

10. Colecciones y diccionarios

10.1 Colecciones: la clase ArrayList

Una colección en Java es una estructura de datos que permite almacenar muchos valores del mismo tipo; por tanto, conceptualmente es prácticamente igual que un *array*. Según el uso y según si se permiten o no repeticiones, Java dispone de un amplio catálogo de colecciones: `ArrayList` (lista), `ArrayBlockingQueue` (cola), `HashSet` (conjunto), `Stack` (pila), etc. En este manual estudiaremos la colección `ArrayList`.

Un `ArrayList` es una estructura en forma de lista que permite almacenar elementos del mismo tipo (pueden ser incluso objetos); su tamaño va cambiando a medida que se añaden o se eliminan esos elementos.

Nos podemos imaginar un `ArrayList` como un conjunto de celdas o cajoncitos donde se guardan los valores, exactamente igual que un *array* convencional. En la práctica será más fácil trabajar con un `ArrayList`.

En capítulos anteriores hemos podido comprobar la utilidad del *array*; es un recurso imprescindible que cualquier programador debe manejar con soltura. No obstante, el *array* presenta algunos inconvenientes. Uno de ellos es la necesidad de conocer el tamaño exacto en el momento de su creación. Una colección, sin embargo, se crea sin que se tenga que especificar el tamaño; posteriormente se van añadiendo y quitando elementos a medida que se necesitan.

Trabajando con *arrays* es frecuente cometer errores al utilizar los índices; por ejemplo al intentar guardar un elemento en una posición que no existe (índice fuera de rango). Aunque las colecciones permiten el uso de índices, no es necesario indicarlos siempre. Por ejemplo, en una colección del tipo `ArrayList`, cuando hay que añadir el elemento "Amapola", se puede hacer simplemente `flores.add("Amapola")`. Al no especificar índice, el elemento "Amapola" se añadiría justo al final de `flores` independientemente del tamaño y del número de elementos que se hayan introducido ya.

La clase `ArrayList` es muy similar a la clase `Vector`. Ésta última está obsoleta y, por tanto, no se recomienda su uso.

10.1.1 Principales métodos de ArrayList

Las operaciones más comunes que se pueden realizar con un objeto de la clase `ArrayList` son las siguientes:

add(elemento)

Añade un elemento al final de la lista.

add(indice, elemento)

Inserta un elemento en una posición determinada, desplazando el resto de elementos hacia la derecha.

clear()

Elimina todos los elementos pero no borra la lista.

contains(elemento)

Devuelve `true` si la lista contiene el elemento que se especifica y `false` en caso contrario.

get(indice)

Devuelve el elemento de la posición que se indica entre paréntesis.

indexOf(elemento)

Devuelve la posición de la primera ocurrencia del elemento que se indica entre paréntesis.

isEmpty()

Devuelve `true` si la lista está vacía y `false` en caso de tener algún elemento.

remove(indice)

Elimina el elemento que se encuentra en una posición determinada.

remove(elemento)

Elimina la primera ocurrencia de un elemento.

set(indice, elemento)

Machaca el elemento que se encuentra en una determinada posición con el elemento que se pasa como parámetro.

size()

Devuelve el tamaño (número de elementos) de la lista.

toArray()

Devuelve un *array* con todos y cada uno de los elementos que contiene la lista.

Puedes consultar todos los métodos disponibles en la [documentación oficial de la clase ArrayList](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)¹.

¹<http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

10.1.2 Definición de un ArrayList e inserción, borrado y modificación de sus elementos

A continuación se muestra un ejemplo en el que se puede ver cómo se declara un ArrayList y cómo se insertan y se extraen elementos.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList01 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        System.out.println("Nº de elementos: " + a.size());

        a.add("rojo");
        a.add("verde");
        a.add("azul");

        System.out.println("Nº de elementos: " + a.size());

        a.add("blanco");

        System.out.println("Nº de elementos: " + a.size());

        System.out.println("El elemento que hay en la posición 0 es " + a.get(0));
        System.out.println("El elemento que hay en la posición 3 es " + a.get(3));
    }
}
```

Observa que al crear un objeto de la clase ArrayList hay que indicar el tipo de dato que se almacenará en las celdas de esa lista. Para ello se utilizan los caracteres < y >. No hay que olvidar los paréntesis del final.



Es necesario importar la clase ArrayList para poder crear objetos de esta clase, para ello debe aparecer al principio del programa la línea `import java.util.ArrayList;`. Algunos IDEs (por ej. Netbeans) insertan esta línea de código de forma automática.

En el siguiente ejemplo se muestra un ArrayList de números enteros.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList011 {
    public static void main(String[] args) {

        ArrayList<Integer> a = new ArrayList<Integer>();

        a.add(18);
        a.add(22);
        a.add(-30);

        System.out.println("Nº de elementos: " + a.size());

        System.out.println("El elemento que hay en la posición 1 es " + a.get(1));
    }
}
```

Se define la estructura de la siguiente manera:

```
ArrayList<Integer> a = new ArrayList<Integer>();
```

Fíjate que no se utiliza el tipo simple `int` sino el wrapper `Integer`. Recuerda que los wrapper son clases que engloban a los tipos simples y les añaden nuevas funcionalidades (p. ej. permiten tratar a las variables numéricas como objetos). El wrapper de `int` es `Integer`, el de `float` es `Float`, el de `double` es `Double`, el de `long` es `Long`, el de `boolean` es `Boolean` y el de `char` es `Character`.

En el siguiente ejemplo podemos ver cómo extraer todos los elementos de una lista a la manera tradicional, con un bucle `for`.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList02 {
```

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

    ArrayList<String> a = new ArrayList<String>();

    a.add("rojo");
    a.add("verde");
    a.add("azul");
    a.add("blanco");
    a.add("amarillo");

    System.out.println("Contenido de la lista: ");

    for(int i=0; i < a.size(); i++) {
        System.out.println(a.get(i));
    }
}
```

Si estás acostumbrado al `for` clásico, habrás visto que es muy sencillo recorrer todos los elementos del `ArrayList`. No obstante, al trabajar con colecciones es recomendable usar el `for` al estilo `foreach` como se muestra en el siguiente ejemplo. Como puedes ver, la sintaxis es más corta y no se necesita crear un índice para recorrer la estructura.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList03 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add("rojo");
        a.add("verde");
        a.add("azul");
        a.add("blanco");
        a.add("amarillo");

        System.out.println("Contenido de la lista: ");

        for(String color: a) {
```

```

        System.out.println(color);
    }
}

```

Fíjate en estas líneas:

```

for(String color: a) {
    System.out.println(color);
}

```

El significado de este código sería el siguiente:

```

for(String color: a)

```

→ Saca uno a uno todos los elementos de `a` y ve metiéndolos en una variable de nombre `color`.

```

System.out.println(color);

```

→ Muestra por pantalla el contenido de la variable `color`.

Veamos ahora en otro ejemplo cómo eliminar elementos de un `ArrayList`. Se utiliza el método `remove()` y se puede pasar como parámetro el índice del elemento que se quiere eliminar, o bien el valor del elemento. O sea, `a.remove(2)` elimina el elemento que se encuentra en la posición 2 de `a` mientras que `a.remove("blanco")` elimina el valor "blanco".

Es importante destacar que el `ArrayList` se reestructura de forma automática después del borrado de cualquiera de sus elementos.

```

/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList04 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add("rojo");
    }
}

```

```
a.add("verde");
a.add("azul");
a.add("blanco");
a.add("amarillo");
a.add("blanco");

System.out.println("Contenido de la lista: ");

for(String color: a) {
    System.out.println(color);
}

if (a.contains("blanco")) {
    System.out.println("El blanco está en la lista de colores");
}

a.remove("blanco");

System.out.println("Contenido de la lista después de quitar la " +
    "primera ocurrencia del color blanco: ");

for(String color: a) {
    System.out.println(color);
}

a.remove(2);
System.out.println("Contenido de la lista después de quitar el " +
    "elemento de la posición 2: ");

for(String color: a) {
    System.out.println(color);
}
}
```

A continuación se muestra un ejemplo en el que se “machaca” una posición del `ArrayList`. Al hacer `a.set(2, "turquesa")`, se borra lo que hubiera en la posición 2 y se coloca el valor "turquesa". Sería equivalente a hacer `a[2] = "turquesa"` en caso de que `a` fuese un *array* tradicional en lugar de un `ArrayList`.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList05 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add("rojo");
        a.add("verde");
        a.add("azul");
        a.add("blanco");
        a.add("amarillo");

        System.out.println("Contenido del vector: ");

        for(String color: a)
            System.out.println(color);

        a.set(2, "turquesa");

        System.out.println("Contenido del vector después de machacar la posición 2: ");

        for(String color: a) {
            System.out.println(color);
        }
    }
}
```

El método `add` permite añadir elementos a un `ArrayList` como ya hemos visto. Por ejemplo, `a.add("amarillo")` añade el elemento "amarillo" al final de `a`. Este método se puede utilizar también con un índice de la forma `a.add(1, "turquesa")`. En este caso, lo que se hace es insertar en la posición indicada. Lo mejor de todo es que el `ArrayList` se reestructura de forma automática desplazando el resto de elementos.


```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList06 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add("rojo");
        a.add("verde");
        a.add("azul");
        a.add("blanco");
        a.add("amarillo");

        System.out.println("Contenido de la lista: ");

        for(String color: a) {
            System.out.println(color);
        }

        a.add(1, "turquesa");

        System.out.println("Contenido del vector después de insertar en la posición 1: ");

        for(String color: a) {
            System.out.println(color);
        }
    }
}
```

10.1.3 ArrayList de objetos

Una colección `ArrayList` puede contener objetos que son instancias de clases definidas por el programador. Esto es muy útil sobre todo en aplicaciones de gestión para guardar datos de alumnos, productos, libros, etc.

En el siguiente ejemplo, definimos una lista de gatos. En cada celda de la lista se almacenará un objeto de la clase `Gato`.

```
/**
 * Uso de un ArrayList de objetos
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList07 {
    public static void main(String[] args) {

        ArrayList<Gato> g = new ArrayList<Gato>();

        g.add(new Gato("Garfield", "naranja", "mestizo"));
        g.add(new Gato("Pepe", "gris", "angora"));
        g.add(new Gato("Mauri", "blanco", "manx"));
        g.add(new Gato("Ulises", "marrón", "persa"));

        System.out.println("\nDatos de los gatos:\n");

        for (Gato gatoAux: g) {
            System.out.println(gatoAux+"\n");
        }
    }
}
```

En el siguiente apartado se muestra la definición de la clase `Gato`.

10.1.4 Ordenación de un ArrayList

Los elementos de una lista se pueden ordenar con el método `sort`. El formato es el siguiente:

```
Collections.sort(lista);
```

Observa que `sort` es un método de clase que está definido en `Collections`. Para poder utilizar este método es necesario incluir la línea

```
import java.util.Collections;
```

al principio del programa. A continuación se muestra un ejemplo del uso de `sort`.

```
/**
 * Ordenación de un ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.Collections;
import java.util.ArrayList;

public class EjemploArrayList071 {
    public static void main(String[] args) {

        ArrayList<Integer> a = new ArrayList<Integer>();

        a.add(67);
        a.add(78);
        a.add(10);
        a.add(4);

        System.out.println("\nNúmeros en el orden original:");
        for (int numero: a) {
            System.out.println(numero);
        }

        Collections.sort(a);

        System.out.println("\nNúmeros ordenados:");
        for (int numero: a) {
            System.out.println(numero);
        }
    }
}
```

También es posible ordenar una lista de objetos. En este caso es necesario indicar el criterio de ordenación en la definición de la clase. En el programa principal, se utiliza el método `sort` igual que si se tratase de una lista de números o de palabras como se muestra a continuación.

```

/**
 * Ordenación de un ArrayList de objetos
 *
 * @author Luis José Sánchez
 */

import java.util.Collections;
import java.util.ArrayList;

public class EjemploArrayList08 {
    public static void main(String[] args) {

        ArrayList<Gato> g = new ArrayList<Gato>();

        g.add(new Gato("Garfield", "naranja", "mestizo"));
        g.add(new Gato("Pepe", "gris", "angora"));
        g.add(new Gato("Mauri", "blanco", "manx"));
        g.add(new Gato("Ulises", "marrón", "persa"));
        g.add(new Gato("Adán", "negro", "angora"));

        Collections.sort(g);

        System.out.println("\nDatos de los gatos ordenados por nombre:");

        for (Gato gatoAux: g) {
            System.out.println(gatoAux+"\n");
        }
    }
}

```

Ahora bien, en la definición de la clase `Gato` hay que indicar de alguna manera cómo se debe realizar la ordenación, ya que Java no sabe de antemano si los gatos se ordenan según el color, el nombre, el peso, etc.

Lo primero que hay que hacer es indicar que los objetos de la clase `Gato` se pueden comparar unos con otros. Para ello, cambiamos la siguiente línea:

```
public class Gato
```

por esta otra:

```
public class Gato implements Comparable<Gato>
```

Lo siguiente y no menos importante es definir el método `compareTo`. Este método debe devolver un 0 si los elementos que se comparan son iguales, un número negativo si

el primer elemento que se compara es menor que el segundo y un número positivo en caso contrario. Afortunadamente, las clases `String`, `Integer`, `Double`, etc. ya tienen implementado su propio método `compareTo` así que tenemos hecho lo más difícil. Lo único que deberemos escribir en nuestro código es un `compareTo` con los atributos que queremos comparar.

En el caso que nos ocupa, si queremos ordenar los gatos por nombre, tendremos que implementar el `compareTo` de la clase `Gato` de tal forma que nos devuelva el resultado del `compareTo` de los nombres de los gatos que estamos comparando, de la siguiente manera:

```
public int compareTo(Gato g) {  
    return (this.nombre).compareTo(g.getNombre());  
}
```

Si en lugar de ordenar por nombre, quisiéramos ordenar por raza, el método `compareTo` de la clase `Gato` sería el siguiente:

```
public int compareTo(Gato g) {  
    return (this.raza).compareTo(g.getRaza());  
}
```

A continuación se muestra la definición completa de la clase `Gato`.

```
/**  
 * Definición de la clase Gato  
 *  
 * @author  
 */  
  
public class Gato implements Comparable<Gato> {  
    private String nombre;  
    private String color;  
    private String raza;  
  
    public Gato(String nombre, String color, String raza) {  
        this.nombre = nombre;  
        this.color = color;  
        this.raza = raza;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
}
```

```
public String getRaza() {
    return raza;
}

public String toString() {
    return "Nombre: " + this.nombre + "\nColor: " + this.color + "\nRaza: " + this.raza;
}

public int compareTo(Gato g) {
    return (this.nombre).compareTo(g.getNombre());
}

public boolean equals(Gato g) {
    return (this.nombre).equals(g.getNombre());
}
}
```

10.2 Diccionarios: la clase HashMap

Imagina un diccionario inglés-español. Queremos saber qué significa la palabra “*stiff*”. Sabemos que en el diccionario hay muchas entradas y en cada entrada tenemos una palabra en inglés y su correspondiente traducción al español. Buscando por la “s” encontramos que “*stiff*” significa “agujetas”.

Un diccionario en Java funciona exactamente igual. Contiene una serie de elementos que son las entradas que a su vez están formadas por un par (clave, valor). La clave (key) permite acceder al valor. No puede haber claves duplicadas. En el ejemplo anterior, la clave sería “*stiff*” y el valor “agujetas”.

Java dispone de varios tipos de diccionarios: `HashMap`, `EnumMap`, `Hashtable`, `IdentityHashMap`, `LinkedHashMap`, etc. Nosotros estudiaremos el diccionario `HashMap`.

10.2.1 Principales métodos de HashMap

Algunos de los métodos más importantes de la clase `HashMap` son:

get(clave)

Obtiene el valor correspondiente a una clave. Devuelve `null` si no existe esa clave en el diccionario.

put(clave, valor)

Añade un par (clave, valor) al diccionario. Si ya había un valor para esa clave, se machaca.

keySet()

Devuelve un conjunto (set) con todas las claves.

values()

Devuelve una colección con todos los valores (los valores pueden estar duplicados a diferencia de las claves).

entrySet()

Devuelve una colección con todos los pares (clave, valor).

containsKey(clave)

Devuelve `true` si el diccionario contiene la clave indicada y `false` en caso contrario.

getKey()

Devuelve la clave de la entrada. Se aplica a una sola entrada del diccionario (no al diccionario completo), es decir a una pareja (clave, valor).

Por ejemplo:

```
for (Map.Entry pareja: m.entrySet()) {  
    System.out.println(pareja.getKey());  
}
```

getValue()

Devuelve el contenido de la entrada. Se aplica a una entrada del diccionario (no al diccionario completo), es decir a una pareja (clave, valor).

Por ejemplo:

```
for (Map.Entry pareja: m.entrySet()) {  
    System.out.println(pareja.getValue());  
}
```

10.2.2 Definición de un HashMap e inserción, borrado y modificación de entradas

Al declarar un diccionario hay que indicar los tipos tanto de la clave como del valor. En el siguiente ejemplo definimos el diccionario `m` que tendrá como clave un número entero y una cadena de caracteres como valor. Este diccionario se declara de esta forma:

```
HashMap<Integer, String> m = new HashMap<Integer, String>();
```

No hay que olvidar importar la clase al principio del programa:

```
import java.util.HashMap;
```

Para insertar una entrada en el diccionario se utiliza el método `put` indicando siempre la clave y el valor. Veamos un ejemplo completo.

```
/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.HashMap;

public class EjemploHashMap01 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.println("Los elementos de m son: \n" + m);
    }
}
```

Para extraer valores se utiliza el método `get`. Se proporciona una clave y el diccionario nos devuelve el valor, igual que un diccionario de verdad. Si no existe ninguna entrada con la clave que se indica, se devuelve `null`.

```
/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.HashMap;

public class EjemploHashMap011 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
```



```

        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.println(m.get(921));
        System.out.println(m.get(605));
        System.out.println(m.get(888));
    }
}

```

¿Y si queremos extraer todas las entradas? Tenemos varias opciones. Podemos usar el método `print` directamente sobre el diccionario de la forma `System.out.print(diccionario)` como vimos en un ejemplo anterior; de esta manera se muestran por pantalla todas las entradas encerradas entre llaves. También podemos convertir el diccionario en un `entrySet` (conjunto de entradas) y mostrarlo con `print`; de esta forma se obtiene una salida por pantalla muy parecida a la primera (en lugar de llaves se muestran corchetes). Otra opción es utilizar un `for` para recorrer una a una todas las entradas. En este último caso hay que convertir el diccionario en un `entrySet` ya que no se pueden sacar las entradas directamente del diccionario. Estas dos últimas opciones se ilustran en el siguiente ejemplo.

```

/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.HashMap;
import java.util.Map;

public class EjemploHashMap02 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.println("Todas las entradas del diccionario extraídas con entrySet:");
        System.out.println(m.entrySet());

        System.out.println("\nEntradas del diccionario extraídas una a una:");
    }
}

```

```

        for (Map.Entry pareja: m.entrySet()) {
            System.out.println(pareja);
        }
    }
}

```

A continuación se muestra el uso de los métodos `getKey` y `getValue` que extraen la clave y el valor de una entrada respectivamente.

```

/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.*;

public class EjemploHashMap03 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.println("Código\tNombre\n-----\t-----");

        for (Map.Entry pareja: m.entrySet()) {
            System.out.print(pareja.getKey() + "\t");
            System.out.println(pareja.getValue());
        }
    }
}

```

En el último programa de ejemplo hacemos uso del método `containsKey` que nos servirá para saber si existe o no una determinada clave en un diccionario y del método `get` que, como ya hemos visto, sirve para extraer un valor a partir de su clave.

```
/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.*;

public class EjemploHashMap04 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.print("Por favor, introduzca un código: ");
        int codigoIntroducido = Integer.parseInt(System.console().readLine());

        if (m.containsKey(codigoIntroducido)) {
            System.out.print("El código " + codigoIntroducido + " corresponde a ");
            System.out.println(m.get(codigoIntroducido));
        } else {
            System.out.print("El código introducido no existe.");
        }
    }
}
```