

Resumen PAV (8)

Este resumen ha sido elaborado para ser leído después de haber asistido a clase: pretende fijar y profundizar en los conceptos ya explicados.

La librería de colecciones de Java

Introducción

La librería `java.util` contiene un juego bastante completo y sofisticado de clases que permiten almacenar y manejar objetos. Las principales clases son `List`, `Set`, y `Map`. Estas clases se conocen también como colecciones (*collection classes*).

Hasta la versión 1.4 de Java, cuando se almacenaba un objeto en una colección se perdía la información de la clase. Es decir, el objeto era almacenado como un objeto de tipo `Object`. Esto obligaba al programador a usar un *downcast* al recuperar el objeto de la colección.

```
ArrayList lista = new ArrayList();

// añadimos un objeto a la lista
lista.add(new Persona("Fulano"));

// recuperamos el objeto de la lista
Persona p = (Persona) lista.get(0); // downcast
```

A partir de la versión 1.5, Java introduce el uso de *generics* en su librería de colecciones, lo que simplifica la sintaxis de programación y evita el uso innecesario de *downcasts*:

```
ArrayList<Persona> lista = new ArrayList<Persona>();

lista.add(new Persona("Fulano"));

// ya no es necesario el downcast
Persona p = lista.get(0);
```

En general, siempre que la librería de colecciones sea suficiente para resolver los problemas que tenemos entre manos, es preferible usar la librería a desarrollar nuestras propias colecciones. (En el *Collections Framework Overview* de la documentación de Java se dan algunos motivos).

Estructura de la librería

(No vamos a detallar “toda” la librería de colecciones, sólo las clases principales)

La librería de colecciones define nueve interfaces:

- La interfaz más básica es `Collection`.
- Cinco interfaces extienden a `Collection`: `Set`, `List`, `SortedSet`, `Queue` y `BlockingQueue`.
- Otras tres interfaces no extienden a `Collection`: `Map`, `SortedMap`, `ConcurrentMap`. Estas tres interfaces no son propiamente colecciones, pero definen métodos que permiten ver los objetos que almacenan como si fueran una colección, y manipularlos en cierto modo como si fueran una colección.

A partir de estas interfaces se implementan una serie de clases: primero unas clases abstractas, y luego unas clases que extienden a las abstractas y que pueden ser usadas directamente para almacenar colecciones de objetos. En la tabla siguiente¹ se resumen las interfaces que nos interesan y las clases que las implementan (no se muestran las clases abstractas):

		Clases (Implementación)			
		Tabla hash (hash table)	Arreglo redimensionable (resizable array)	Árbol balanceado (balanced tree)	Lista enlazada (linked list)
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Algunos métodos de Collection

<code>void add(E o)</code> ²	añade o a la colección (opcional)
<code>boolean remove(Object o)</code>	devuelve true si se pudo eliminar el objeto o (opcional)
<code>int size()</code>	devuelve el número de elementos de la colección
<code>boolean contains(Object o)</code>	devuelve true si la colección contiene o
<code>void clear()</code>	elimina todos los objetos de la colección (opcional)
<code>boolean equals(Object o)</code>	devuelve true si el objeto o y la colección son iguales
<code>boolean isEmpty()</code>	devuelve true si la colección no tiene elementos
<code>Iterator<E> iterator()</code>	devuelve un objeto iterador para la la colección
<code>Object[] toArray()</code>	devuelve un array de Objects con todos los objetos de la colección.
<code><T> T[] toArray(T[] a)</code>	devuelve un array de objetos de tipo T con todos los objetos de la colección

Los métodos de la interfaz que está marcados como opcionales pueden ser o no implementados por las clases que implementen la interfaz Collection³. Es necesario ver la documentación de cada clase concreta para saber cuáles de los métodos implementa. Si usamos un método opcional que no está implementado, se produce una excepción (UnsupportedOperationException).

Al final de resumen se incluyen los métodos de las principales clases.

Iterators

Cualquiera que sea la clase concreta de colección que usemos, necesitamos un modo de insertar y luego recuperar los elementos de la colección.

¹ Tomada de la documentación de Java.

² En vez de usar la notación `add(Object o)` vamos a usar la notación `add(E o)` cuando se pueda especificar el tipo de dato en vez de Object, donde E indica el tipo de dato que almacena la colección si se usa un *generic* al declarar la variable. Es decir, o es de tipo E. Nótese que algunos métodos siguen usando un parámetro Object aunque se use *generics* (por ejemplo, `equals`, porque `equals` es un método heredado y ya definido con parámetro Object o en la clase Object).

³ Cabe preguntarse qué tan acertado es diseñar una librería de clases de este modo, cuando se supone que al declarar una interfaz estamos justamente declarando qué metodos tienen que tener necesariamente las clases que la implementan.

Varias de las clases implementan los métodos `add()` y `get()`. Sin embargo, puede haber situaciones en las que nos interese diseñar las clases que manipulan los objetos de tal modo que no dependan del tipo de colección concreto que se use para almacenarlos. Por ejemplo, quizá inicialmente podemos usar un `ArrayList`, porque es más fácil de utilizar, pero luego, conforme profundizamos en el problema, nos damos cuenta de que realmente no necesitamos el acceso aleatorio, y en cambio es crucial la velocidad de inserción y remoción de los objetos, y entonces decidimos cambiar el `ArrayList` por un `Set` (que sólo tiene acceso secuencial).

Los iteradores⁴ (*iterators*) nos permiten lograr ese nivel de abstracción. Un *iterator* es un objeto cuya función es moverse dentro de la colección y seleccionar cada objeto de la colección sin que el programador tenga que conocer ni tenga que preocuparse de cómo es la estructura de la colección o qué métodos implementa.

Además un *iterator* es lo que se llama un *lightweight object* (objeto liviano): crear un iterador no es costoso en términos de memoria ni ciclos del procesador. Por eso tienen algunas restricciones: por ejemplo, la mayoría de iteradores sólo puede recorrer la colección en una dirección.

La interfaz `Iterator` de `java.util` está definida así:

```
public interface Iterator {
    boolean hasNext(); // true si quedan elementos por iterar
    E next();           // devuelve el siguiente elemento de la colección
    void remove();      // elimina el elemento que acaba de
                        // ser devuelto por next()
}
```

Es importante que durante la iteración la lista no sea modificada (salvo con el método `remove()` del iterador). Si Java detecta alguna modificación, se producirá una excepción.

Conforme se expliquen las clases que implementan colecciones veremos algunos ejemplos de iteradores.

Listas

Las clases que implementan la interfaz `List` permiten almacenar una colección ordenada⁵ de objetos. Algunos métodos de `List` son `add(E elemento)`, `add(int pos, E elemento)`, `get(int pos)`, etc.

ArrayList

`ArrayList` permite crear arreglos redimensionables y acceder rápidamente a los elementos individuales usando un índice entero (como los arreglos, la numeración empieza en cero). Permite elementos duplicados⁶ y `null`.

Implementa todos los métodos de la interfaz `List`. Los métodos `size`, `isEmpty`, `get`, `set` e `iterator` funcionan en tiempo constante (independientemente del número de elementos). Las operaciones de inserción y remoción de objetos no son tan eficientes.

⁴ *Iterator* es un pattern.

⁵ Ordenada no en el sentido de *sort*, sino en el sentido de *ordered*. Es decir, el orden en que se añaden los objetos a la lista es respetado.

⁶ En el contexto de colecciones, `e1` y `e2` se consideran duplicados si `e1.equals(e2) == true`. (Lo que quiere decir que no necesariamente todos los atributos de los dos objetos son idénticos: sólo los que compara el método `equals`).

Para poder mostrar un ejemplo, necesitamos poder almacenar algunos objetos. Para eso, definamos una clase `Persona`, que tenga como atributos el nombre de la persona y su edad, implementa la interfaz `Comparable`. Dos objetos `Persona` son iguales si tiene el mismo nombre y edad:

```
Personal.java
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;

    Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }

    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setEdad(int edad) { this.edad = edad; }

    public String toString() {
        return nombre + " (" + edad + " años)";
    }

    public boolean equals(Object p) {
        if ((edad == ((Persona) p).edad) &&
            (nombre.equals(((Persona) p).nombre)))
            return true;
        else
            return false;
    }

    public int compareTo(Persona p) {
        if (edad < p.edad)
            return -1;
        else if (edad > p.edad)
            return 1;
        else
            return nombre.compareTo(p.nombre);
    }
}
```

Ahora escribimos una clase que hace diversas operaciones con `Persona`, usando un `ArrayList`.

```
TestArrayList1.java
import java.util.ArrayList;
import java.util.Iterator;

public class TestArrayList1
{
    ArrayList<Persona> list = new ArrayList<Persona>();

    /**
     * Constructor
     */
    public TestArrayList1() {
        // Añadimos algunos elementos a la lista
        // add inserta siempre al final
        1 → list.add( new Persona("Max", 10));
        list.add( new Persona("Otto", 12));
        list.add( new Persona("Altea", 13));
    }
}
```

```

    } // end TestArrayList1

    public void printAll() {
        System.out.print("[ ");
        // usamos un iterador para imprimir los objetos
        3 ➡ Iterator it = list.iterator();
        while(it.hasNext()) {
            System.out.print(it.next() + ", ");
        }
        System.out.println(" ]");
    }

    public void printAll(String s) {
        System.out.println(s);
        printAll();
    }

    public void printAllConFor() {
        System.out.print("[ ");
        4 ➡ for(Persona p: list) {
            System.out.print(p + ", ");
        }
        System.out.println(" ]");
    }

    public void printAllConFor(String s) {
        System.out.println(s);
        printAllConFor();
    }

    public void test() {
        printAll("\nElementos iniciales (usando add(element)):" );

        2 ➡ list.add(1, new Persona("Pedro", 15));
        list.add(0, new Persona("Juan", 12));
        list.add(2, new Persona("Napoleón", 10));

        printAll("\nElementos añadidos usando add(i, element)");

        // buscamos Juan de 12 años y le sumamos uno a su edad
        int i;
        if( (i = list.indexOf(new Persona("Juan", 12))) != -1 ) {
            // recuperamos el objeto, cambiamos la edad,
            // y lo devolvemos a la lista
            Persona p = list.get(i);
            p.setEdad(p.getEdad()+1);
            list.set(i, p);

            printAllConFor("\nHemos sumado un año a la edad de Juan, e
            imprimimos con printAllConFor()");
        }

    }

    public static void main(String[]args) {
        TestArrayList1 tal = new TestArrayList1();
        tal.test();

    } // end main
}

```

El código usa dos métodos add distintos: 1 ➡ add(Persona p), que añade el objeto p al final de la lista; y 2 ➡ add(int pos, Persona p), que añade p en la posición pos del ArrayList.

`indexOf(Persona p)` devuelve la posición del objeto `p` en la lista, o `-1` si no existe⁷.

Hemos usado un `Iterator` para implementar un método `printAll`, que imprime todos los objetos de la lista (el método `iterator()` devuelve un `iterator` para la lista). En cambio, en el método `printAllConFor` usamos un `for` para imprimir los objetos de la lista. ¿Cuál es la diferencia? Gracias a la posibilidad de usar `for` para iterar colecciones y arreglos, introducida en Java 1.5, en la práctica no mucha. Con el `iterator` nos hemos evitado hacer referencia al tipo de la clase (`Persona`), pero sólo porque al usar `println` el método llamado es un *override* del `toString` de `Object`. En cambio, si quisiéramos llamar a un método que sólo existe en `Persona`, tendríamos que hacer un *downcast* y nos convendría más usar el `for`.

LinkedList

`LinkedList` implementa una lista enlazada usando la interfaz `List`. Aunque como `ArrayList` también permite acceder a un objeto de modo aleatorio (en una posición específica de la lista), la clase está más bien optimizada para ser recorrida con rapidez (acceso secuencial) y para inserciones y remociones rápidas en medio de la lista.

`LinkedList` implementa además algunos métodos que permiten insertar y quitar objetos de los extremos de la lista, lo permite usar la lista como una estructura como un *stack* (LIFO: *last in, first out*), *queue* (FIFO: *first in, first out*) o un *deque* (un *queue* en el que se pueden insertar y remover objetos por cualquiera de los dos extremos).

Algunos de estos métodos, además de los normales de la interfaz `List`:

<code>void addFirst(E o)</code>	inserta el elemento <code>o</code> al inicio de la lista
<code>void addLast(E o)</code>	inserta el elemento <code>o</code> al final de la lista
<code>E getFirst()</code>	devuelve el primer elemento de la lista, sin quitarlo
<code>E peek()</code>	sinónimo de <code>getFirst()</code>
<code>E removeFirst()</code>	devuelve y quita el primer elemento de la lista
<code>E remove()</code>	sinónimo de <code>removeFirst()</code>
<code>E poll()</code>	sinónimo de <code>removeFirst()</code>
<code>E getLast()</code>	devuelve el último elemento de la lista, sin quitarlo
<code>E removeLast()</code>	devuelve y quita el último elemento de la lista

Como ejemplo de `LinkedList`, nos basta cambiar las declaraciones del ejemplo de `ArrayList` por `LinkedList`:

```
TestLinkedList.java
import java.util.LinkedList;
import java.util.Iterator;

public class TestLinkedList
{
    LinkedList<Persona> list = new LinkedList<Persona>();
}
```

⁷ Se entiende que en un caso más realista el objeto `Persona` tendría que tener otra información adicional a nombre y edad, porque de lo contrario, si ya sabemos el nombre y la edad, ¿para qué queremos recuperar el objeto?

```

    /**
     * Constructor
     */
    public TestLinkedList() { ...
        (... resto del método y de la clase exactamente igual)
    }

```

Esto no quiere decir que `ArrayList` y `LinkedList` sean completamente intercambiables: en el código que se ha puesto de ejemplo lo son, porque no se usa ninguno de los métodos de `LinkedList` que no tiene `ArrayList` (`getFirst`, etc.).

Sets (Conjuntos)

Un *set* es una colección que no admite elementos duplicados. Si se intenta almacenar en el *set* más de una instancia de un objeto equivalente, el *set* se encarga de que haya siempre sólo una instancia. Según Eckel⁸ el uso más común de los *sets* es para verificar si un objeto es miembro o no del *set*. Por eso, una de sus operaciones más importantes es el “*lookup*” de un objeto.

La interfaz `Set` declara los mismos métodos que la interfaz `Collection`: a diferencia de los diversos tipos de `Lists`, el `Set` no añade ninguna funcionalidad más. Es como si fuera un `Collection` con un comportamiento distinto.

Como un *set* no admite elementos duplicados, sólo se pueden añadir a un `Set` objetos de clases que implementen correctamente el método `equals`, pues los derivados de `Set` usan `equals` para determinar si dos objetos son iguales.

Las clases de la librería de colecciones que implementan `Set` (por ejemplo, `HashSet`, `TreeSet`, `LinkedHashSet`) imponen distintas restricciones sobre el tipo de objeto que se puede almacenar en el *set*, y sobre métodos que los objetos almacenados en el `Set` deben implementar. La implementación más rápida es la de `HashSet`, de modo que si no hay motivo para usar una de las otras clases, es preferible usar `HashSet` cuando trabajamos con *sets*.

HashSet

La clase `HashSet` implementa un `Set` usando una tabla de hash detrás de las bambalinas para almacenar los datos. Esta clase no garantiza el orden en que se recuperan los elementos si se usa un iterador.

Los objetos que se almacenen utilizando `HashSet` deben implementar los métodos `equals` y `hashCode`. (Además estos dos métodos deben ser tales que si `e1.equals(e2)` entonces `e1.hashCode() == e2.hashCode()`). Como ya se dijo, los tipos primitivos y `String` implementan `hashCode` automáticamente.

La siguiente clase usa un `HashSet` para almacenar los números aleatorios generados por un loop. Como el `HashSet` no permite almacenar números duplicados, y los números aleatorios generados están en el rango [0,29], el *set* tendrá como máximo 30 números distintos.

TestHashSet.java

```
import java.util.*;
```

```
public class TestHashSet
{
```

```
    public static void main(String[] args) {
        Random rand = new Random();
```

⁸ ECKEL, BRUCE. *Thinking in Java*, 4th Edition, Prentice Hall, p. 415.

```

        Set<Integer> intSet = new HashSet<Integer>();

        for (int i=0; i< 10000; i++) {
            intSet.add(rand.nextInt(30));
        }
        System.out.println(intSet);
    }
}

```

Nótese que declaramos la variable `intSet` como tipo `Set` pero guardamos en `intSet` una referencia a un `HashSet`. También cabe notar que cuando se intenta insertar un elemento duplicado en `intSet` no se produce ningún error.

TreeSet

La clase `TreeSet` implementa un `Set` de modo tal que los elementos insertados están siempre ordenados. Para mantener el orden de los objetos, `TreeSet` espera que los objetos almacenados implementen la interfaz `Comparable`. El método `compareTo` debe ser coherente con el resultado del método `equals`.

Podemos cambiar el `HashSet` del ejemplo anterior por un `TreeSet`, y los enteros generados aleatoriamente aparecerán automáticamente ordenados.

```

TestTreeSet.java
import java.util.*;

public class TestTreeSet
{
    public static void main(String[] args) {
        Random rand = new Random(47);
        Set<Integer> intSet = new TreeSet<Integer>();

        for (int i=0; i< 10000; i++) {
            intSet.add(rand.nextInt(30));
        }
        System.out.println(intSet);
    }
}

```

hashCode, versión sencilla

El ejemplo de usar un set para almacenar enteros es bastante aburrido. ¿Qué tendríamos que hacer si queremos almacenar objetos de tipo `Persona` en un `HashSet`?

Para que un objeto `Persona` pueda ser almacenado en un `HashSet`:

- a) tiene que tener un override del método `equals`;
- b) tiene que tener un override del método `hashCode`;

El método `equals` ya está bien definido, de modo que veamos cómo escribir un método `hashCode` que sirva.

`Persona` tiene dos atributos que son los que distinguen a un objeto `Persona` de otro, de modo que parece lógico que nuestro método `hashCode` utilice estos dos atributos para generar un `hashCode` para toda la clase. Como Java ya trae métodos `hashCode` definidos para tipos `String` (el caso de nombre) y para `int` (el tipo de edad), podríamos usar una fórmula así:


```
hash(nombre, edad) = hash(nombre) + hash(edad);
```

Esta fórmula bastaría, porque se cumplirían los requisitos de una función hash:

- a) dos objetos *Persona* que tienen nombre y edad iguales tendrían el mismo hash.
- b) dos objetos *Persona* que tienen nombre y edad distintos tendrían probablemente hashes distintos (aunque se ve por la fórmula que no siempre).

La fórmula podría mejorarse multiplicando el hash de nombre por 100, de modo que por lo menos para personas menores de 100 años los dígitos del hash de edad no se mezclen con los de nombre.

```
hashCode(nombre, edad) = 100*hashCode(nombre) + hashCode(edad);
```

En la práctica, el `hashCode` de un número entero es el mismo número, de modo que nuestra fórmula quedaría así:

```
hashCode(nombre, edad) = 100*hashCode(nombre) + edad;
```

¿Qué dicen los expertos? Bloch⁹ da la siguiente receta para calcular el hash de una clase:

1. Escoge un valor entero constante distinto de cero, y almacénalo en una variable (por ejemplo, `resultado`).
2. Para cada atributo importante¹⁰ de tu clase calcula el código hash y almacénalo en la variable `c`, y recalcula `resultado` usando la siguiente fórmula:

```
resultado = 37* resultado + c
```

Si el atributo es un tipo que ya tiene implementado un método `hashCode` adecuado (tipos primitivos y `String`), entonces ya está el asunto resuelto. Si resulta que el atributo es una clase, tendremos que implementar primero el `hashCode` de esa clase, para poder calcular el `hashCode` de la nuestra.

Si aplicamos esta receta a *Persona*, escogiendo un `resultado` inicial de 17 (totalmente arbitrario) tendríamos:

```
hash(nombre, edad) = (37*17 + hashCode(nombre))*37 + hashCode(edad)
hash(nombre, edad) = 23237 + 37*hashCode(nombre) + hashCode(edad)
```

Como se ve, nuestra fórmula intuitiva para *Persona* no está tan lejos de la que usa Bloch, de modo que en nuestro ejemplo usaremos la nuestra.

Primero, derivamos de *Persona* una clase que implemente bien `hashCode`:

```
PersonaHash.java
public class PersonaHash extends Persona
{
    /**
     * Constructor
     */
    public PersonaHash(String nombre, int edad) {
        super(nombre, edad);
    }

    /**
     * hashCode - override del hashCode de Object
     */
}
```

⁹ BLOCH, JOSHUA. *Effective Java Programming Language Guide* (Addison-Wesley, 2001)

¹⁰ Un atributo importante quiere decir en la práctica los atributos que toman parte en el método `equals`.

```

        */
        @Override
        public int hashCode() {
            return 100*getNombre().hashCode() + getEdad();
        }
    }
}

```

Ahora el código que usa un HashSet para almacenar objetos PersonaHash:

S

```

TestHashSetPersona.java
import java.util.*;

public class TestHashSetPersona
{
    HashSet<PersonaHash> hashset = new HashSet<PersonaHash>();

    /* Constructor */
    public TestHashSetPersona()
    {
        // añadimos varios elementos al set
        hashset.add(new PersonaHash("Max", 10));
        hashset.add(new PersonaHash("Otto", 12));
        hashset.add(new PersonaHash("Altea", 13));
        hashset.add(new PersonaHash("Pedro", 15));
        hashset.add(new PersonaHash("Juan", 12));
        hashset.add(new PersonaHash("Napoleón", 10));
    }

    public void printAll() {
        Iterator it = hashset.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }

    public void test() {
        printAll();
    }

    public static void main(String[] args) {
        TestHashSetPersona th = new TestHashSetPersona();
        th.test();
    }
}

```

Si queremos que los elementos estén ordenados, puesto que Persona ya implementa Comparable, bastaría cambiar HashSet por TreeMap en el ejemplo anterior.

Mapas (*Maps*)

Un mapa es una clase que permite asociar un objeto a una llave y almacenarlo en el mapa, de modo que luego se pueda recuperar el objeto buscándolo por su llave (*key*). El objeto almacenado se llama también “valor” almacenado (*value*). El tipo de objeto de la llave debe implementar correctamente hashCode e equals.

Métodos básicos:

Map<K,V>

el constructor de Map. Devuelve un objeto Map que usa de llave objetos de tipo K y almacena objetos de tipo V. Por ejemplo: `HashMap<String, PersonaHash> personas =`

`new HashMap<String, PersonaHash>();` devuelve un `HashMap` que permite almacenar objetos `PersonaHash` usando `Strings` como llaves.

`V put(K key, V value)` añade el objeto `value` al mapa, asociándolo a la llave `key`. Por ejemplo,
`personas.put("Pérez", new PersonaHash("Pérez", 15));`

`V get(Object key)` devuelve el objeto de tipo `V` asociado a la llave `key`, o `null` si el mapa no contiene ningún objeto con esa llave¹¹.

`V remove(Object key)` igual que `get`, pero quita el objeto del mapa.

`int size()` devuelve el número de pares llave-valor en el mapa.

`Map` tiene algunos métodos que nos permiten extraer las llaves o los valores del mapa:

`Set<K> keySet()` devuelve un `Set` que contiene las llaves (*keys*) del mapa. Útil para iterar en busca de una llave en particular. Este *set* contiene objetos de tipo `K`, que es el tipo del *key*.

`Collection<V> values()` devuelve una colección que contiene los valores (*values*) del mapa. Útil para iterar por todos los objetos que contiene el mapa y realizar alguna acción sobre ellos. Esta colección contiene objetos de tipo `V`, donde `V` es el tipo de los objetos-valor almacenados en el mapa.

Otros métodos interesantes de `Map` son:

`boolean containsKey(Object key)` devuelve `true` si `key` es una de las llaves del mapa.

`boolean containsValue(Object value)` devuelve `true` si `value` es uno de los valores almacenados en el mapa.

HashMap

`HashMap` implementa un `Map` utilizando una tabla hash. El rendimiento para insertar y localizar pares llave-valor es constante.

TreeMap

`TreeMap` implementa un `Map` usando un árbol¹². El resultado es que los valores del `TreeMap` están siempre ordenados **según las llaves**. El tipo de objeto de las llaves, por tanto, deben implementar `Comparable` para que el `TreeMap` ordene los valores almacenados correctamente.

¹¹ Cabe la posibilidad de que se haya almacenado un `null` en el mapa. Por eso puede ser necesario verificar primero si la llave existe en el mapa usando el método `containsKey`. Si `containsKey(key)` devuelve `true`, entonces el `null` que devuelve `get` es el valor almacenado en el mapa; si devuelve `false`, quiere indicar que no se encontró un objeto asociado a esa llave.

¹² Más precisamente, un *red-black tree*. Esto no es importante, es transparente para los que usan la clase.

Clases utilitarias: Arrays y Collections

java.util proporciona dos clases utilitarias, Arrays, para manipular arreglos, y Collections, para manejar derivados de Collection (nótese el plural de estas clases utilitarias: son distintas de Array y Collection).

Los métodos concretos que implementa cada una se pueden consultar en la documentación de Java. A efectos del resumen, ponemos solamente un ejemplo:

```
TestArrayList.java
import java.util.Collections;
import java.util.Collection;
import java.util.ArrayList;

public class TestArrayList
{
    ArrayList<Persona> list = new ArrayList<Persona>();

    public TestArrayList() {
        Collections.addAll( list,
            new Persona("Max", 10),
            new Persona("Otto", 12),
            new Persona("Altea", 13)
        );
    } // end TestArrayList

    public void printAll() {
        System.out.println(list);
    }

    public void printAll(String s) {
        System.out.println(s);
        printAll();
    }

    public void test() {
        printAll("\nElementos iniciales (usando Collections.addAll):");

        Persona[] adicionales = {
            new Persona("Pedro", 15),
            new Persona("Juan", 12),
            new Persona("Napoleón", 10)
        };

        Collections.addAll(list, adicionales);
        printAll("\nElementos después de haber añadido un array usando
Collections.addAll");

        // buscamos Juan de 12 años y le sumamos uno a su edad
        int i;
        if( (i = list.indexOf(new Persona("Juan", 12))) != -1 ) {
            // recuperamos el objeto, cambiamos la edad,
            // y lo devolvemos a la lista
            Persona p = list.get(i);
            p.setEdad(p.getEdad()+1);
            list.set(i, p);

            System.out.println("\nHemos sumado un año a la edad de Juan");
        }

        // ordenamos el arreglo
        Collections.sort(list);
        printAll("\nAhora los objetos ordenados por edad usando
Collections.sort");
    }
}
```

```

        // mezclamos los elementos al azar
        Collections.shuffle(list);
        printAll("\nElementos desordenados usando Collections.shuffle");
        Collections.shuffle(list);
        printAll();
    }

    public static void main(String[]args) {
        TestArrayList tal = new TestArrayList();
        tal.test();
    } // end main
}

```

Comentarios, correcciones y sugerencias: Roberto Zoia (roberto.zoia@gmail.com)

This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.