Gobierno de Canarias
Servicio Canario de Empleo

Centro GASTRONÓMICO del Atlántico

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
MINISTERIO DE TRABAJO Y ECONOMÍA SOCIAL
SERVICIO PÚBLICO DE EMPLEO ESTATAL
SEPE

Programación en Java 2023

Tema 8 Colecciones y Diccionarios, Ficheros y Librerías Externas

Tema 7

1. Colecciones y Diccionarios 2
1.1. Colecciones: la clase ArrayList 2
1.1.1. Principales métodos de ArrayList 3
1.1.2. Definición de un ArrayList e inserción, borrado y modificación de sus elementos 4
1.1.3. ArrayList de objetos 13
1.1.4. Ordenación de un ArrayList 15
1.1.5. Cálculo del número de ocurrencias 21
1.2. Diccionarios: la clase HashMap 21
1.2.1. Principales métodos de HashMap 22
1.2.2. Definición de un HasMap e inserción, borrado y modificación de entradas 23
2. Ficheros 28
2.1. Lectura de un fichero de texto 28
2.2. Escritura sobre un fichero de texto 32
2.3. Lectura y escritura combinadas 33
2.4. Otras operaciones sobre ficheros 34
2.5. Procesamiento de archivos de texto 35
2.6. Archivos CSV 37
2.7. Lectura de un fichero CSV 38
3. Librerías Externas 40
3.1. Maven 40
3.1.1. Maven Central 42
3.2. Librería IText 42
3.3. Librería openCSV 44
3.3.1. Omitir la línea de la cabecera 46
4. Bibliografía 48

1. Colecciones y Diccionarios

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.

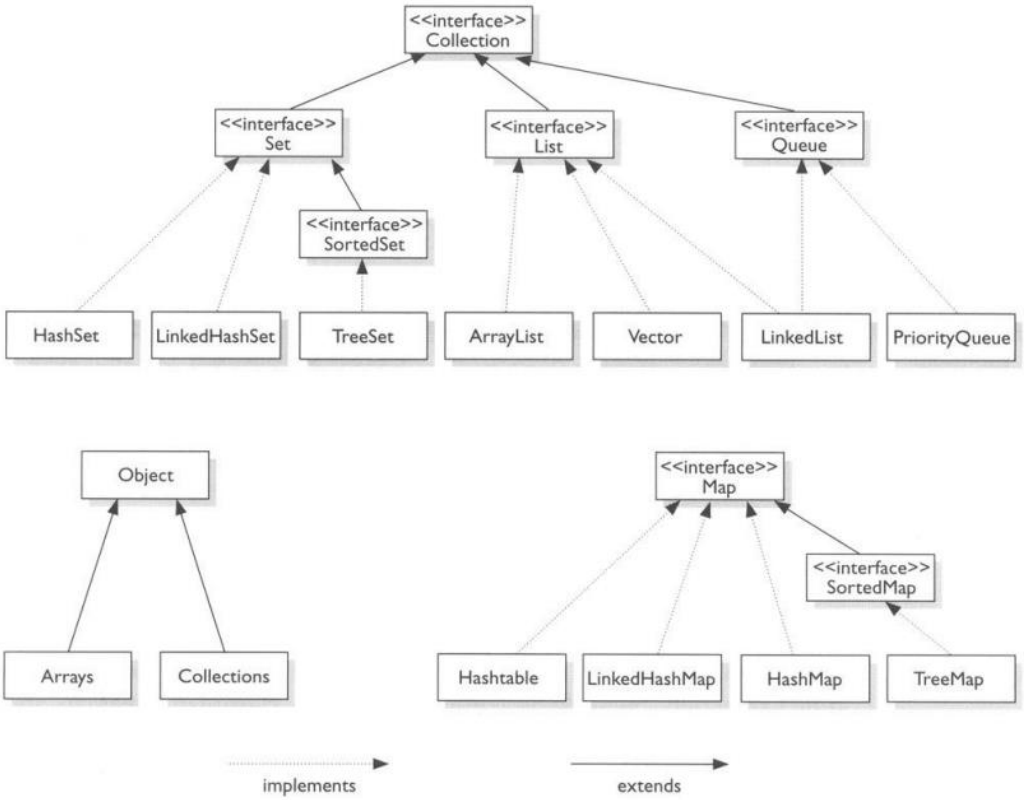

Figura 1 Jerarquía de clases para las Colecciones en Java

1.1. Colecciones: la clase 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 que almacena modelos de coches, cuando hay que añadir un nuevo elemento, se hace de la siguiente manera:

```
coches.add("Opel Zafira");
```

Al no especificar índice, el elemento "Opel Zafira" se añadiría justo al final de coches independientemente del tamaño y del número de elementos que se hayan introducido ya.

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

1.1.1. Principales métodos de ArrayList

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

Método	Descripción
add(elemento)	Añade un elemento al final de la lista
add(índice, 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(índice)	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(índice)	Elimina el elemento que se encuentra en una posición determinada.
remove(elemento)	Elimina la primera ocurrencia de un elemento.

set(índice, elemento)	Sobreescribe 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.

Nota: Se pueden consultar todos los métodos disponibles en la documentación oficial de la clase ArrayList:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html

1.1.2. 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
*
*/
import java.util.ArrayList;
public class EjemploArrayList01 {
 public static void main(String[ ] args) {

 // Declaración e inicialización de un objeto de tipo ArrayList
 ArrayList<String> a = new ArrayList<String>();

 // Imprimimos el número de elementos del array.
 System.out.println("Nº de elementos: " + a.size());

 // Añadimos elementos al ArrayList
 a.add("rojo");
 a.add("verde");
 a.add("azul");

 // Imprimimos de nuevo el número de elementos del array.
```

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

// Añadimos más elementos al ArrayList
a.add("blanco");

// Imprimimos de nuevo el número de elementos del array.
System.out.println("Nº de elementos: " + a.size());

// Imprimimos los elementos que están en una posición determinada.
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));
}
}
```

El resultado de ejecutar el programa anterior es el siguiente:

```
Nº de elementos: 0
Nº de elementos: 3
Nº de elementos: 4
El elemento que hay en la posición 0 es rojo
El elemento que hay en la posición 3 es blanco
```

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:

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

Ejemplo de creación de un objeto de la clase ArrayList utilizando la Interfaz List con polimorfismo:

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

En el caso anterior se crea un ArrayList para almacenar objetos de tipo String.

Nota: Es necesario importar la clase ArrayList para poder crear objetos de esta clase, para ello debe aparecer al principio del programa la línea:

```
/**
* Ejemplo de uso de la clase ArrayList
*
*/
import java.util.ArrayList;
public class EjemploArrayList02 {
 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));
 }
}
```

El resultado de ejecutar el programa anterior es el siguiente:

```
Nº de elementos: 3
El elemento que hay en la posición 1 es 22
```

Se define la estructura de la siguiente manera:

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

Fíjate que no se utiliza el tipo simple int sino la clase **Wrapper** de tipo Integer. Recuerda que los wrapper son clases que engloban a los tipos primitivos de Java y les añaden nuevas funcionalidades (p. ej. permiten tratar a las variables numéricas como objetos). La clase Wrapper de int es Integer, la de float es Float, la de double es Double, la de long es Long, la de boolean es Boolean y la 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
*
*/
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(int i=0; i < a.size(); i++) {
 System.out.println(a.get(i));
 }
 }
}
```

El resultado de ejecutar el código anterior es el siguiente:

```
Contenido de la lista:
rojo
verde
azul
blanco
amarillo
```

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
*
*/
import java.util.ArrayList;
public class EjemploArrayList031{
 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);
 }
 }
}
```

El resultado de ejecutar el código anterior es:

```
Contenido de la lista:
rojo
verde
azul
blanco
amarillo
```

Fíjate en estas líneas:

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

El bucle for, en cada iteración, saca un elemento del ArrayList a y lo mete en una variable con el nombre color, que tendrá el mismo tipo que cada elemento del ArrayList, en este caso String.

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. Es decir, a.remove(2) elimina el elemento que se encuentra en la posición 2 de a mientras que colores.remove("blanco") elimina el valor "blanco" del ArrayList colores.

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
*
*/
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);
}
}
}
```

El resultado de ejecutar el código anterior es el siguiente:

```
Contenido de la lista:
rojo
verde
azul
blanco
amarillo
blanco

El blanco está en la lista de colores
Contenido de la lista después de quitar la primera ocurrencia del color blanco:
rojo
verde
azul
amarillo
blanco
Contenido de la lista después de quitar el elemento de la posición 2:
rojo
verde
amarillo
blanco
```

A continuación se muestra un ejemplo en el que se sobreescribe 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
**/
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 resultado de ejecutar el código anterior es:

```
Contenido del vector:
rojo
verde
azul
blanco
```

```
amarillo
Contenido del vector después de machacar la posición 2:
rojo
verde
turquesa
blanco
amarillo
```

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
*
*/
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);
}
}
}
```

El resultado de ejecutar el código anterior es:

```
Contenido de la lista:
rojo
verde
azul
blanco
amarillo
Contenido del vector después de insertar en la posición 1:
rojo
turquesa
verde
azul
blanco
amarillo
```

1.1.3. 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
*
*/
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");
 }
 }
}
```

El resultado de ejecutar el programa anterior es el siguiente:

```
Datos de los gatos:
Nombre: Garfield
Color: naranja
Raza: mestizo
Nombre: Pepe
Color: gris
Raza: angora
Nombre: Mauri
Color: blanco
Raza: manx
Nombre: Ulises
Color: marrón
```

```
Raza: persa
```

1.1.4. Ordenación de un ArrayList

Los elementos de una lista se pueden ordenar con el método sort() de la clase Collections. El formato es el siguiente:

Collections.sort(lista);

Observa que sort() es un método estático o de clase que está definido en la clase Collections que se encuentra en la API de Java java.util. Para poder utilizar este método es necesario incluir la siguiente línea al principio del programa:

import java.util.Collections;

```
/**
* Ordenación de un ArrayList de enteros
*
*/
import java.util.Collections;
import java.util.ArrayList;
public class EjemploArrayList08 {
 public static void main(String[ ] args) {
 ArrayList<Integer> numeros = new ArrayList<Integer>();

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

 System.out.println("\nNúmeros en el orden original:");
 for (int numero: numeros ) {
 System.out.println(numero);
 }
 Collections.sort(numeros);
 System.out.println("\nNúmeros ordenados:");
 for (int numero: numeros) {
```

```
System.out.println(numero);
}
}
}
```

El resultado de ejecutar el código anterior es el siguiente:

```
Números en el orden original:
67
78
10
4
Números ordenados:
4
10
67
78
```

A continuación, se muestra un ejemplo del uso de sort() que ordena un ArrayList de palabras:

```
/**
* Ordenación de un ArrayList de String
*
*/
import java.util.Collections;
import java.util.ArrayList;
public class EjemploArrayList09 {
 public static void main(String[ ] args) {
 ArrayList<String> palabras = new ArrayList<String>();

 palabras.add("Juan");
 palabras.add("Pedro");
 palabras.add("Ana");
 palabras.add("Diebo");
```

```
System.out.println("\nPalabras en el orden original:");
for (String palabra: palabras) {
System.out.println(palabra);
}
Collections.sort(palabras);

System.out.println("\nPalabras ordenadas:");
for (String palabra: palabras) {
System.out.println(palabra);
}
}
}
```

El resultado de ejecutar el código anterior es el siguiente:

```
Palabras en el orden original:
Juan
Pedro
Ana
Diego
Números ordenados:
Ana
Diego
Juan
Pedro
```

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.

Tenemos una clase llamada Estudiante:

```
public class Estudiante {
 private int codigo;
 private String nombre;
 private String direccion;
```

```
// Constructor
public Estudiante(int codigo, String nombre, String direccion)
{
this.codigo = codigo;
this.nombre = nombre;
this.direccion = direccion;
}

public int getCodigo(){
return codigo;
}
public String getNombre(){
return nombre;
}

// Used to print student details in main()
public String toString()
{
return this.codigo + "" + this.nombre + "" + this.direccion;
}
}
```

La clase Estudiante contiene 3 atributos: codigo, nombre y direccion. Para ordenar a los estudiantes por un campo de tipo entero, como puede ser código, tenemos que crear una clase que implemente de la interfaz Comparator e implementar el método compare() de la siguiente manera:

```
import java.util.Comparator;
public class SortByCodigo implements Comparator<Estudiante>{
// Usado para ordenador de forma ascendente
// por el campo codigo.
public int compare(Estudiante a, Estudiante b)
{
return a.getCodigo() - b.getCodigo();
}
}
```

Para ordenar a los estudiantes por un campo de tipo entero, como puede ser código, tenemos que crear una clase que implemente de la interfaz Comparator e implementar el método compare() de la siguiente manera:

```
import java.util.Comparator;
public class SortByNombre implements Comparator<Estudiante>{
 // Usado para ordenador de manera ascendente por
 // el campo nombre.
 public int compare(Estudiante a, Estudiante b)
 {
 int res = String.CASE_INSENSITIVE_ORDER.compare(
 a.getNombre(), b.getNombre());
 if (res == 0){
 res = a.getNombre().compareTo(b.getNombre());
 }
 return res;
 }
}
```

A continuación, en el programa principal, tenemos un ArrayList llamado estudiantes, que contendrá un listado de todos los estudiantes. Para ordenar la lista debemos invocar al método Collections.sort() con dos parámetros:

- 1er parámetros: lista de elementos.
- 2do parámetro: objeto a partir de una clase que tenga implementado el método compare().

Así, si queremos ordenar por el campo codigo, le pasamos un objeto de la clase SortByCodigo como segundo parámetro:

Collections.sort(estudiantes, new SortByCodigo());

Y si queremos ordenar por el campo nombre, le pasamos un objeto de la clase SortByNombre como segundo parámetro:

Collections.sort(estudiantes, new SortByNombre());

El código de la clase principal quedaría de la siguiente manera:

```
import java.util.ArrayList;
```

```
import java.util.Collections;
public class Main {
 public static void main(String[ ] args) {
 ArrayList<Estudiante> estudiantes = new ArrayList<Estudiante>();
 estudiantes.add(new Estudiante(111, "Jaime Rodríguez", "Londres"));
 estudiantes.add(new Estudiante(131, "Ana Méndez", "Nueva York"));
 estudiantes.add(new Estudiante(121, "Silvia Suárez", "Hong Kong"));

 System.out.println("Lista desordenada:");
 for (int i=0; i < estudiantes.size(); i++)
 System.out.println(estudiantes.get(i));

 Collections.sort(estudiantes, new SortByCodigo());

 System.out.println("\nLista ordenada por el campo codigo:");
 for (Estudiante e: estudiantes)
 System.out.println(e);

 Collections.sort(estudiantes, new SortByNombre());

 System.out.println("\nLista ordenada por el campo nombre:");
 for (Estudiante e: estudiantes)
 System.out.println(e);
 }
}
```

A continuación, se muestra la salida del programa:

```
Lista desordenada:
111 Jaime Rodríguez Londres
131 Ana Méndez Nueva York
121 Silvia Suárez Hong Kong

Lista ordenada por el campo codigo:
111 Jaime Rodríguez Londres
121 Silvia Suárez Hong Kong
```

```
131 Ana Méndez Nueva York


Lista ordenada por el campo nombre:
131 Ana Méndez Nueva York
111 Jaime Rodríguez Londres
121 Silvia Suárez Hong Kong
```

1.1.5. Cálculo del número de ocurrencias

En Java 7, para encontrar cuántas veces se repetía una cadena en un ArrayList, había que recorrer mediante un bucle for toda la estructura, y utilizar una variable que se iba incrementando a medida que encontrábamos alguna ocurrencia, por ejemplo, imaginemos que, dada la clase Estudiante definida en el apartado anterior, queremos encontrar todos los estudiantes que se llaman Juan:

```
int count = 0;
for (Estudiantes estudiante : estudiantes){
 if (estudiante.getName().equals("Juan")){
 count++;
 }
}
System.out.println(count);
```

A partir de Java 8 podemos utilizar los Streams, que son una característica avanzada de Java que nos permite realizar operaciones sobre Colecciones, en este caso, encontrar el número de estudiantes que se llaman de una forma en particular:

```
int count =(int)estudiantes.stream()
 .filter(p -> p.getNombre().equals("Juan"))
 .count();
System.out.println(count);
```

1.2. 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.

1.2.1. Principales métodos de HashMap

Algunos de los métodos más importantes de la clase HasMap son:

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

1.2.2. Definición de un HasMap 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 HasMap
*
*/
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);
 }
}
```

El resultado de ejecutar el programa anterior es el siguiente:

```
Los elementos de m son:
{921=Cindy Nero, 537=Alan Brito, 219=Víctor Tilla, 924=Amalia Núñez, 700=César Vázquez,
605=Esteban Quito}
```

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 HasMap
*
*/
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));
 }
}
```

El resultado de ejecutar el programa anterior es el siguiente:

```
Cindy Nero
Esteban Quito
null
```

¿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
*
*/
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);
 }
 }
}
```

El resultado de ejecutar el programa anterior es el siguiente:

```
Todas las entradas del diccionario extraídas con entrySet:
[921=Cindy Nero, 537=Alan Brito, 219=Víctor Tilla, 924=Amalia Núñez, 700=César Vázquez,
```

```
605=Esteban Quito ]
Entradas del diccionario extraídas una a una:
921=Cindy Nero
537=Alan Brito
219=Víctor Tilla
924=Amalia Núñez
700=César Vázquez
605=Esteban Quito
```

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
*
*/
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());
 }
 }
}
```

El resultado de ejecutar el programa anterior es el siguiente:

```
Código Nombre
------ ------------
921 Cindy Nero
537 Alan Brito
219 Víctor Tilla
924 Amalia Núñez
700 César Vázquez
605 Esteban Quito
```

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
*
*/
import java.util.*;
public class EjemploHashMap04 {
 public static void main(String[ ] args) {
Scanner sc = new Scanner(System.in);
 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 = sc.nextInt();
 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.");
}
sc.close();
}
}
```

2. Ficheros

Mediante un programa en Java se puede acceder al contenido de un fichero grabado en un dispositivo de almacenamiento (por ejemplo, en el disco duro) tanto para leer como para escribir (grabar) datos.

La información contenida en las variables, los arrays, los objetos o cualquier otra estructura de datos es volátil, es decir, se pierde cuando se cierra el programa. Los ficheros permiten tener ciertos datos almacenados y disponibles en cualquier momento para cuando nuestro programa los necesite.

Pensemos por ejemplo en un juego. Podríamos utilizar un fichero para guardar el ranking de jugadores con las mejores puntuaciones. De esta manera, estos datos no se perderían al salir del juego. De igual forma, en un programa de gestión, puede ser útil tener un fichero de configuración con datos como el número máximo de elementos que se muestran en un listado o los distintos tipos de IVA aplicables a las facturas. Al modificar esta información, quedará almacenada en el fichero correspondiente y no se perderá cuando se cierre el programa.

Cuando un programa se cierra, se pierde la información almacenada en variables, arrays, objetos o cualquier otra estructura. Si queremos conservar ciertos datos, debemos guardarlos en ficheros.

La creación y uso de ficheros desde un programa en Java se lleva a cabo cuando hay poca información que almacenar o cuando esa información es heterogénea. En los casos en que la información es abundante y homogénea es preferible usar una base de datos relacional (por ejemplo, MySQL) en lugar de ficheros.

2.1. Lectura de un fichero de texto

Aunque Java puede manejar también ficheros binarios, vamos a centrarnos exclusivamente en la utilización de ficheros de texto. Los ficheros de texto tienen una gran ventaja, se pueden crear y manipular mediante cualquier editor como por ejemplo GEdit, Notepad, etc.

Siempre que vayamos a usar ficheros, deberemos incluir al principio del programa una o varias líneas para cargar las clases necesarias. Aunque se pueden cargar varias clases en una sola línea usando el asterisco de la siguiente manera:

import java.io.;*

nosotros no lo haremos así. Se importarán las clases usando varias líneas, una línea por cada clase que se importa en el programa:

import java.io.BufferedReader;

import java.io.FileReader;

Nota: No es necesario saber de memoria los nombres de las clases, el entorno de desarrollo Eclipse detecta qué clases hacen falta cargar y añade los import de forma automática.

Todas las operaciones que se realicen sobre ficheros deberán estar incluidas en un bloque try-catch. Esto nos permitirá mostrar mensajes de error y terminar el programa de una forma ordenada en caso de que se produzca algún fallo, por ejemplo, el fichero que estamos intentando abrir no existe o no tenemos permisos para acceder a él, etc.

El bloque try-catch tiene el siguiente formato:

```
try {
 Operaciones_con_fichero
} catch (tipo_de_error nombre_de_variable) {
 Mensaje_de_error
}
```

Tanto para leer como para escribir utilizamos lo que en programación se llama un manejador de fichero. Es algo así como una variable que hace referencia al fichero con el que queremos trabajar.

En nuestro ejemplo, el manejador de fichero se llama bf y se crea de la siguiente manera:

BufferedReader bf = new BufferedReader(new FileReader("tenerife.txt"));

El fichero con el que vamos a trabajar tiene por nombre tenerife.txt y contiene información extraída de la Wikipedia sobre la isla de Tenerife. A partir de aquí, siempre que queramos realizar una operación sobre ese archivo, utilizaremos su manejador de fichero asociado, es decir, bf.

En el ejemplo que se muestra a continuación, el programa lee el contenido del fichero tenerife.txt y lo muestra tal cual en pantalla, igual que si lo mostráramos a través de un terminal:

cat tenerife.txt

Observa que se va leyendo el fichero línea a línea mediante el método bf.readLine() hasta que se acaban las líneas. Cuando no quedan más líneas por leer se devuelve el valor null.

```
/**
* Ejemplo de uso de la clase File
```

```
* Lectura de un fichero de texto
*
*/
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
class EjemploFichero01 {
 public static void main(String[ ] args) {
 try {
 BufferedReader bf = new BufferedReader(new FileReader("tenerife.txt"));
 String linea = "";
 while (linea != null) {
 System.out.println(linea);
 linea = bf.readLine();
 }
 bf.close();
 } catch (FileNotFoundException e) { // qué hacer si no se encuentra el fichero
 System.out.println("No se encuentra el fichero tenerife.txt");
 } catch (IOException e) { // qué hacer si hay un error en la lectura del fichero
 System.out.println("No se puede leer el fichero tenerife.txt");
 }
 }
}
```

Es importante cerrar el fichero cuando se han realizado todas las operaciones necesarias sobre él. En este ejemplo, esta acción se ha llevado a cabo con el método bf.close().

A continuación, se muestra un programa un poco más complejo. Se trata de una aplicación que pide por teclado un nombre de fichero. Previamente en ese fichero (por ejemplo, numeros.txt) habremos introducido una serie de números, a razón de uno por línea. Se podrían leer también los números si estuvieran separados por comas o espacios, aunque sería un poco más complicado (no mucho más). Los números pueden contener decimales ya que se van a leer como Double. Cada número que se lee del fichero se va sumando de tal forma que la suma total estará contenida en la variable suma; a la par se va llevando la cuenta de los elementos que se van leyendo en la variable i. Finalmente, dividiendo la suma total entre el número de elementos obtenemos la media aritmética de los números contenidos en el fichero.

```
/**
* Ejemplo de uso de la clase File
* Calcula la media de los números que se encuentran en un fichero de texto
*/
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
class EjemploFichero02 {
 public static void main(String[ ] args) {
 Scanner sc = new Scanner(System.in);
 System.out.print("Introduzca el nombre del archivo donde se encuentran los
números: ");
 String nombreFichero = sc.nextLine();
 try {
 BufferedReader bf = new BufferedReader(new FileReader(nombreFichero));
 String linea = "0";
 int i = 0;
 double suma = 0;

 while (linea != null) {
 i++;
 suma += Double.parseDouble(linea);
 linea = bf.readLine();
 }
 i--;
 bf.close();
 System.out.println("La media es " + suma / (double)i);
 } catch (IOException e) {
 System.out.println(e.getMessage());
 }
 sc.close();
 }
}
```

2.2. Escritura sobre un fichero de texto

La escritura en un fichero de texto es, si cabe, más fácil que la lectura. Solo hay que cambiar System.out.print("texto") por bw.write("texto"). Se pueden incluir saltos de línea, tabuladores y espacios igual que al mostrar un mensaje por pantalla.

Es importante ejecutar el método bw.close() después de realizar la escritura; de esta manera nos aseguramos que se graba toda la información en el disco.

Nota: Para probar antes de escribir en fichero podemos imprimir todo primero por consola todo lo que quieras escribir en el fichero. Cuando compruebes que lo que se ve por pantalla es realmente lo que quieres grabar en el fichero, entonces y solamente entonces, cambia System.out.print("texto") por manejador.write("texto").

A continuación, se muestra un programa de ejemplo que crea un fichero de texto y luego escribe en él tres palabras, una por cada línea.

```
/**
* Ejemplo de uso de la clase File
* Escritura en un fichero de texto
*/
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
class EjemploFichero03 {
 public static void main(String[ ] args) {
 try {
 BufferedWriter bw = new BufferedWriter(new FileWriter("fruta.txt"));
 bw.write("naranja\n");
 bw.write("limón\n");
 bw.write("manzana\n");
 bw.close();
 } catch (IOException e) {
 System.out.println("No se ha podido escribir en el fichero");
 }
 }
}
```

2.3. Lectura y escritura combinadas

Las operaciones de lectura y escritura sobre ficheros se pueden combinar de tal forma que haya un flujo de lectura y otro de escritura, uno de lectura y dos de escritura, tres de lectura, etc.

En el ejemplo que presentamos a continuación hay dos flujos de lectura y uno de escritura. Observa que se declaran en total tres manejadores de fichero (dos para lectura y uno para escritura). El programa va leyendo, de forma alterna, una línea de cada fichero (una línea de fichero1.txt y otra línea de fichero2.txt) mientras queden líneas por leer en alguno de los ficheros y, al mismo tiempo, va guardando esas líneas en otro fichero con nombre mezcla.txt.

```
/**
* Ejemplo de uso de la clase File
* Mezcla de dos ficheros
*/
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
class EjemploFichero04 {
 public static void main(String[ ] args) {
 try {
 BufferedReader bf1 = new BufferedReader(new FileReader("fichero1.txt"));
 BufferedReader bf2 = new BufferedReader(new FileReader("fichero2.txt"));
 BufferedWriter bw = new BufferedWriter(new FileWriter("mezcla.txt"));
 String linea1 = "";
 String linea2 = "";
 while ( (linea1 != null)||(linea2 != null) ) {
 linea1 = bf1.readLine();
 linea2 = bf2.readLine();
 if (linea1 != null)
 bw.write(linea1 + "\n");
 if (linea2 != null)
 bw.write(linea2 + "\n");
 }
 bf1.close();
```

```
bf2.close();
bw.close();
} catch (IOException ioe) {
System.out.println("Se ha producido un error de lectura/escritura");
System.err.println(ioe.getMessage());
}
}
}
```

2.4. Otras operaciones sobre ficheros

Además de leer desde o escribir en un fichero, hay otras operaciones relacionadas con los archivos que se pueden realizar desde un programa escrito en Java.

Veremos tan solo un par de ejemplos. Si quieres profundizar más, échale un vistazo a la documentación oficial de la clase File en la URL:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/File.html

El siguiente ejemplo muestra por pantalla un listado con todos los archivos que contiene un directorio.

```
/**
* Ejemplo de uso de la clase File
* Listado de los archivos del directorio actual
*/
import java.io.File;
class EjemploFichero05 {
public static void main(String[ ] args) {
File fichero = new File("."); // se indica la ruta entre comillas
// el punto (.) es el directorio actual
String[ ] listaArchivos = fichero.list();
for(int i = 0; i < listaArchivos.length; i++){
System.out.println(listaArchivos[i]);
}
}
}
```

El siguiente programa de ejemplo comprueba si un determinado archivo existe o no mediante el método exists() y, en caso de que exista, lo elimina mediante el método delete(). Si intentáramos borrar un archivo que no existe obtendríamos un error.

```
/**
* Ejemplo de uso de la clase File
* Comprobación de existencia y borrado de un fichero
*
*/
import java.io.File;
class EjemploFichero06 {
 public static void main(String[ ] args) {
 Scanner sc = new Scanner(System.in);
 System.out.print("Introduzca el nombre del archivo que desea borrar: ");
 String nombreFichero = sc.nextLine();
 File fichero = new File(nombreFichero);
 if (fichero.exists()) {
 fichero.delete();
 System.out.println("El fichero se ha borrado correctamente.");
 }
 else {
 System.out.println("El fichero " + nombreFichero + " no existe.");
 }

 sc.close();
 }
}
```

2.5. Procesamiento de archivos de texto

La posibilidad de realizar desde Java operaciones con ficheros abre muchas posibilidades a la hora de procesar archivos: cambiar una palabra por otra, eliminar ciertos caracteres, mover de sitio una línea o una palabra, borrar espacios o tabulaciones al final de las líneas o cualquier otra cosa que se nos pueda ocurrir.

Cuando se procesa un archivo de texto, los pasos a seguir son los siguientes:

1. Leer una línea del fichero origen mientras quedan líneas por leer.
2. Modificar la línea (normalmente utilizando los métodos que ofrece la clase String).
3. Grabar la línea modificada en el fichero destino.

4. Volver al paso 1.

A continuación, tienes algunos métodos de la clase String que pueden resultar muy útiles para procesar archivos de texto:

Método	Descripción
charAt(int n)	Devuelve el carácter que está en la posición n-ésima de la cadena (Recuerda que la primera posición es la número 0).
indexOf(String palabra)	Devuelve un número que indica la posición en la que comienza una palabra determinada.
length()	Devuelve la longitud de la cadena.
replace(char c1, char c2)	Devuelve una cadena en la que se han cambiado todas las ocurrencias del carácter c1 por el carácter c2.
substring(int inicio,int fin)	Devuelve una subcadena.
toLowerCase()	Convierte todas las letras en minúsculas.
toUpperCase()	Convierte todas las letras en mayúsculas.

Puedes consultar todos los métodos de la clase String en la documentación oficial en la siguiente URL:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html

A continuación, tienes un ejemplo en el que se usan los métodos descritos anteriormente.

```
/**
* Ejemplos de uso de String
*/
public class EjemplosString {
public static void main(String[ ] args) {
 System.out.println("\nEjemplo 1");
 System.out.println("En la posición 2 de \"berengena\" está la letra " +
 "berengena".charAt(2));

 System.out.println("\nEjemplo 2");
 String frase = "Hola caracola.";
 char[ ] trozo = new char[10];
 trozo[0] = 'z'; trozo[1] = 'z'; trozo[2] = 'z';
```

```
frase.getChars(2, 7, trozo, 1);
System.out.print("El array de caracteres vale ");
System.out.println(trozo);
System.out.println("\nEjemplo 3");
System.out.println("La secuencia \"co\" aparece en la frase en la posición " +
frase.indexOf("co"));
System.out.println("\nEjemplo 4");
System.out.println("La palabra \"murciélago\" tiene " + "murciélago".length() +
" letras");

System.out.println("\nEjemplo 5");
String frase2 = frase.replace('o', 'u');
System.out.println(frase2);
System.out.println("\nEjemplo 6");
frase2 = frase.substring(3, 10);
System.out.println(frase2);

System.out.println("\nEjemplo 7");
System.out.println(frase.toLowerCase());

System.out.println("\nEjemplo 8");
System.out.println(frase.toUpperCase());
}
}
```

2.6. Archivos CSV

Los archivos CSV (del inglés comma-separated values) son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas (o punto y coma en donde la coma es el separador decimal) y las filas por saltos de línea.

El formato CSV es muy sencillo y no indica un juego de caracteres concreto, ni cómo van situados los bytes, ni el formato para el salto de línea. Estos puntos deben indicarse muchas veces al abrir el archivo, por ejemplo, con una hoja de cálculo.

El formato CSV no está estandarizado. La idea básica de separar los campos con una coma es muy clara, pero se vuelve complicada cuando el valor del campo también contiene

comillas dobles o saltos de línea. Las implementaciones de CSV pueden no manejar esos datos, o usar comillas de otra clase para envolver el campo. Pero esto no resuelve el problema: algunos campos también necesitan embeber estas comillas, así que las implementaciones de CSV pueden incluir caracteres o secuencias de escape.

Además, el término "CSV" también denota otros formatos de valores separados por delimitadores que usan delimitadores diferentes a la coma (como los valores separados por tabuladores). Un delimitador que no está presente en los valores de los campos (como un tabulador) mantiene el formato simple.

Ejemplo de representación de datos:

Año	Marca	Modelo	Descripción	Precio
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture	Extended Edition	4900.00
1999	Chevy	Venture	Extended Edition, Very Large	5000.00
1996	Jeep	Grand Cherokee	air, moon roof, loaded	4799.00

La tabla superior puede ser representada en el siguiente archivo CSV:

```
Año,Marca,Modelo,Descripción,Precio
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
1996,Jeep,Grand Cherokee,"air, moon roof, loaded",4799.00
```

2.7. Lectura de un fichero CSV

Dado el siguiente fichero:

```
"1.0.0.0","1.0.0.255","16777216","16777471","AU","Australia"
"1.0.1.0","1.0.3.255","16777472","16778239","CN","China"
"1.0.4.0","1.0.7.255","16778240","16779263","AU","Australia"
"1.0.8.0","1.0.15.255","16779264","16781311","CN","China"
"1.0.16.0","1.0.31.255","16781312","16785407","JP","Japan"
"1.0.32.0","1.0.63.255","16785408","16793599","CN","China"
"1.0.64.0","1.0.127.255","16793600","16809983","JP","Japan"
"1.0.128.0","1.0.255.255","16809984","16842751","TH","Thailand"
```

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class CSVReader {
 public static void main(String[ ] args) {
 String csvFile = "country.csv";
 BufferedReader br = null;
 String line = "";
 String cvsSplitBy = ",";
 try {
 br = new BufferedReader(new FileReader(csvFile));
 while ((line = br.readLine()) != null) {
 // Uso de coma como separador
 String[ ] country = line.split(cvsSplitBy);
 System.out.println("Country [code= " + country[4] +
 ", name=" + country[5] + "]");
 }
 } catch (FileNotFoundException e) {
 e.printStackTrace();
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
}
```

La salida del programa anterior:

```
Country [code= "AU", name="Australia"]
Country [code= "CN", name="China"]
Country [code= "AU", name="Australia"]
Country [code= "CN", name="China"]
Country [code= "JP", name="Japan"]
Country [code= "CN", name="China"]
Country [code= "JP", name="Japan"]
```

```
Country [code= "TH", name="Thailand"]
```

3. Librerías Externas

Las librerías externas nos permiten añadir funcionalidades a nuestra aplicación mediante librerías que han desarrollado terceros. Muchas de estas funcionalidades tienen que ver con: generación de informes, desarrollo de test, creación de logs, mapeo objeto-relacional, ...

Estas librerías pueden ser añadidas a nuestro proyecto de manera manual (la manera más fácil), o bien, configurando nuestro proyecto para que use Maven.

3.1. Maven

Son cientos o quizás miles de librerías que podemos utilizar para múltiples propósitos. Programar no significa inventar la rueda sino aprovechar los recursos que ya existen de la mejor forma para lograr los objetivos propuestos.

Una de las herramientas más útiles a la hora de utilizar librerías de terceros es Maven. Maven se utiliza en la gestión y construcción de software. Posee la capacidad de realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. Es decir, hace posible la creación de software con dependencias incluidas dentro de la estructura del JAR. Es necesario definir todas las dependencias del proyecto (librerías externas utilizadas) en un fichero propio de todo proyecto Maven, el POM (Project Object Model). Este es un archivo en formato XML que contiene todo lo necesario para que a la hora de generar el fichero ejecutable de nuestra aplicación este contenga todo lo que necesita para su ejecución en su interior.

Sin embargo, la característica más importante de Maven es su capacidad de trabajar en red. Cuando definimos las dependencias de Maven, este sistema se encargará de ubicar las librerías que deseamos utilizar en Maven Central, el cual es un repositorio que contiene cientos de librerías constantemente actualizadas por sus creadores. Maven permite incluso buscar versiones más recientes o más antiguas de un código dado y agregarlas a nuestro proyecto.

Todo se hará de forma automática sin que el usuario tenga que hacer nada más que definir las dependencias.

Ejemplo de fichero pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.prog.applications</groupId>
<artifactId>application1</artifactId>
<version>1.0</version>
<packaging>jar</packaging>

<name>Maven Quick Start Archetype</name>
<url>http://maven.apache.org</url>

<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.8.2</version>
</dependency>
</dependencies>

</project>
```

En el fichero anterior, en el fichero pom.xml se aprecia que la aplicación tiene la dependencia de la librería externa JUnit.

Nota: JUnit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java. Es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

En cada fichero pom.xml se deben definir al menos los siguientes elementos:

- **groupId:** Es el conjunto de proyectos al que pertenece nuestro proyecto. Por ejemplo, yo puedo tener todos mis programas de ejemplo en un grupo llamado "com.prog.applicacion". Este nombre que pongamos aquí va a servir de paquete inicial para todas las clases del proyecto. Todos los proyectos maven deben pertenecer a un grupo, aunque sea único para él, que se denominará groupId.
- **artifactId:** Es el nombre que queramos dar al proyecto. Maven creará un directorio con este nombre y el jar que genere para el proyecto tendrá también este nombre. Todos los proyectos Maven tienen un nombre para identificarlos, que se denominará artifactId.

- **version:** Es la versión de la librería, a medida que vaya introduciendo cambios en ella se irá modificando el número de versión.
- **packaging:** Es cómo está empaquetada la librería, la mayoría de librerías están empaquetadas con fichero de extensión .jar.

3.1.1. Maven Central

Maven Central es un repositorio en el que están alojados las principales librerías hechas por terceros. Desde Maven Central podemos descargarnos las librerías que nos hagan falta, bien de manera manual, o bien haciendo uso de la herramienta Maven.

La URL de Maven Central:

https://mvnrepository.com/repos/central

3.2. Librería IText

La librería iText es una librería OpenSource para crear y manipular documentos PDF. La URL en Maven Central es:

https://mvnrepository.com/artifact/com.itextpdf/itextpdf

A continuación, se muestra un ejemplo de uso de la librería para generar un informe:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import com.itextpdf.text.Document;
import com.itextpdf.text.pdf.PdfWriter;
import com.itextpdf.text.Paragraph;
public class EjemploPersonasPDF {
public static void main(String args[ ]) {
try {
// Listado de Personas
ArrayList<String> lista = new ArrayList<String>();
lista.add("Juan Garcia");
lista.add("Andres Hernández");
lista.add("Reyes Estévez");
lista.add("Pedro Moreno");
// Se crea la instancia del Documento.
```

```
Document document = new Document();
// Se crea el OutputStream para el fichero donde queremos
 // escribir el pdf.
OutputStream outputStream = new FileOutputStream(
 new File("Usuarios.pdf"));
// Se crea el PDFWriter que realiza la asociación entre el
 // Document y el OutputStream.
PdfWriter.getInstance(document, outputStream);
// Se abre el documento.
document.open();
// Se añade la cabecera
document.add(new Paragraph("LISTADO:"));
document.add(Chunk.NEWLINE);
// Se añade el contenido del ArrayList.
for (String persona: lista) {
document.add(new Paragraph(persona));
}
// Se cierra el DocumentoClose document and outputStream.
document.close();
outputStream.close();
System.out.println("Documento PDF creado correctamente.");
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

- **document.add(new Paragraph("LISTADO:"));:** añade al documento pdf el texto listado como si fuera un párrafo. Lo bueno de utilizar el párrafo es que nos añade un retorno de carro al final de la línea.
- **document.add(Chunk.NEWLINE);:** añade al documento pdf una línea en blanco.
- **document.add(new Paragraph(persona));:** añade al documento la información de una persona.

Para utilizar este ejemplo correctamente es necesario importar la librería externa itext:

```
<!-- https://mvnrepository.com/artifact/com.itextpdf/itextpdf -->
```

```
<dependency>
 <groupId>com.itextpdf</groupId>
 <artifactId>itextpdf</artifactId>
 <version>5.5.13</version>
</dependency>
```

Más ejemplos del uso de la librería itextpdf:

- https://tutorialspointexamples.com/itext-tutorial-java-beginners-eclipse
- http://chuwiki.chuidiang.org/index.php?title=Ejemplo_sencillo_de_creaci%C3%B3n_de_un_pdf_con_iText

3.3. Librería openCSV

La librería iTextPDF es una librería OpenSource para crear y manipular documentos CSV. La URL en Maven Central es:

https://mvnrepository.com/artifact/com.opencsv/opencsv

Esta librería tiene métodos más avanzados para leer ficheros CSV ya que son capaces de distinguir el delimitador cuando se encuentra incluido dentro de unos de los campos.

A continuación, se muestra un ejemplo de uso de la librería para leer un documento CSV:

Fichero usuarios.csv:

```
Rajeev Kumar Singh,rajeevs@example.com,+91-9999999999,India
Sachin Tendulkar,sachin@example.com,+91-9999999998,India
Barak Obama,barak.obama@example.com,+1-1111111111,United States
Donald Trump,donald.trump@example.com,+1-2222222222,United States
```

Ejemplo de código:

```
import com.opencsv.CSVReader;
import java.io.IOException;
import java.io.Reader;
import java.nio.file.Files;
import java.nio.file.Paths;
public class EjemploCSV01 {
 private static final String SAMPLE_CSV_FILE_PATH = "usuarios.csv";
 public static void main(String[ ] args) throws IOException {
 try (
```

```
Reader reader = Files.newBufferedReader(Paths.get(SAMPLE_CSV_FILE_PATH));
CSVReader csvReader = new CSVReader(reader);
){
// Reading Records One by One in a String array
String[ ] nextRecord;
while ((nextRecord = csvReader.readNext()) != null) {
System.out.println("Name : " + nextRecord[0]);
System.out.println("Email : " + nextRecord[1]);
System.out.println("Phone : " + nextRecord[2]);
System.out.println("Country : " + nextRecord[3]);
System.out.println("==========================");
}
}
}
}
```

En el ejemplo anterior, leemos las entradas del fichero CSV una a una mediante el método readNext().

CSVReader proporciona un método llamado readAll() para leer todas las entradas y guardarlas en una estructura List<String[ ]>:

```
import com.opencsv.CSVReader;
import java.io.IOException;
import java.io.Reader;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
public class EjemploCSV02 {
private static final String SAMPLE_CSV_FILE_PATH = "users.csv";
public static void main(String[ ] args) throws IOException {
try (
Reader reader = Files.newBufferedReader(Paths.get(SAMPLE_CSV_FILE_PATH));
CSVReader csvReader = new CSVReader(reader);
){
// Reading Records One by One in a String array
List<String[ ]> records = csvReader.readAll();
for (String[ ] record : records) {
```

```
System.out.println("Name : " + record[0]);
System.out.println("Email : " + record[1]);
System.out.println("Phone : " + record[2]);
System.out.println("Country : " + record[3]);
System.out.println("----------------------------");
}
}
}
}
```

3.3.1. Omitir la línea de la cabecera

Si estamos intentando leer un fichero CSV que contiene una cabecera, si queremos omitirla, podemos usar la clase CSVReaderBuilder para construir un CSVReader especificándole el número de líneas a omitir.

```
import com.opencsv.CSVReaderBuilder;
CSVReader csvReader = new CSVReaderBuilder(reader).withSkipLines(1).build();
```

Para utilizar este ejemplo correctamente es necesario importar la librería externa openCSV:

```
<!-- https://mvnrepository.com/artifact/com.opencsv/opencsv -->
<dependency>
 <groupId>com.opencsv</groupId>
 <artifactId>opencsv</artifactId>
 <version>3.8</version>
</dependency>
```

A parte, la librería openCSV tiene dependencias, Maven es capaz de gestionar las dependencias de una librería, pero si añadimos a mano la librería openCSV, deberemos añadir a mano también las librerías commons-beanutils y commons-lang3:

```
<!-- https://mvnrepository.com/artifact/commons-beanutils/commons-beanutils -->
<dependency>
 <groupId>commons-beanutils</groupId>
 <artifactId>commons-beanutils</artifactId>
 <version>1.9.2</version>
</dependency>
```

```
<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->
<dependency>
 <groupId>org.apache.commons</groupId>
 <artifactId>commons-lang3</artifactId>
 <version>3.4</version>
</dependency>
```

4. Bibliografía

- Material suministrado por CGA.
- https://www.arquitecturajava.com/java-arraylist-for-y-sus-opciones/
- http://www.dit.upm.es/~santiago/doc/fprg/ejemplos/solucion/interfaces/procesador/Diccionario.java
- http://www.w3big.com/es/java/java-dictionary-class.html
- https://misapuntesdeprogramacion.wordpress.com/2013/02/14/paquete-java-io/
- https://javadesdecero.es/avanzado/io-entrada-salida/
- https://www.programarya.com/Cursos/Java/Librerias
- https://profile.es/blog/librerias-java/
- https://soajp.blogspot.com/2017/02/anadir-librerias-manualmente-usando.html