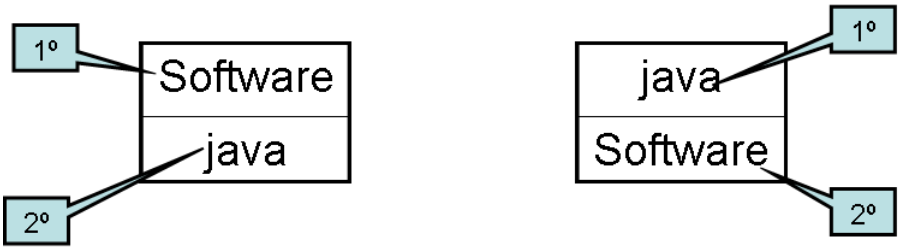
En esta clase, tanto algunos parámetros como el tipo de resultado de los métodos son objetos. Cuando una instancia de esa clase reciba el mensaje `crearConjuntoConMenores`, por ejemplo, recibirá también dos parámetros. El segundo de ellos es el conjunto (si se quiere, la dirección de memoria donde está ese conjunto) cuyos elementos se van a incluir también en el conjunto resultante. A continuación se muestra la nueva versión del programa 21, usando objetos de las clases que se han indicado.

```
1  package ehu.student.collections;
2
3  import java.util.TreeSet;
4
5  /**
6   * Calculo de las primeras palabras de una secuencia dada.
7   *
8   * El usuario introducira tantas palabras como desee.
9   * A continuacion, para cada valor K que indique el usuario,
10  * se escribiran las K primeras palabras distintas de todas ellas.
11  */
12 public class SeleccionDePalabrasDemo {
13
14     public static void main(String[] args)
15     {
16         LectorDeConjuntosDeString lector =
17             new LectorDeConjuntosDeString();
18         TreeSet<String> palabras =
19             lector.nuevoConjuntoDeString();
20
21         SeleccionadorDeElementosDeUnConjunto seleccionador =
22             new SeleccionadorDeElementosDeUnConjunto();
23
24         acm.io.IODialog dialog = new acm.io.IODialog();
25         while(dialog.readBoolean("Continuar examinando el conjunto?")){
26             int K = dialog.readInt("K? ");
27
28             TreeSet<String> seleccion =
29                 //seleccionador.crearConjuntoConMayores(K, palabras);
30                 seleccionador.crearConjuntoConMenores(K, palabras);
31
32             System.out.println(seleccion.toString());
33         }
34         System.exit(0);
35     }
36 }
```

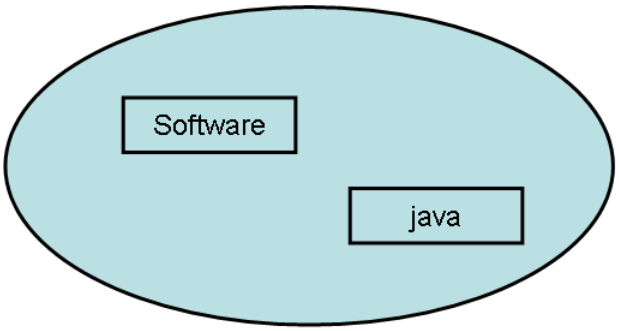
6.2. Listas

Hablando informalmente, en programación diríamos que cosas tales como una cordada de montañeros marchando en hilera, una cola de individuos que esperan ser atendidos en un mostrador, etc, son *secuencias* de personas. Más precisamente, una **secuencia** es una colección de elementos cada uno de los cuales tiene asociado consigo un **índice de posición**, siendo esos índices números enteros consecutivos. Dicho con otras palabras, en una secuencia

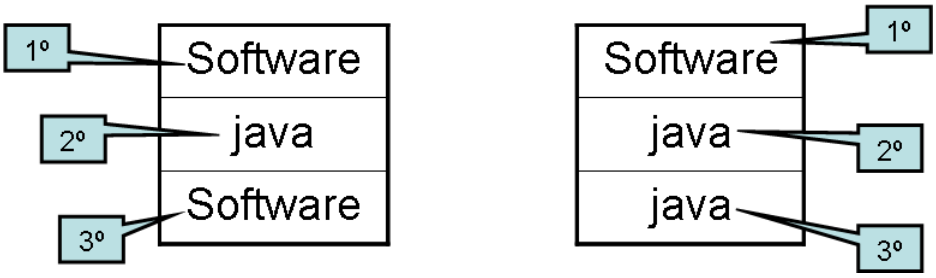
hay un elemento que ocupa la primera posición, otro que ocupa la segunda, otro que ocupa la tercera, y así sucesivamente. Entre las estructuras de datos usadas con más frecuencia en programación se encuentran las que representan secuencias de elementos, por ejemplo, secuencias de números enteros. Esas estructuras de datos se suelen denominar también **listas**. Desde el punto de vista de la programación, entre listas y conjuntos hay dos diferencias importantes. Por una parte, una lista es una colección en la cual cada elemento ocupa una posición distinta. Así pues, con las palabras `Software` y `java` podemos formar dos secuencias distintas, y representarlas gráficamente así:



mientras que con esas mismas dos palabras solamente podemos formar el conjunto siguiente:



Por otra parte, un conjunto es una colección en la cual cada elemento solamente aparece una vez, mientras que en una lista un mismo elemento puede aparecer varias veces en posiciones distintas, como se indica a continuación:



En *java*, un objeto de la clase `ArrayList<Integer>` almacena una secuencia de números enteros; además

las posiciones de esa secuencia se numeran consecutivamente a partir de cero.

Como en el caso de los conjuntos, un `ArrayList<Integer>` recién creado está vacío y no contiene ningún elemento, pero podemos añadirle tantos elementos como deseemos. Por ejemplo, con las instrucciones siguientes:

```
ArrayList<Integer> a = new ArrayList<Integer>();
ArrayList<Integer> b = new ArrayList<Integer>();
ArrayList<Integer> c = new ArrayList<Integer>();
a.add(1); /* se añade el 1 a la lista a */
a.add(1); /* se añade el 1 a la lista a */
a.add(1); /* se añade el 1 a la lista a */
b.add(2); /* se añade el 2 a la lista b */
b.add(2); /* se añade el 2 a la lista b */
c.add(4); /* se añade el 4 a la lista c */
c.add(3); /* se añade el 3 a la lista c */
c.add(2); /* se añade el 2 a la lista c */
c.add(1); /* se añade el 1 a la lista c */
```

se crearán tres objetos `a`, `b` y `c` con el contenido que se indica a continuación:

`a`

Posición	*Elemento*
0	1
1	1
2	1

`b`

Posición	*Elemento*
0	2
1	2

`c`

Posición	*Elemento*
0	4
1	3
2	2
3	1

Como en el caso de los conjuntos, también se pueden manipular listas con otros tipos de elementos. Los nombres de las clases correspondientes son:

Clase	*tipo de elementos*
`ArrayList<Double>`	Valores en coma flotante
`ArrayList<Boolean>`	Valores lógicos (`true`, `false`)
`ArrayList<Character>`	caracteres (Códigos Unicode)
`ArrayList<String>`	ristras de caracteres

Todas esas clases incluyen los mismos métodos. Algunos de ellos se recogen en la figura 6.2; también ahora hay que sustituir $\mathbf{T}$ por el nombre del tipo adecuado a cada caso. Además de añadir y eliminar elementos, hay otras maneras de modificar el contenido de una lista; en particular, se puede:

- insertar un elemento en una posición cualquiera

  Por ejemplo, si la variable `list` se refiere a un objeto `ArrayList<String>` con el contenido siguiente:

Posición	*Elemento*
0	`Intro`
1	`Programming`

  se puede insertar un elemento en las posiciones con índice igual a cero, uno o dos; por ejemplo, a continuación se representa el contenido de ese objeto después de insertar `java` en cada una de esas posiciones:

public boolean `add(T s)`
se añade el elemento `s` al final de la lista
devuelve `true` siempre

public boolean `add(int index, T s)`
se *inserta* el elemento `s` en la posición de la lista indicada por el parámetro `index`
devuelve `true` siempre

public boolean `contains(T s)`
devuelve `true` si la lista contiene el elemento `s`; `false` en caso contrario

public void `clear()`
se eliminan todos los elementos de la lista (queda vacía)

public int `size()`
devuelve el número de elementos de la lista

public T `set(int index, T s)`
sustituye por `s` el elemento que ocupaba la posición indicada por `index`. devuelve el elemento que estaba en la posición referida.

public T `get(int index)`
devuelve el elemento de la lista que se encuentra en la posición indicada por `index`.

public boolean `remove(T s)`
se retira el elemento `s` de la lista de la lista; si el elemento está en varias posiciones de la lista, se retira de la primera de esas posiciones
devuelve `true` si el elemento `s` estaba en la lista

public T `remove(int index)`
retira de la lista que se encontraba en la posición indicada por el parámetro `index`.
devuelve el elemento retirado

Figura 6.2: Algunos de los métodos de las clases `ArrayList<T>`

`lst.add(0, "java");`	`lst.add(1, "java");`	`lst.add(2, "java");`
`java`	`Intro`	`Intro`
`Intro`	`java`	`Programming`
`Programming`	`Programming`	`java`

- eliminar el elemento situado en una posición cualquiera

  Como antes, si la variable `list` se refiere a una lista formada por las palabras `Intro` y `Programming`, se puede eliminar el elemento en las posiciones con índice igual a cero o uno; el contenido de esa lista después de eliminar el elemento en cada una de esas posiciones será:

`lst.remove(0);`	`lst.remove(1);`
`Programming`	`Intro`

- reemplazar el elemento situado en una posición cualquiera

  Volviendo a la lista formada por `Intro` y `Programming`, se puede reemplazar el elemento en las posiciones con índice igual a cero o uno; el contenido de esa lista después de reemplazar el elemento en cada una de esas posiciones será:

`lst.set(0, "java");`	`lst.set(1, "java");`
`java` `Programming`	`Intro` `java`

Naturalmente, cualquier manipulación que se refiera a una posición incorrecta produce el error `IndexOutOfBoundsException` en la ejecución del programa.

6.2.1. Un ejemplo

Programa 22 Inserción en una lista de palabras

```
1  package ehu.student.collections;
2
3  /**
4   * Ejemplo simple de uso de ArrayList<String>.
5   *
6   * Se memorizan las palabras que diga el usuario, eliminando las repeticiones,
7   * y se repiten en el orden en que las dio el usuario.
8   */
9  public class RepetirPalabrasSinRepeticiones {
10
11     public static void main(String[] args){
12         acm.io.IODialog dialog = new acm.io.IODialog();
13
14         /* Las palabras se van añadiendo a una lista */
15         java.util.ArrayList<String> palabras =
16             new java.util.ArrayList<String>();
17
18         while(dialog.readBoolean("Continuar?")){
19             String s = dialog.readLine("Siguiente palabra?");
20
21             if(!palabras.contains(s)){//Si una palabra no esta en la lista...
22                 palabras.add(s);      //... se añade
23             }
24         }
25         System.out.print(palabras.toString());
26         System.exit(0);
27     }
28 }
```

Imaginemos que debemos reproducir todas las palabras que introduzca el usuario, en el mismo orden en que él las haya introducido pero eliminando las palabras repetidas. En este caso, no es adecuado usar un conjunto, ya que se presentarían las palabras en orden alfabético, y no en el orden en que fueron añadidas al conjunto. Sin embargo, usando una lista el problema es muy simple. Cada palabra que introduzca el usuario se añade a la lista, siempre y cuando no se encuentre ya en ella. De esta forma, cuando se hayan consumido todas las palabras que el usuario nos proporcione, esa lista estará formada por las palabras introducidas por el usuario, en el orden en que fueron introducidas pero sin repetir ninguna de ellas. El programa 22 muestra la solución.

6.2.1.1. Instrucciones de iteración y listas

Una forma muy habitual de procesar cada uno de los elementos de una lista, formada por `Strings` por ejemplo, es mediante una instrucción de repetición como la siguiente:

```
//se supone que la variable
//    ArrayList<String> list
//esta ya declarada, y la lista creada
for (int i = 0; i < list.size(); i++){
  String item = list.get(i);
  /*
  * procesamiento de item
  */
  }
```

No obstante, también puede utilizarse la variante de la instrucción **for** mencionada antes:

```
for (String item: list){
  /*
  * procesamiento de item
  */
  }
```

Por ejemplo, en el programa anterior podríamos sustituir la línea:

```
System.out.print(palabras.toString());
```

que se usa para presentar los resultados por lo siguiente:

```
int indiceDePosicion = 0;
for (String s: palabras){
  System.out.println(indiceDePosicion + " > " + s);
  indiceDePosicion++;
}
```

De esta forma se escribíría cada palabra de la lista referida junto con el índice de su posición en la lista.

6.3. Un inciso: las clases wrapper

Por razones de eficiencia, más o menos afortunadas, en *java* se distingue entre valores de ciertos tipos (numéricos, lógicos y caracteres) por un lado, y objetos por otro. Sin embargo, y por otros motivos, la librería de clases standard de **java** incluye clases que corresponden a esos

mismos tipos de valores y que se denominan *clases wrapper*. Por ejemplo, tenemos la clase `Integer`, cuyas instancias representan también números enteros de tipo `int`; análogamente, las instancias de `Double` representan números enteros de tipo `double`. Eso quiere decir que podemos crear objetos `Integer`, y enviarles mensajes:

```
Integer intObject = new Integer(5);    /* crear un objeto Integer */
String tmp = intObject.toString();     /* envio de mensaje a intObject */
```

Sin embargo, si tratamos de escribir algo parecido:

```
int unInt = 5;
String tmp = unInt.toString(); /* unInt NO es un objeto! */
```

resulta que la última línea no es una instrucción de *java*.

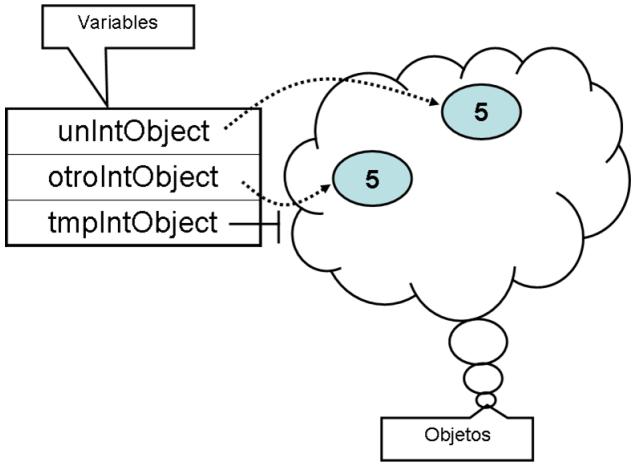


Figura 6.3: Variables y objetos Integer

La duplicidad entre valores de tipo `int` y objetos `Integer` puede resultar confusa en un primer acercamiento a *java*. Por otro lado, un mínimo conocimiento de esas clases es inevitable cuando se quieren usar ciertas clases standard, y muy útiles. Cuando menos, conviene recordar que una variable de tipo `Integer` contiene la referencia de un objeto que, a su vez, contiene un valor entero; por ejemplo, en la figura 6.3 se representa la situación que se producirá al ejecutarse las instrucciones siguientes:

```
Integer unIntObject = new Integer(5);
Integer otroIntObject = new Integer(5);
Integer tmpIntObject = null;
```

Como indica la figura, cuando esas instrucciones se ejecuten se habrán creado dos objetos, y los valores de las variables `unIntObject` y `otroIntObject` serán distintos, aunque el contenido de ambos objetos sea igual. Además, el valor de una variable de tipo `Integer` puede ser la referencia `null`, cosa que no podría ocurrir con una variable de tipo `int`. Por lo demás, y afortunadamente, desde la versión `1.5` de *java*, la distinción entre valores de tipo `int` y objetos `Integer` se ha suavizado. Ahora el compilador se encarga de hacer las conversiones necesarias, y podemos escribir cosas como:

```
Integer intObject = 5; /* Conversion de int a Integer */
int tmp = intObject+1; /* Conversion de Integer a int */
```

Los comentarios anteriores valen también para las clases correspondientes a los demás tipos de valores del lenguaje, cuyos nombres se recogen en la tabla siguiente:

clase	*tipo*
`Byte`	`byte`
`Short`	`short`
`Integer`	`long`
`Long`	`long`
`Character`	`char`
`Double`	`double`
`Float`	`float`

6.3.1. Uso de listas y enteros: peligros

El mecanismo de conversión automática entre valores de tipo `int` y objetos `Integer` puede llevar a cometer errores cuando se usan listas formadas por números enteros. Sustituyendo el símbolo **T** por `Integer` en la figura 6.2, resulta que la clase `ArrayList<Integer>` incluye los métodos siguientes:

public boolean `add(int index, Integer s)`
: se *inserta* el elemento `s` en la posición de la lista indicada por el parámetro `index`
devuelve `true` siempre

public boolean `remove(Integer s)`
: se retira el elemento `s` de la lista de la lista; si el elemento está en varias posiciones de la lista, se retira de la primera de esas posiciones
devuelve `true` si el elemento `s` estaba en la lista

public Integer `remove(int index)`
: retira de la lista que se encontraba en la posición indicada por el parámetro `index`.
devuelve el elemento retirado

Teniendo eso en cuenta, después de unas instrucciones como:

```
ArrayList<Integer> lst = new ArrayList<Integer>();
/*
* Colocar elementos en lst
*/
int x;
Integer y;
/* asignaciones adecuadas */
```

se podría añadir cualquiera de estas dos instrucciones:

`lst.add(x, y);`	`lst.add(y, x);`
insertar `y` en la posición `x`	insertar `x` en la posición `y`

Como puede verse, al insertar un elemento en una lista se corre el riesgo de escribir los argumentos en el orden equivocado: el índice en lugar del valor y viceversa. A causa de la conversión automática entre `Integer` e `int`, Ese error no puede ser detectado por el

compilador, por lo que no queda más remedio que prestar la debida atención cuando se insertan elementos en una lista.

Una situación parecida puede darse cuando se retiran elementos de una lista. En efecto, también podríamos haber añadido cualquiera de las dos instrucciones siguientes:

`lst.remove(x);`	`lst.remove(y);`
retirar el elemento en la posición `x`	retirar el elemento `y`

En este caso, el uso de un argumento de tipo inadecuado, como `int` en lugar de `Integer`, puede llevar a eliminar el elemento situado en una posición particular cuando, en realidad, se pretendía eliminar un elemento particular de la lista, y viceversa.

6.4. Conjuntos y listas de ...

Las diferentes clases de colecciones que se han mencionado en los apartados anteriores estaban formadas por elementos de los diferentes tipos de valores de *java*. Ahora bien, las clases correspondientes a conjuntos y secuencias de enteros se llaman, respectivamente:

`TreeSet<Integer>` `ArrayList<Integer>`

y no, como se podría pensar:

`TreeSet<int>` `ArrayList<int>`

En el nombre de las clases referidas no aparece el nombre del tipo de los elementos, sino el nombre de la clase asociada a ese tipo de valores: `Integer` por `int`, `Double` por `double`, o `Character` por `char`. Eso se debe a que en todas las clases de colecciones de *java*

los elementos que forman una colección son objetos

o más exactamente, referencias de objetos. Una instrucción para añadir un número a una lista de enteros, como

```
miArrayListInteger.add(1);
```

lleva consigo una conversión de un valor de tipo `int` a un objeto de tipo `Integer`; es decir, que se creará un objeto de esta última clase, y ése será el elemento añadido a la colección. De hecho, en *java* podemos crear y manipular conjuntos y listas cuyos elementos sean objetos de cualquier clase. Es decir, que si se han definido clases de objetos llamadas

`CocheConLuz` `CartaBarajaInmutable`

también disponemos de las correspondiente clases de listas:

`ArrayList<CocheConLuz>` `ArrayList<CartaBarajaInmutable>`

Incluso se pueden crear listas cuyos elementos sean listas:

`ArrayList<ArrayList<CocheConLuz>>`
`ArrayList<ArrayList<CartaBarajaInmutable>>`

Programa 23 Ordenación de una lista de cartas

```
1  package ehu.student.collections;
2
3  /**
4   * Ejemplo de listas ArrayList<CartaBarajaInmutable>
5   *
6   * Se crea una lista con tantas cartas como indique el usuario.
7   * Se presentan esas cartas en orden creciente, según palos y figuras.
8   */
9  public class BubbleSort4ArrayListCartaBarajaInmutableTest {
10
11     public static void main(String[] args)
12     {
13         acm.io.IODialog dialog = new acm.io.IODialog();
14
15         /* lista a la cual se añadiran las cartas creadas */
16         java.util.ArrayList<CartaBarajaInmutable> cartas =
17             new java.util.ArrayList<CartaBarajaInmutable>();
18
19         /* lectura y creacion de cartas  */
20         while(dialog.readBoolean("Continuar?"))
21             {
22                 int palo = dialog.readInt("Palo ?");
23                 int figura = dialog.readInt("Figura ?");
24
25                 CartaBarajaInmutable tmp =
26                     new CartaBarajaInmutable(palo, figura);
27
28                 /* se añade la nueva carta a la lista */
29                 cartas.add(tmp);
30             }
31
32         BubbleSort4ArrayListCartaBarajaInmutable ordenador =
33             new BubbleSort4ArrayListCartaBarajaInmutable();
34
35         /* la lista se reorganiza y queda ordenada */
36         ordenador.ordenar(cartas);
37
38         /* presentar las cartas en orden segun palos y figuras */
39         for (CartaBarajaInmutable carta: cartas){
40             System.out.println(carta.toString());
41         }
42         System.exit(0);
43     }
44 }
```

6.4.1. Ejemplo: ordenación de una lista de cartas

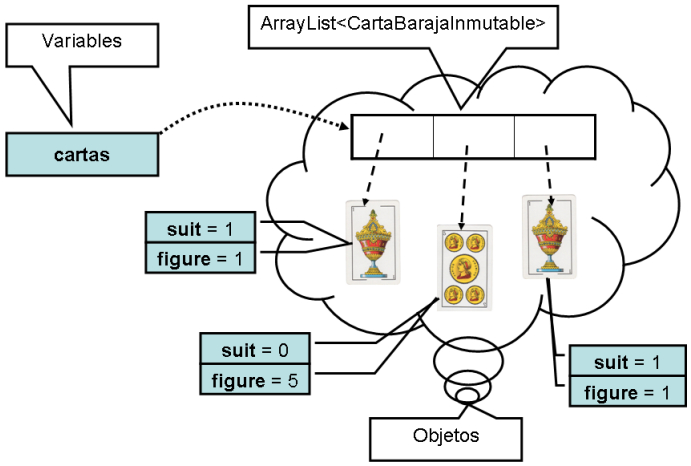
El programa 23 permite al usuario introducir el palo y la figura de tantos naipes como desee; hablamos de naipes de la baraja española, con palos y figuras numerados consecutivamente como sigue:

Palos	
Oros	0
Copas	1
Espadas	2
Bastos	3

Figuras	
As	0
Dos	1
Tres	2
...	...
Caballo	8
Rey	9

Cuando el usuario termine de introducir los datos de los naipes, el programa escribirá el palo y la figura de cada uno de ellos en orden creciente: primero los oros, luego las copas, después las espadas y, finalmente, los bastos.

Para resolver el problema propuesto, el programa 23 crea un nuevo objeto cada vez que el usuario introduce el palo y la figura de una carta: ver línea 26; la clase `CartaBarajaInmutable` ya ha sido definida y no se vuelve a comentar aquí. Los objetos creados de esa manera son añadidos, ver línea 29, a la lista de cartas creada en la línea 17. A continuación, se representa el contenido de esa lista después de que el usuario haya introducido el palo y la figura de tres cartas.

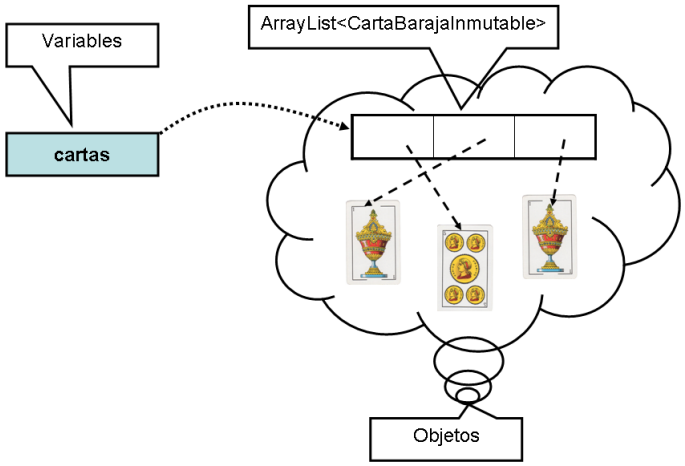


Una vez construida la lista de cartas, conforme a los datos introducidos por el usuario, la solución adoptada reorganiza el contenido de esa lista, de manera que las cartas queden dispuestas conforme a lo dicho arriba. Para ello, en la línea 33 se crea un nuevo objeto, llamado `sorter`, que será el encargado de llevar a cabo esa reorganización. Puesto que lo que necesitamos de ese objeto es la capacidad de ordenar una lista de cartas, parece razonable definir la nueva clase como sigue:

```
public class BubbleSort4ArrayListCartaBarajaInmutable {

  public void sort(ArrayList<CartaBarajaInmutable> lista)
  {
    /* reorganizar la lista de cartas */
    }
  }
```

De esa manera, en el programa 23 podremos usar el llamado `sorter` como se indica en la línea 36. Después de que ese objeto haya completado la tarea solicitada, y si se comporta como se espera, la lista de cartas mostrada arriba habrá quedado reorganizada así:



Obsérvese que después de ejecutarse la línea 36, la variable `cartas` sigue asociada con el mismo objeto. En términos de objetos, la lista es la misma que teníamos antes, solamente que su contenido ha sido reorganizado. Finalmente, para producir los resultados deseados, el programa 23 debe recorrer la lista donde se almacenan las cartas, y escribir la información correspondiente a cada una. Las líneas 39 a 41 muestran cómo hacer ese recorrido usando una instrucción **for** del tipo referido antes.

La clase `BubbleSort4ArrayListCartaBarajaInmutable`, que define el comportamiento de los objetos ordenadores de listas de cartas, se muestra en el programa 24. El método `ordenar` declarado en esa clase define el procedimiento a seguir para ordenar la lista que se reciba como parámetro. Con el bucle de las líneas 20 a 30, se compara cada elemento de la lista con el situado en la posición siguiente; si éste último debe estar situado antes que el primero, entonces se reorganiza la lista situando cada elemento en la posición que le corresponda. Este proceso se repite una y otra vez en el bucle de las líneas 17 a 35, hasta llegar a un punto en el cual ningún elemento de la lista es resituado, momento en el cual la lista habrá quedado ordenada.

Además del método `ordenar`, la clase `BubbleSort4ArrayListCartaBarajaInmutable` incluye también el método `esMenor`. Ese método permite saber si un naipe de la baraja española precede a otro, y podría usarse así:

```
BubbleSort4ArrayListCartaBarajaInmutable sorter =
    new BubbleSort4ArrayListCartaBarajaInmutable();
CartaBarajaInmutable asOros = new CartaBarajaInmutable(0, 0);
CartaBarajaInmutable reyCopas = new CartaBarajaInmutable(1, 9);
boolean b = sorter.esMenor(asOros, reyCopas);
System.out.println(asOros + " precede a " + reyCopas + "?" + b);
```

Como se ve en la línea 25 del programa 24, los métodos de una clase pueden usarse también para definir otros métodos de la misma clase. En términos de mensajes, podríamos decir que el objeto que recibe el mensaje para ordenar una lista, se envía a sí mismo el mensaje para comparar dos cartas. Este mecanismo es muy importante para evitar que los métodos de una clase engorden excesivamente. En definitiva, resultará más fácil comprender un número de métodos con pocas instrucciones cada uno, que otro método que incluya las instrucciones de todos ellos.

Programa 24 Ordenación de una lista de cartas

```
1  package ehu.student.collections;
2
3  import java.util.ArrayList;
4
5  /**
6   * Una clase para ordenar listas de cartas.
7   */
8  public class BubbleSort4ArrayListCartaBarajaInmutable {
9
10     /**
11      * Se reorganiza la "lista" de cartas recolocando sus elementos.
12      * Quedan ordenados segun palos; dentro de cada palo segun figuras
13      */
14     public void ordenar(ArrayList<CartaBarajaInmutable> lista)
15     {
16         boolean hayElementosResituados;
17         do {
18             hayElementosResituados = false;
19             /* comparar cada carta con la siguiente... */
20             for (int i = 0; i < lista.size()-1; i++)
21                 {
22                     CartaBarajaInmutable cartaEnI = lista.get(i);
23                     CartaBarajaInmutable steCarta = lista.get(i+1);
24                     /* ... y si es preciso, intercambiar sus posiciones */
25                     if (esMenor(steCarta, cartaEnI)){
26                         lista.set(i+1, cartaEnI);
27                         lista.set(i, steCarta);
28                         hayElementosResituados = true;
29                     }
30                 }
31             /*
32              * Si hay hayElementosResituados == false, entonces
33              * cada carta esta situada en la posicion que le corresponde
34              */
35         } while (hayElementosResituados);
36     }
37
38     /**
39      * Devuelve "true" si la carta "x" debe ir situada antes que la "y";
40      * "false" en caso contrario.
41      */
42     public boolean esMenor(CartaBarajaInmutable x, CartaBarajaInmutable y)
43     {
44         boolean r =
45             (x.getSuit() < y.getSuit()) ||
46             (x.getSuit() == y.getSuit()
47              && x.getFigure() < y.getFigure());
48         return r;
49     }
50 }
```

6.4.2. Ejemplo: permutaciones de un conjunto de palabras

Programa 25 Permutaciones de un conjunto de palabras

```
1  package ehu.student.collections;
2
3  import java.util.ArrayList;
4  import java.util.TreeSet;
5
6  /**
7   * Ejemplo de uso de ArrayList<ArrayList<String>>
8   *
9   * Se calculan todas las permutaciones posibles de un conjunto de palabras.
10  */
11 public class Permutador4ArrayListStringTest {
12
13     public static void main(String[] args)
14     {
15         /* Para crear el conjunto con las palabras que diga el usuario */
16         LectorDeConjuntosDeString input = new LectorDeConjuntosDeString();
17         //FabricanteDeConjuntos input = new FabricanteDeConjuntos();
18
19         /* El conjunto con las palabras a permutar */
20         TreeSet<String> palabras = input.nuevoConjuntoDeString();
21
22         /* Usado para crear todas las permutaciones posibles */
23         Permutador4ArrayListString permutador =
24             new Permutador4ArrayListString();
25
26         /*
27          * Crear todas las permutaciones de las palabras
28          * Cada permutacion es una lista de palabras.
29          * La totalidad de las permutaciones se almacena
30          * en una lista cuyos elementos son listas de palabras.
31          */
32         ArrayList<ArrayList<String>> permutaciones =
33             permutador.permutar(palabras);
34
35         /* presentar cada permutacion de la lista de permutaciones */
36         for (ArrayList<String> p: permutaciones){
37             System.out.println(p);
38         }
39         System.exit(0);
40     }
41 }
```

El programa 25 permite al usuario introducir tantas palabras como desee. Cuando el usuario termine de introducir palabras, el programa escribirá todas las permutaciones que pueden formarse con las palabras introducidas. Por ejemplo, si el usuario introduce las palabras:

```
C++  java  Python
```

el programa escribirá las $3! = 3 \times 2 = 6$ permutaciones siguientes:

```
 java   Python    C++
Python   java     C++
Python    C++     java
 C++      C++    Python
 C++      java   Python
 C++    Python    java
```

La estructura del programa 25 es relativamente simple. En la línea 16 se crea un nuevo objeto, llamado `input`, que será el encargado de construir un conjunto formado por las palabras que el usuario proporcione: la clase `LectorDeConjuntosDeString` se definió en la página 6.1.3.1. Después de que ese objeto haya completado la tarea que se le solicita en la línea 20, tendremos construido el conjunto`palabras`, formado por las palabras introducidas por el usuario. Dicho sea de paso, si el usuario repite alguna palabra, esa palabra solamente aparecerá en ese conjunto una vez, conforme a la propia definición de lo que es un conjunto.

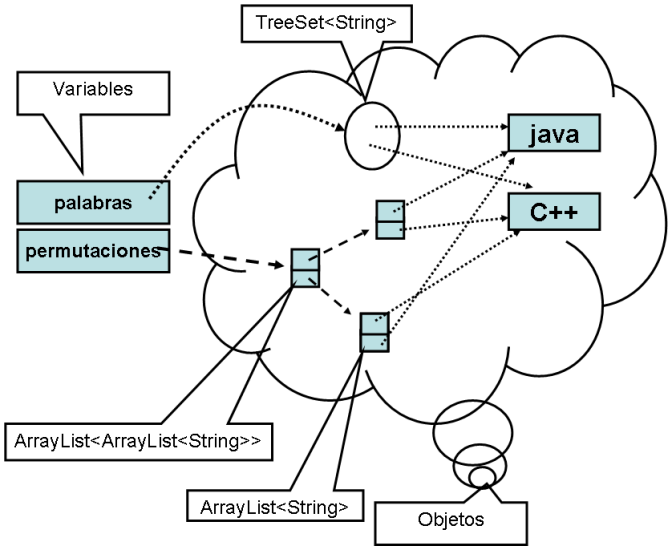
A continuación se deben construir todas las permutaciones que pueden formarse con ese conjunto de palabras. Para ello, se crea un nuevo objeto llamado `permutador` en la línea 24. Lo que deseamos de ese objeto es que, dándole el conjunto de palabras sea capaz de construir todas las permutaciones. Una permutación es una secuencia de palabras, por lo que cabe representar cada permutación mediante un objeto `ArrayList<String>`. En estas condiciones, el resultado que debe producir el `permutador` puede representarse, en principio, mediante un objeto de cualquiera de las clases siguientes:

`ArrayList<ArrayList<String>>` `TreeSet<ArrayList<String>>`

Sin embargo, por razones que comentaremos después, la segunda de las opciones no puede adoptarse. Así pues, la clase que define el comportamiento del `permutador` se definirá, *grosso modo*, como sigue:

```
public class Permutador4ArrayListString
{
    public ArrayList<ArrayList<String>> permutar(TreeSet<String> palabras)
    {
      /*
       * se construye una lista con todas las permutaciones del conjunto
       */
    }
}
```

Así podremos pedir al llamado `permutador` que construya la lista con todas las permutaciones, como se indica en la línea 33. Después de que ese objeto haya completado la tarea solicitada, nos encontraremos en la situación que se representa a continuación, suponiendo que el usuario ha introducido solamente las palabras `java` y `C++`:



La figura anterior pretende indicar que las variables `palabras` y `permutaciones` están asociadas, respectivamente, con dos objetos:

`TreeSet<String>` `ArrayList<ArrayList<String>>`

El primero, que representa el conjunto con las palabras, está formado por dos elementos. El segundo, que representa la lista con las permutaciones tiene también dos elementos que son, a su vez, secuencias formadas por dos palabras. Además, las palabras en sí están representadas por dos únicos objetos, que, por decirlo de alguna manera, son compartidos por el conjunto y por cada una de las dos permutaciones.

En la solución propuesta, la responsabilidad mayor recae en la clase que define el comportamiento del permutador. Un procedimiento eficiente para construir las permutaciones de un conjunto es como sigue. Empezamos seleccionando, y retirando, un elemento cualquiera del conjunto; por ejemplo, si el conjunto en cuestión es:

$$\{\text{C++},\ \text{java},\ \text{Python}\}$$

podríamos empezar eligiendo `C++`. A continuación, tomamos otro elemento del conjunto y lo combinamos con el anterior para construir dos secuencias con dos elementos:

$$\text{C++} \rightarrow \text{java}$$
$$\text{java} \rightarrow \text{C++}$$

A continuación, tomamos otro elemento del conjunto y lo combinamos con cada una de las secuencias anteriores, construyendo secuencias con tres elementos:

$$\text{Python} \rightarrow \text{C++} \rightarrow \text{java}$$
$$\text{C++} \rightarrow \text{Python} \rightarrow \text{java}$$
$$\text{C++} \rightarrow \text{java} \rightarrow \text{Python}$$
$$\text{Python} \rightarrow \text{java} \rightarrow \text{C++}$$
$$\text{java} \rightarrow \text{Python} \rightarrow \text{C++}$$
$$\text{java} \rightarrow \text{C++} \rightarrow \text{Python}$$

Este proceso se continúa hasta consumir todos los elementos del conjunto de palabras original, momento en el cual se habrán construido todas las permutaciones buscadas.

Teniendo en cuenta lo dicho, parece conveniente incluir en la clase `Permutador4ArrayListString` un método para combinar una secuencia de palabras con otra palabra. Por ejemplo, el resultado de combinar la secuencia de palabras

$$\text{C++} \rightarrow \text{java}$$

y la palabra `Python` serán las secuencias de palabras

$$\text{Python} \rightarrow \text{C++} \rightarrow \text{java}$$
$$\text{C++} \rightarrow \text{Python} \rightarrow \text{java}$$
$$\text{C++} \rightarrow \text{java} \rightarrow \text{Python}$$

Está claro que con ese método un permutador deberá producir tantas secuencias de palabras como elementos tenía la secuencia original, mas uno. El resultado del método podría ser por tanto un objeto

`ArrayList<ArrayList<String>>`

Además, cada una de las secuencias de palabras producida será una copia de la secuencia original, con la palabra a combinar insertada en una posición adecuada. Podemos definir el método `combinar` para hacer esa tarea como sigue:

```
public ArrayList<ArrayList<String>> combinar(ArrayList<String> lista,
                                              String s)
{
    ArrayList<ArrayList<String>> listas = new ArrayList<ArrayList<String>>();
    for (int i = 0; i <= lista.size(); i++){
        ArrayList<String> tmp = clonar(lista);
        tmp.add(i, s);   /* INSERTAR "s" en la posicion "i" */
        listas.add(tmp); /* añadir "tmp" a las listas ya calculadas */
    }
    return listas;
}
```

Para clonar una lista de palabras podemos definir un nuevo método:

```
public ArrayList<String> clonar(ArrayList<String> lista){
    ArrayList<String> r = new ArrayList<String>();
    r.addAll(lista); //añadir todos los elementos de "lista"
    return r;
}
```

La descripción informal hecha antes sugiere que una parte importante de la tarea consiste en construir nuevas secuencias de palabras extendiendo las secuencias construidas previamente. Esa extensión se hace combinando cada una de éstas secuencias con una nueva palabra. Así que conviene definir también el método `extender`:

```
public ArrayList<ArrayList<String>> extender(ArrayList<ArrayList<String>> listas,
                                             String p)
{
    ArrayList<ArrayList<String>> ext = new ArrayList<ArrayList<String>>();
    for (ArrayList<String> l: listas){
        ArrayList<ArrayList<String>> rs = combinar(l, p);
        ext.addAll(rs); /* añadir todas las listas de palabras de "rs" */
    }
    return ext;
}
```

En estas condiciones, la definición del método `permutar`, que era el que nos interesaba originalmente, queda con un grado de complejidad razonable. Se deben extender las secuencias de palabras contenidas en una lista, con una de las palabras del conjunto original. Las secuencias obtenidas de esta manera se vuelven a extender con otra palabra, y así sucesivamente, hasta haber consumido todas las palabras disponibles. A grandes rasgos, el método `permute` se definirá como sigue:

```
public ArrayList<ArrayList<String>> permutar(TreeSet<String> palabras)
{
    ArrayList<ArrayList<String>> perm;
    /* establecer valor inicial adecuado de perm */

    for (String s: palabras){
        buff = extender(buff, s);
    }

    return buff;
}
```

En cada iteración, se toma una palabra y se usa para extender las secuencias de palabras contenidas en la lista `perm`. La nueva lista de secuencias de palabras, que se obtiene como resultado del método `extender`, es asignada a la variable `perm`, de manera que esas secuencias sean extendidas en la iteración siguiente con otra palabra, y así sucesivamente. Solo queda por decidir cuáles son las secuencias de palabras con las cuales se debe empezar. La opción más sencilla parece ser empezar con una lista que contiene una única secuencia: a saber, la secuencia vacía. El programa siguiente muestra la definición completa de la clase.

```
1  package ehu.student.collections;
2
3  import java.util.ArrayList;
4  import java.util.TreeSet;
5
6  /**
7   * Una clase para crear todas las permutaciones de un conjunto de palabras.
8   */
9  public class Permutador4ArrayListString {
10
11     /**
12      * Devuelve una NUEVA lista con las mismas palabras que "lista"
13      */
14     public ArrayList<String> clonar(ArrayList<String> lista){
15         ArrayList<String> r = new ArrayList<String>();
```

```
16          r.addAll(lista); //añadir todos los elementos de "lista"
17          return r;
18      }
19
20      /**
21       * Devuelve todas las listas de palabras que se pueden formar intercalando
22       * la palabra "p" en cada una de las posiciones de una "lista" de palabras
23       */
24      public ArrayList<ArrayList<String>> combinar(ArrayList<String> lista,
25                                                   String s)
26      {
27          ArrayList<ArrayList<String>> listas = new ArrayList<ArrayList<String>>();
28          for (int i = 0; i <= lista.size(); i++){
29              ArrayList<String> tmp = clonar(lista);
30              tmp.add(i, s);   /* INSERTAR "s" en la posicion "i" */
31              listas.add(tmp); /* añadir "tmp" a las listas ya calculadas */
32          }
33          return listas;
34      }
35
36      /**
37       * Devuelve todas las listas de palabras que se pueden formar intercalando
38       * la palabra "p" en cada una de las posiciones de cada una de las
39       * listas de palabras incluidas en "listas"
40       */
41      public ArrayList<ArrayList<String>> extender(ArrayList<ArrayList<String>> listas,
42                                                   String p)
43      {
44          ArrayList<ArrayList<String>> ext = new ArrayList<ArrayList<String>>();
45          for (ArrayList<String> l: listas){
46              ArrayList<ArrayList<String>> rs = combinar(l, p);
47              ext.addAll(rs); /* añadir todas las listas de palabras de "rs" */
48          }
49          return ext;
50      }
51
52      /**
53       * Devuelve todas las listas de palabras que se pueden formar permutando
54       * las palabras de un conjunto.
55       */
56      public ArrayList<ArrayList<String>> permutar(TreeSet<String> palabras)
57      {
58          ArrayList<ArrayList<String>> perm = new ArrayList<ArrayList<String>>();
59
60          perm.add(new ArrayList<String>());
61          /*
62           * Las listas de palabras de "perm" se extienden con una palabra;
63           * las nuevas listas se extienden con otra palabra,
64           * y las nuevas con otra, y asi sucesivamente.
65           * Las listas de palabras obtenidas finalmente son las permutaciones
66           * del conjunto de palabras.
67           */
68          for (String s: palabras){
69              perm = extender(perm, s);
```

```
70          }
71          return perm;
72      }
73
74      /**
75       * Devuelve un nuevo String con todas las listas de palabras de "listas",
76       * incluyendo un separador de línea entre cada dos listas.
77       */
78      public String toString(ArrayList<ArrayList<String>> listas){
79          String buff = "";
80          for (ArrayList<String> x: listas){
81              buff = buff + x + "\n";
82          }
83          return buff;
84      }
85  }
```

6.5. Aplicaciones (mappings)

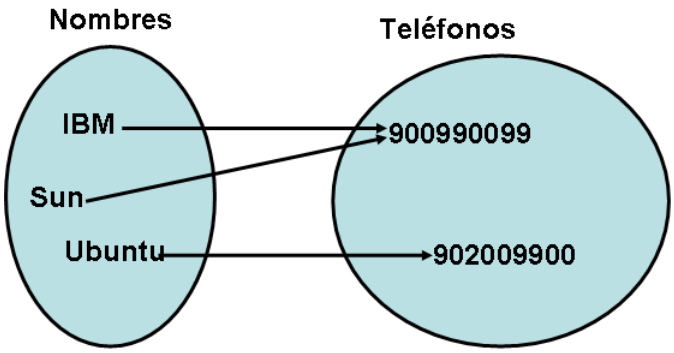


Figura 6.4: Una agenda de teléfonos: representación gráfica

Imaginemos una aplicación en la cual sea necesario almacenar en la memoria del computador los teléfonos de un grupo de individuos e instituciones. Además, en contra de lo que suele ocurrir, supongamos que cada entidad, persona o institución, solamente tiene un número de teléfono. Seguramente, la estructura de datos construida para almacenar esa información se usará para hacer consultas como, por ejemplo, obtener el teléfono de alguien. Posiblemente, también será preciso modificar el contenido de esa estructura, bien para incorporar nuevas personas con sus correspondientes números de teléfono, o bien para reemplazar el número de teléfono registrado de una persona por otro número. Olvidando los detalles referentes a cómo se almacena esa información, podemos representar gráficamente la estructura de datos referida como se muestra en la figura 5.3. Cosas como esa son conocidas en matemáticas como **aplicaciones**, concepto que también es muy útil en programación, donde es más frecuente usar el término **mapping**.

En ocasiones un programa debe almacenar en la memoria del computador pares de elementos que representan las correspondencias existentes entre los elementos de un conjunto, las **claves**, y los elementos de otro conjunto, los **valores**. En otras palabras, el programa debe memorizar qué *valor* le corresponde a cada **clave**. Por lo general, las estructuras de datos usadas para almacenar esa información no permanecen inalteradas durante la ejecución del programa, ya que suele ser necesario cambiar el dato asociado a una clave, o eliminar la pareja formada por una clave y el valor correspondiente. En cualquier caso, su aspecto más relevante es que son usadas, sobre todo, para saber qué dato le corresponde a una clave dada. En *java* es posible crear objetos que representan *aplicaciones* como las referidas. Así, un objeto de la clase

`TreeMap<String, Integer>`

representa una aplicación en la cual las claves son ristras de caracteres y los valores son números enteros, y se crea de la manera usual:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
```

Como los conjuntos y las listas, los objetos de esa clase se crean vacíos y, posteriormente, se les añaden elementos; en este caso, pares formados por una clave y un valor. Por ejemplo, después de ejecutarse las instrucciones siguientes:

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>;
Integer tfno4IBM = 900990099;
Integer tfno4Ubuntu = 902009900;
map.put("IBM", tfno4IBM); /* IBM => tfno4IBM */
map.put("Sun", tfno4IBM); /* Sun => tfno4IBM  */
map.put("Ubuntu", tfno4Ubuntu); /* Ubuntu => tfno4Ubuntu */
```

el contenido de `map` será el representado en la figura 5.3. Aparte de para almacenar pares de elementos, el uso más relevante de una aplicación es la búsqueda del valor correspondiente a una clave. Así, después de las instrucciones anteriores, la aplicación `map` podría usarse como sigue:

```
Integer valor4IBM = map.get("IBM");
Integer valor4Sun = map.get("Sun");
```

Seguramente no será una sorpresa descubrir que podemos crear aplicaciones en las cuales tanto claves como valores sean de otros tipos. En efecto, además de la clase anterior, también existen clases llamadas:

$$\texttt{TreeMap}< \mathbf{K}, \mathbf{V} >$$

donde los símbolos $\mathbf{K}$ y $\mathbf{V}$ deben sustituirse, respectivamente, por el nombre del tipo correspondiente a *claves* y *valores*:

```
TreeMap<String, Double>   ...   TreeMap<String, String>
                ...
TreeMap<Integer, Double>  ...   TreeMap<Integer, String>
```

Todas esas clases incluyen métodos similares, algunos de los cuales se recogen a continuación:

public V `put(K key, V value)`
la clave `key` queda asociada con `value`
devuelve el valor que correspondía a esa clave; el resultado es `null` si la clave no se encontraba en el conjunto de claves

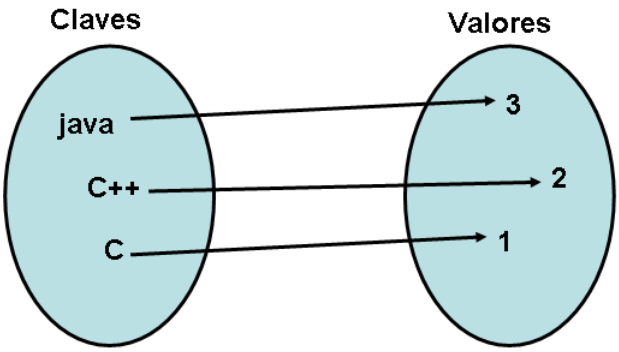
public V `remove(K key)`
eliminar la clave `key` del conjunto de claves
devuelve el valor que correspondía a esa clave; la respuesta es `null` si esa clave no se encontraba en el conjunto de claves

public V `get(K key)`
devuelve el valor que le corresponde a la clave `key`; el resultado es `null` si la clave no se encontraba en el conjunto de claves

6.5.1. Ejemplo: frecuencias de palabras

El programa 22 permite al usuario introducir tantas palabras como desee, y, a continuación, reproduce cada una de las palabras introducidas, evitando repeticiones, indicando cuántas veces ha aparecido cada una de ellas. La solución propuesta utiliza una aplicación para registrar las palabras distintas y cuántas veces aparece cada una de ellas. En la figura siguiente se representa el estado de ese objeto cuando el usuario ha introducido las palabras

```
C  C++  java  java  C++ java
```



Está claro que, en esa aplicación, las claves son objetos `String` y los valores objetos `Integer`, por lo que el programa crea en la línea 21 un objeto

```
TreeMap<String, Integer>
```

El uso que se hace de esa aplicación en el programa es bastante sencillo. Cada vez que el usuario introduce una palabra se le pide a la aplicación el número de veces que esa palabra ha sido introducida con anterioridad, haciendo una llamada al método `get`, en la línea 27. Y aquí aparece un aspecto a tener en cuenta cuando se utilizan aplicaciones. Si es la primera vez que el usuario introduce una palabra, esa palabra no estará registrada en la aplicación y el resultado de ese método será la referencia `null`: recuérdese que todas las colecciones de *java* contienen, en definitiva, referencias (burdamente, direcciones de memoria) de objetos.

Resumiendo, cuando el resultado del método `get` es igual a `null`, el número de veces que la palabra ha aparecido con anterioridad es cero; esto se refleja en la instrucción condicional entre las líneas 32 y 35. Después de incrementar el número de veces que la palabra ha aparecido, ya solo queda cambiar el número que esa palabra tiene asociado en la aplicación; eso se hace en la línea 40 mediante el método `put`. Finalmente, solo queda por escribir el texto que describe el contenido de la aplicación, mediante el método `toString`.

6.5.2. Instrucciones de iteración y aplicaciones

Como en el caso de conjuntos y listas, a menudo queremos realizar ciertas acciones con cada uno de los elementos de una aplicación. Ese tipo de recorridos puede hacerse también con una variante de la instrucción **for**. En este sentido, hay que considerar que una aplicación está formada por pares formados por una clave y un valor. Por ejemplo, la representada en la figura 5.3 puede verse también como el conjunto de pares

(IBM, 900990099) (Sun, 900990099) (Ubuntu, 902009900)

Esto hace que las instrucciones para recorrer una aplicación sean algo más complejas que las usadas antes. Con una instrucción como la siguiente:

```
for (Map.Entry<String, Integer> entry: miMap.entrySet()){
  String clave = entry.getKey();
  Integer valor = entry.getValue();

  /*
   * bloque de instrucciones
   */
}
```

se repite el bloque de instrucciones indicado *para cada uno de los pares* de la aplicación referida. En cada iteración, a las variables `clave` y `valor` se les asigna la clave y el valor del par procesado en esa iteración. Así pues, si queremos cambiar la forma de presentar los resultados del programa 22, podemos sustituir la línea 42, por:

```
for (Map.Entry<String, Integer> entry: map.entrySet()){
    String s = entry.getKey();
    Integer v = entry.getValue();

    System.out.println(s + "  => " + v);
}
```

De manera más general, las instrucciones para el recorrido de esa aplicación de tipo `TreeMap`$< \mathbf{K}, \mathbf{V} >$ tienen el formato siguiente:

```
for ( Map.Entry<K,V> [v_e]: [M].entrySet()){
    K [v_k] = [v_e].getKey();
    V [v_v] = [v_e].getValue();
    /*
     * bloque de instrucciones
     */
}
```

donde los símbolos $\mathbf{v}_e$, $\mathbf{v}_k$, y $\mathbf{v}_v$ deben sustituirse por nombres de variables no declaradas ya, y $\mathbf{M}$ debe sustituirse por el nombre de una variable ya declarada que sea un `TreeMap`$< \mathbf{K}, \mathbf{V} >$. Una instrucción así da lugar a la repetición de su bloque de instrucciones, una vez por cada par *(clave, valor)* de la aplicación $\mathbf{M}$, empezando por el que tenga menor clave. En cada iteración, a las variables $\mathbf{v}_k$, y $\mathbf{v}_v$ se les asigna la clave y el valor del par correspondiente a esa iteración.

6.5.3. Ejemplo: agrupación de palabras por longitudes

Supongamos que debemos reproducir todas las palabras introducidass por el usuario, evitando repeticiones, y ordenadas alfabéticamente, agrupadas según longitudes crecientes. Así, si el usuario introduce las palabras siguientes:

```
C++  java  ruby  python  groovy  pascal  clu
```

el programa producirá un resultado como el siguiente:

```
[3]  =>          [C++, clu]
[4]  =>          [java, ruby]
[6]  =>  [groovy, pascal, python]
```

Si bien el problema puede resolverse usando una simple lista de palabras, usando una aplicación algo más compleja, el esfuerzo a realizar será menor. Hechando un vistazo a los resultados que debe producir el programa, es relativamente claro que esa información puede almacenarse en una aplicación en la cual las claves sean las longitudes de las palabras, y los valores conjuntos de palabras. Es decir, esa información puede representarse mediante un objeto

```
TreeMap<Integer, TreeSet<String>
```

En el programa 27 se ve la solución propuesta. Para cada palabra introducida por el usuario, se le pide a la aplicación `map` el conjunto de palabras que tienen la misma longitud que la palabra referida. Análogamente a lo visto en el ejemplo anterior, cuando el usuario introduce una palabra con una longitud distinta a las que introdujo antes, el resultado obtenido será la referencia `null`. En ese caso, será necesario crear un nuevo conjunto y modificar la aplicación `map`, de manera que ese nuevo conjunto quede asociado a la longitud de la palabra. Esto se hace en la línea 39, usando el método `put`. En cualquier caso, ya sea una palabra con una longitud nueva o una palabra con una longitud ya registrada, la palabra se añade al conjunto en cuestión.

6.6. Colecciones de . . . : limitaciones

Las clases que se han presentado para el uso de diferentes estructuras de datos tienen algunas limitaciones, cuando los elementos que forman una colección no se corresponden con los tipos primitivos del lenguaje. Por ejemplo, el siguiente programa crea un conjunto cuyos elementos son cartas de la baraja española:

```java
package ehu.student.collections.limitaciones;

import java.util.TreeSet;

import ehu.student.collections.CartaBarajaInmutable;

/**
 * Limitaciones en el uso de TreeSet<CartaBarajaInmutable>
 */
public class ErrorUsoTreeSetDeCartaBarajaInmutable {

    public static void main(String[] args)
    {
        TreeSet<CartaBarajaInmutable> conjunto =
            new TreeSet<CartaBarajaInmutable>();

        /* añadir elementos */
        CartaBarajaInmutable unaCarta = new CartaBarajaInmutable(0, 0);
        conjunto.add(unaCarta);
        conjunto.add(unaCarta); /* ERROR! */
    }
}
```

Aparentemente, en ese programa no hay nada peculiar. Sin embargo, al ejecutarlo se producirá un error al tratar de añadir el segundo elemento. Otro tanto ocurre cuando usamos una aplicación cuyas claves sean también cartas de la baraja española, como en el siguiente programa:

```java
package ehu.student.collections.limitaciones;

import java.util.TreeMap;

import ehu.student.collections.CartaBarajaInmutable;

/**
 * Limitaciones en el uso de TreeMap<CartaBarajaInmutable, ...>
 */
public class ErrorUsoTreeMapDeCartaBarajaInmutable {

    public static void main(String[] args)
    {
        TreeMap<CartaBarajaInmutable, String> map =
            new TreeMap<CartaBarajaInmutable, String>();

        /* añadir elementos */
        CartaBarajaInmutable unaCarta = new CartaBarajaInmutable(0, 0);
        map.put(unaCarta, "una carta");
        map.put(unaCarta, "otra carta"); /* ERROR! */
    }
}
```

Lo que ocurre es que los elementos que forman parte o bien de un conjunto, representado por un `TreeSet`$< \mathbf{T} >$, o bien de una aplicación, representada por un `TreeMap`$< \mathbf{T}, \mathbf{K} >$,

tienen que poseer ciertas características. En concreto, la clase correspondiente tiene que incluir un par de métodos cuyos detalles no vamos a comentar ahora, y que no están declarados, por ejemplo, en la clase `CartaBarajaInmutable`.

Por lo que se refiere al uso de listas, representadas por objetos `ArrayList`$< \mathbf{T} >$, también se dan unas limitaciones parecidas. Considérese, por ejemplo, el siguiente programa:

```
1  package ehu.student.collections.limitaciones;
2
3  import java.util.ArrayList;
4
5  import ehu.student.collections.CartaBarajaInmutable;
6
7  /**
8   * Limitaciones en el uso de ArrayList<CartaBarajaInmutable>
9   */
10 public class ErrorUsoArrayListDeCartaBarajaInmutable {
11
12     public static void main(String[] args)
13     {
14         ArrayList<CartaBarajaInmutable> lista =
15             new ArrayList<CartaBarajaInmutable>();
16
17         /* añadir elementos */
18         CartaBarajaInmutable unaCarta = new CartaBarajaInmutable(0, 0);
19         lista.add(unaCarta);
20         lista.add(unaCarta);
21
22         /* buscar elementos */
23         boolean estaUna = lista.contains(unaCarta);
24
25         CartaBarajaInmutable otraCartaIgual = new CartaBarajaInmutable(0, 0);
26         boolean estaOtra =
27             lista.contains(otraCartaIgual); /* Resultado incorrecto? */
28
29         String msg =
30             unaCarta + " esta? " + estaUna + "\n" +
31             otraCartaIgual + " esta? " + estaOtra;
32         System.out.println(msg);
33     }
34 }
```

Como puede verse, se crea una lista y se añade a ella una carta, que representa el as de oros. A continuación, se pregunta a la lista si contiene una carta, que representa el as de oros, y otra carta, que también representa el as de oros. En este caso, la ejecución del programa no se aborta, sino que se producen los resultados siguientes:

```
[Palo: 0; Figura: 0] esta? true
[Palo: 0; Figura: 0] esta? false
```

Ahora bien, la corrección de esos resultados es discutible pero justificable. Como se ha dicho antes

los elementos que forman una colección son objetos

Cuando a la lista se le pregunta si contiene una carta, la respuesta es `true` si en la colección se encuentra la carta en cuestión, y no basta con que se encuentre *otro* objeto similar. Este comportamiento es algo distinto del que exhiben las listas formadas objetos correspondientes a los tipos primitivos del lenguaje (`Integer`, `String`, etc). Para que una lista, representada por un `ArrayList`$< \mathbf{T} >$, responda que contiene un elemento cuando en la colección se encuentra el objeto en cuestión, o bien otro que sea igual, en algún sentido, a él, la clase correspondiente debe declarar un método particular, que no está declarado en la clase `CartaBarajaInmutable`. Más adelante se verá cómo resolver estas cuestiones, y su justificación.

Programa 26 Contando frecuencias de palabras

```
1  package ehu.student.collections;
2
3  import java.util.TreeMap;
4
5  /**
6   * Ejemplo de uso de TreeMap<String, Integer>.
7   *
8   * El usuario introduce tantas palabras como desee.
9   * El programa presenta esas palabras, en orden alfabetico
10  * y sin repeticiones, junto con la frecuencia de cada una:
11  * cuantas veces ha sido leida.
12  */
13 public class ContadorFrecuenciasDePalabras {
14
15     public static void main(String[] args)
16     {
17         acm.io.IODialog dialog = new acm.io.IODialog();
18
19         /* Para memorizar cada palabra y cuantas veces aparece */
20         TreeMap<String, Integer> map =
21             new TreeMap<String, Integer>();
22
23         while(dialog.readBoolean("Continuar?")){
24             String w = dialog.readLine("Sigte palabra ?");
25
26             /* Evaluar cuantas veces ha sido leida antes */
27             Integer numeroDeApariciones = map.get(w);
28
29             /*
30              * numeroDeApariciones es un objeto:
31              *      podria ser "null"!... */
32             if (numeroDeApariciones == null){
33                 /* ...si la palabra se lee por primera vez! */
34                 numeroDeApariciones = 0;
35             }
36             /* sea lo que sea, se cuanta una vez mas */
37             numeroDeApariciones = numeroDeApariciones+1;
38
39             /* cambiar el valor asociado a la palabra */
40             map.put(w, numeroDeApariciones);
41         }
42         System.out.println(map.toString());
43         System.exit(0);
44     }
45 }
```

Programa 27 Agrupación de palabras por longitudes

```
1  package ehu.student.collections;
2
3  import java.util.TreeMap;
4  import java.util.TreeSet;
5
6  /**
7   * Ejemplo de uso de TreeMap<Integer, TreeSet<String>>
8   *
9   * El usuario introduce tantas palabras como desee.
10  * El programa presenta esas palabras, en orden alfabetico,
11  * sin repeticiones, agrupadas según longitudes crecientes.
12  */
13 public class AgrupadorPalabrasPorLongitud {
14
15     public static void main(String[] args){
16         acm.io.IODialog dialog = new acm.io.IODialog();
17
18         /*
19          * Logitudes de las palabras leidas,
20          * junto con las palabras leidas de cada longitud
21          */
22         TreeMap<Integer, TreeSet<String>> map =
23             new TreeMap<Integer, TreeSet<String>>();
24
25         while(dialog.readBoolean("Continuar?")){
26             String w = dialog.readLine("Sigte palabra ?");
27             int longitud = w.length();
28
29             /*
30              * Ver cual es el conjunto de palabras de esa longitud
31              * leidas hasta ahora:
32              *     puede ser "null"...
33              */
34             TreeSet<String> set = map.get(longitud);
35             if (set == null){
36                 /* ... si no se ha leido ninguna palabra con esa "longitud" */
37                 set = new TreeSet<String>();
38                 /* el conjunto de palabras correspondiente a esa "longitud" */
39                 map.put(longitud, set);
40             }
41             /*
42              * en todo caso, la palabra se añade al conjunto
43              * de palabras con esa "longitud"
44              */
45             set.add(w);
46         }
47         System.out.println(map.toString());
48         System.exit(0);
49     }
50 }
```