



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor:* Tista García Edgar Ing.

*Asignatura:* Estructura de Datos y Algoritmos II

*Grupo:* 5

*No de Práctica(s):* 10

*Integrante(s):* Téllez González Jorge Luis

*No. de Equipo de  
cómputo empleado:* ---

*No. de Lista o Brigada:* X

*Semestre:* 2020-2

*Fecha de entrega:* 06/05/2020

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_



# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Objetivos</b>	<b>3</b>
<b>3. Bibliotecas en Java</b>	<b>3</b>
3.1. Clase File . . . . .	3
3.2. FileReader . . . . .	4
3.3. BufferedReader . . . . .	4
3.4. FileWriter . . . . .	5
3.5. BufferedWriter . . . . .	5
<b>4. Manipulación básica de archivos</b>	<b>6</b>
4.1. Creación de un archivo . . . . .	7
4.2. Sobreescritura de un archivo . . . . .	8
4.3. Edición de un archivo . . . . .	9
4.4. Eliminación de un archivo . . . . .	10
<b>5. Conclusiones</b>	<b>11</b>

# 1. Introducción

El uso de archivos es una actividad cotidiana hoy en día. Entre los diferentes tipos de archivos se encuentran los archivos de texto, de video, ejecutables, entre otros. Sus usos y aplicaciones son múltiples y representan el componente básico de un *sistema operativo*.



Figura 1: *Diferentes tipos de archivos. Su extensión indica el contenido esperado de cada uno.*

En general, se define a un **archivo** como un objeto presente en una computadora capaz de almacenar información de diversa índole, la cual puede ser manipulada por otros programas a su conveniencia. Estos archivos tienen un *nombre*, el cuál representa la etiqueta principal para diferenciarlos y acceder a la ubicación donde se encuentran almacenados en memoria.

El sistema operativo Windows implementa el uso de extensiones como una forma de identificar y manipular adecuadamente un archivo; en el caso de sistemas operativos basados en Linux esta práctica es únicamente hecha con fines de formato o convención. Prácticamente todo sistema operativo actual cuenta con *sistemas de archivos* capaces de manipular estos objetos de acuerdo a las necesidades del usuario y el sistema.

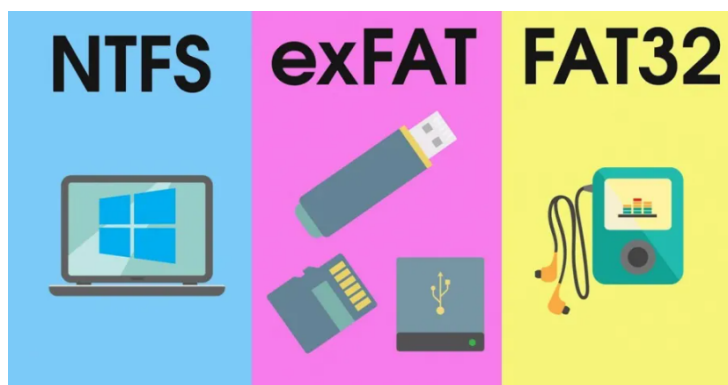


Figura 2: *Sistemas de archivos principales del S.O. Windows.*

Existen 2 aproximaciones o enfoques respecto a los archivos. La primer aproximación se refiere a los archivos como un *stream* o *flujo de datos* desde una fuente a un receptor. Las entradas y salidas de datos en **Java** se manejan mediante *streams*, los cuales representan una conexión entre la fuente emisora y el destino de los datos. En **Java**, pueden destacarse 4 jerarquías sobre flujos de entrada/salida (I/O):

- **Flujo de bytes:** Las clases derivadas de *InputStream* y *OutputStream* manejan los flujos de datos como un *stream* de bytes.
- **Flujo de caracteres:** Las clases derivadas de *Reader* y *Writer* trabajan con *streams* o *flujos* de caracteres.



Figura 3: *Streams* o *flujos de datos*.

El segundo enfoque, el cuál es característico del curso de *EDA II*, observa a los archivos como *estructuras de datos complejas* utilizadas para operar con el *sistema operativo* en la memoria externa de la computadora. Es por ello que estos objetos tienen una importancia fundamental en el funcionamiento de una computadora.

En el siguiente reporte escrito se abordarán los métodos y las clases más comunes para operar con archivos de texto plano (*.txt*), detallando las operaciones realizadas y mostrando las salidas obtenidas contenidas en un archivo generado por medio de **Java**.

## 2. Objetivos

- El estudiante conocerá e identificará aspectos sobre los archivos, como las operaciones, el tipo de acceso y organización lógica.

## 3. Bibliotecas en Java

### 3.1. Clase File

La clase *File* es utilizada para generar una referencia abstracta de un archivo y su directorio de ubicación. De esta forma, es posible manipular un archivo haciendo uso de los métodos que implementa. El constructor básico de esta clase sigue la siguiente notación:

*File archivo = new File("Ruta del archivo");*

Debido a que el formato de los directorios puede variar entre diferentes S.O, se debe de tomar en consideración el correcto formato de la ruta donde se encuentra el archivo. Es importante señalar que, para efectos de esta práctica, se ha utilizado *Windows 10*.

A continuación, se enlistan los métodos más comunes de esta clase:

- **createNewFile()**: Genera un nuevo archivo nombrado a partir de la ruta abstracta del objeto *File* si y solo si no existe un archivo con el mismo nombre en la ruta especificada.
- **delete()**: Elimina el archivo denotado a partir de su ruta abstracta.
- **exists()**: Verifica que el archivo identificado a partir de su ruta abstracta exista.
- **mkdir()**: Genera una carpeta o directorio a partir de la ruta abstracta del objeto *File*.
- **renameTo(File dest)**: Renombra un archivo denotado por su ruta abstracta.

Una vez mencionado lo anterior, se abordarán las clases principales utilizadas para manipular el contenido de los archivos por medio de *flujos de caracteres*.

### 3.2. FileReader

Esta clase hereda de *InputStreamReader*, la cuál es utilizada para obtener los caracteres ingresados desde una fuente externa a partir de un flujo de bytes. Esta clase permite leer flujos de caracteres en un archivo de texto. Su constructor principal es el siguiente:

*FileReader fr = new FileReader(File archivo);*

### 3.3. BufferedReader

Por medio de esta clase puede leerse el texto contenido en un flujo de caracteres de forma eficiente. El tamaño del *buffer de datos* puede ser especificado; aunque el tamaño por defecto suele ser suficiente para la mayoría de propósitos.

Los objetos de este método se inicializan con un objeto de la clase *Reader* o que herede características de la misma; como el caso de *FileReader*:

*BufferedReader br = new BufferedReader(Reader in);*

Entre sus métodos más comunes se encuentran:

- **read()**: Método utilizable para leer y recuperar un único carácter, o leer caracteres a partir de un arreglo; recuperados en su representación entera.
- **readLine()**: Lee y recupera una línea de texto. Considera que una línea termina al detectarse un salto de línea (`\n`) o un retorno de carro (`\r`).
- **close()**: Cierra el flujo de datos y libera cualquier recurso del sistema asociado al mismo.
- **skip(long n)**: Utilizado para saltar una determinada cantidad de caracteres.

### 3.4. FileWriter

Esta clase hereda de *OutputStreamReader*, la cuál funciona como un puente de flujos de caracteres a flujos de bytes. Esta clase permite escribir una secuencia de caracteres en un archivo de texto. Su constructor principal es el siguiente:

```
FileWriter fw = new FileWriter(File archivo);
```

Existe un segundo constructor para los objetos de esta clase que, a diferencia del primero, contiene un valor booleano como segundo parámetro. Dependiendo de su valor, sucederá lo siguiente:

- Un valor **true** hara que, al momento de escribir un flujo de caracteres sobre un archivo, este se escriba al final del mismo.
- Un valor **false** provocará que cualquier flujo de caracteres sea escrito al inicio de un archivo.

### 3.5. BufferedWriter

Por medio de esta clase puede escribirse en un archivo una cadena de texto contenida en un flujo de caracteres de forma eficiente. El tamaño del *buffer de datos* puede ser especificado; aunque el tamaño por defecto suele ser suficiente para la mayoría de propósitos; como en el caso de *BufferedReader*.

Los objetos de este método se inicializan con un objeto de la clase *Writer* o que herede características de la misma; como el caso de *FileWriter*:

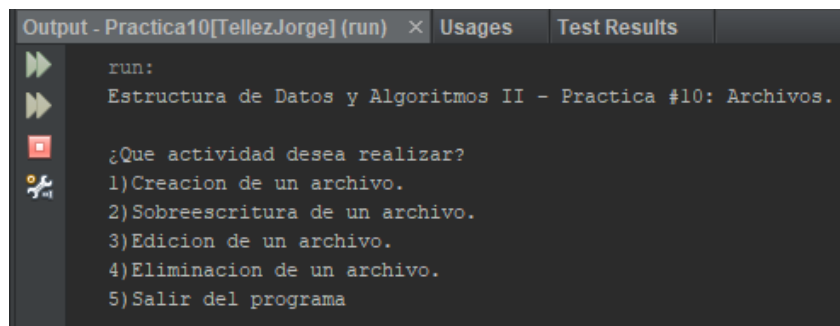
```
BufferedWriter bw = new BufferedWriter(Writer in);
```

Entre sus métodos más comunes se encuentran:

- **write():** Método utilizable para escribir un único caracter o una cadena de caracteres provenientes de un arreglo o de un *String*.
- **newLine():** Escribe un salto de línea establecido por la propiedad del sistema *line.separator*. Es importante señalar que no será necesariamente un separador de la forma (`\n`).
- **append(CharSequence csq, int start, int end):** Método utilizado para escribir una secuencia de caracteres definidos por una posición inicial y una posición final.
- **close():** Utilizado para cerrar el flujo de escritura de caracteres.

## 4. Manipulación básica de archivos

A continuación, se ha creado un proyecto en **NetBeans** con el fin de ejemplificar las operaciones básicas que pueden realizarse con un archivo de texto plano. En la clase principal del proyecto, se ha implementado un menú que permite seleccionar 4 operaciones distintas sobre un archivo así como la opción de terminar la ejecución del programa.

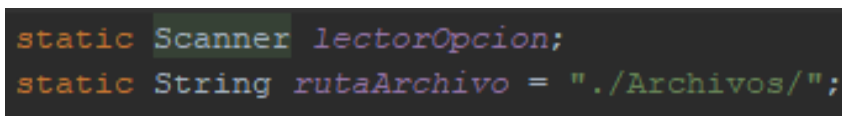


```
Output - Practica10[TellezJorge] (run) × Usages Test Results
run:
Estructura de Datos y Algoritmos II - Practica #10: Archivos.

¿Que actividad desea realizar?
1) Creacion de un archivo.
2) Sobreescritura de un archivo.
3) Edicion de un archivo.
4) Eliminacion de un archivo.
5) Salir del programa
```

Figura 4: Menú principal del programa.

Los archivos que serán generados para los propósitos de esta práctica se encontrarán en el directorio *Archivos* contenido en la carpeta del proyecto. El atributo *rutaArchivo* será utilizado para tener un acceso rápido a la ubicación de la carpeta al momento de manipular los objetos *File*.



```
static Scanner lectorOpcion;
static String rutaArchivo = "./Archivos/";
```

Figura 5: Atributos estáticos de la clase principal.

Por otra parte es importante destacar que, al momento de realizar procesos de I/O, el código escrito debe de estar contenido en un bloque **try** con su respectivo **catch** con el fin de detectar cualquier excep-

ción relacionada a estos procesos y que la ejecución del programa no termine abruptamente en caso de presentarse una.

#### 4.1. Creación de un archivo

La clase *CrearArchivo* implementa en sus atributos un objeto *Scanner* utilizado con el propósito de obtener el nombre del archivo que se desea generar. El único método de esta clase se denomina **genFile()** y durante su ejecución solicita el nombre del archivo que se desea generar (sin incluir la extensión .txt).

A continuación, se genera un objeto *File* a partir del nombre ingresado. Si resulta que el archivo creado ya existe en la carpeta, se envía un mensaje en pantalla indicando lo anterior y se terminará la ejecución del método.

```
String nombreDelArchivo = lectorCadena.nextLine();
File archivoCreado = new File(Practical0TellezJorge.rutaArchivo + nombreDelArchivo + ".txt");

if (archivoCreado.exists()) {
    System.out.println("\nYa existe un archivo con el nombre indicado!\n");
}
```

Figura 6: Generación de la referencia al archivo deseado.

En caso contrario, se utiliza el método **createNewFile()** para generar el archivo en carpeta y, posteriormente, se solicitará que se escriba contenido en el mismo; el cuál será escrito por medio del método **write()** de *BufferedWriter*. Finalmente, se imprime un mensaje de estado indicando que la operación ha sido exitosa.

```
Opción: 1

Usted ha seleccionado: Creacion de un archivo.

Ingrese el nombre del archivo que desea crear: archivoDePrueba
A continuacion, ingrese contenido en el archivo:
Este es un archivo de prueba!

Operacion realizada con exito.
```

Figura 7: Ejecución del método **genFile()**.



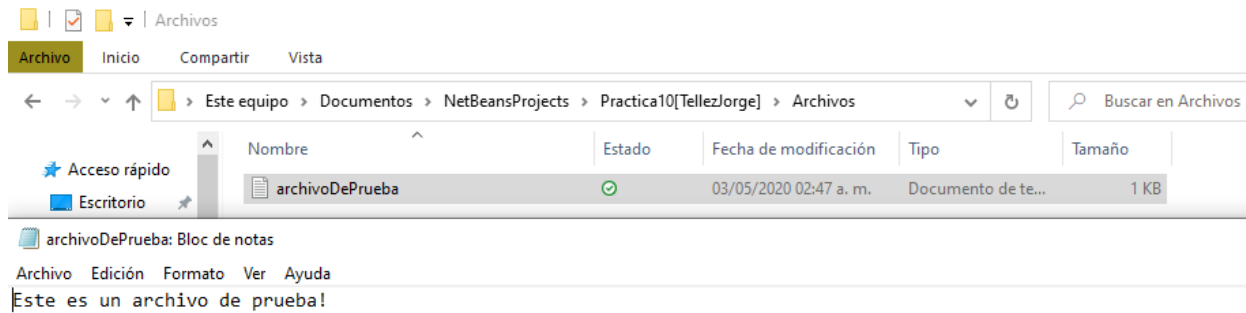


Figura 8: Archivo generado con la cadena escrita en el programa.

## 4.2. Sobreescritura de un archivo

La clase *SobreescrituraArchivo* tiene un *Scanner* inicializado como atributo como en la clase anterior. El método **overwriteFile()** que contiene solicita el nombre del archivo a sobrescribir y genera un objeto *File* de forma semejante a la clase *CrearArchivo*. La diferencia radica en que, si el archivo existe (verificándose lo anterior por medio del método **exists()**), se solicita que se ingrese el nuevo contenido para el archivo especificado.

Otra diferencia clave reside en la creación del objeto *FileWriter* involucrado en la creación del objeto de tipo *BufferedWriter*: añadiendo el valor booleano *false* como segundo argumento del constructor de *FileWriter* el flujo de caracteres será escrito al inicio del archivo; sobrescribiendo de forma efectiva el contenido anterior del archivo.

```
try (BufferedWriter bufferEscriotor = new BufferedWriter(new FileWriter(archivoTarget, false))) {  
    bufferEscriotor.write(lectorCadena.nextLine());  
    System.out.println("\nOperacion realizada con exito.\n");  
    bufferEscriotor.close();  
}
```

Figura 9: Modificación realizada con el fin de sobrescribir el archivo.

```
Opción: 2  
  
Usted ha seleccionado: Sobreescritura de un archivo.  
  
Ingrese el nombre del archivo que desea sobrescribir: archivoDePrueba  
A continuacion, ingrese el nuevo contenido en el archivo:  
Esta es una sobreescritura del archivo anterior!  
  
Operacion realizada con exito.
```

Figura 10: Ejecución del método **overwriteFile()**.

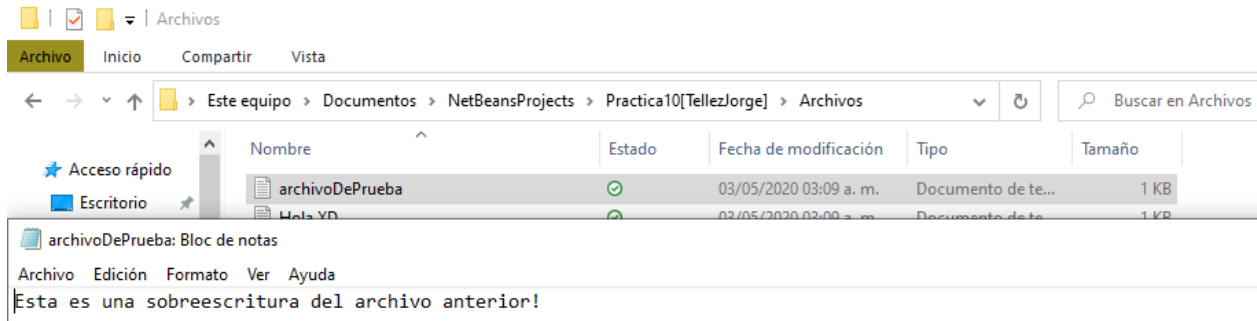


Figura 11: *Sobrescritura realizada en el archivo generado en el ejemplo de [8].*

### 4.3. Edición de un archivo

La clase *EdicionArchivo* contiene un *Scanner* inicializado como atributo. El método **modifyFile()** que contiene solicita el nombre del archivo a modificar y genera un objeto *File* de forma semejante a las clases anteriores.

Aunque su estructura es prácticamente idéntica a la de *SobrescrituraArchivo*, en esta clase se introduce como parámetro del objeto *FileWriter* creado el valor booleano *true*; de esta forma el contenido que sea escrito en el archivo será colocado al final del mismo sin eliminar o sobrescribir el contenido que ya estuviera presente en el mismo.

```
try (BufferedWriter bufferEscriotor = new BufferedWriter(new FileWriter(archivoTarget, true))) {
    bufferEscriotor.write(lectorCadena.nextLine());
    System.out.println("\nOperacion realizada con exito.\n");
    bufferEscriotor.close();
}
```

Figura 12: *Modificación realizada con el fin de introducir nuevo contenido sin eliminar el anterior.*

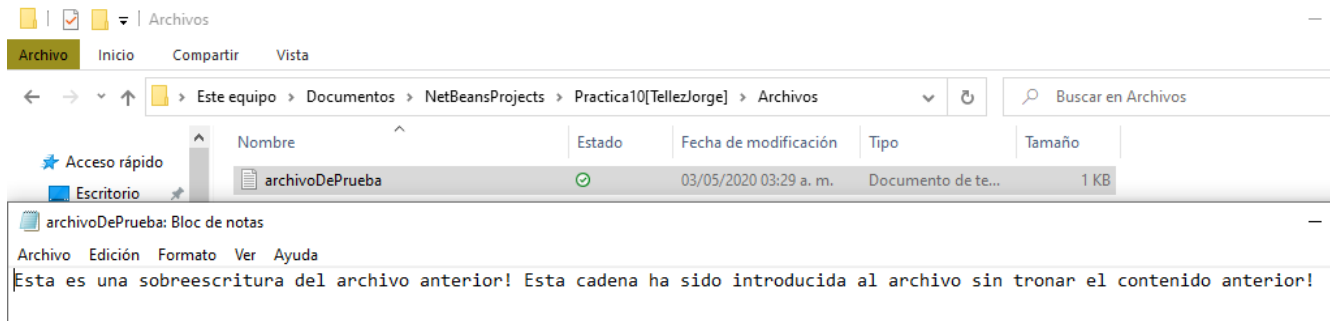
```
Opción: 3

Usted ha seleccionado: Edicion de un archivo.

Ingrese el nombre del archivo que desea editar: archivoDePrueba
A continuacion, ingrese contenido en el archivo:
Esta cadena de texto ha sido añadida al archivo sin tronar el contenido anterior!

Operacion realizada con exito.
```

Figura 13: *Ejecución del método **modifyFile()**.*

Figura 14: *Modificación realizada al archivo.*

#### 4.4. Eliminación de un archivo

La clase *EliminacionArchivo* contiene un *Scanner* inicializado como atributo. El método **deleteFile()** que contiene solicita el nombre del archivo a eliminar y genera un objeto *File* utilizado como referencia al mismo. En caso de que el archivo exista en la carpeta *Archivos* (verificándolo con el método **exists()**), se ejecuta el método **delete()** sobre el objeto *File* creado y se imprime un mensaje de estado en pantalla.

```
if (archivoAEliminar.exists()) {  
  
    archivoAEliminar.delete();  
    System.out.println("\nEl archivo: " + nombreDelArchivo + " ha sido eliminado."  
        + "\nOperacion realizada con exito.\n");  
}
```

Figura 15: *Eliminación del archivo seleccionado.*

```
Opción: 4  
  
Usted ha seleccionado: Eliminacion de un archivo.  
  
Ingresa el nombre del archivo que desea eliminar: archivoDePrueba  
  
El archivo: archivoDePrueba ha sido eliminado.  
Operacion realizada con exito.
```

Figura 16: *Ejecución del método **deleteFile()**.*

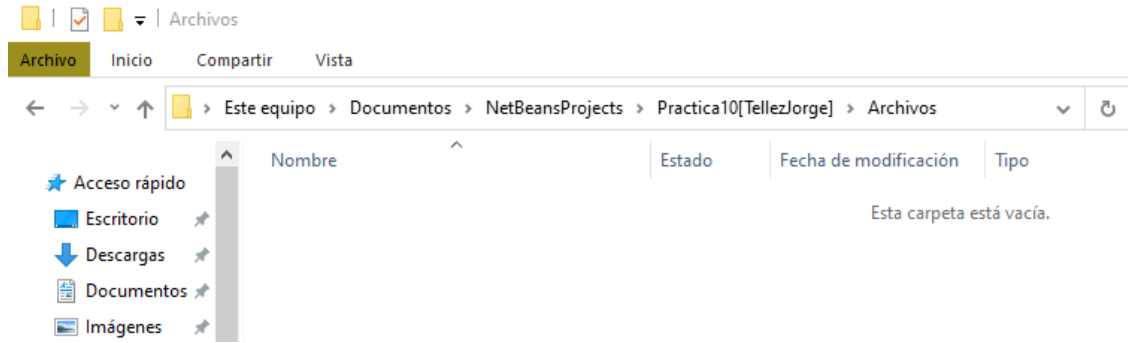


Figura 17: Verificación del resultado en la carpeta Archivos.

## 5. Conclusiones

Gran parte de los métodos utilizados en el trabajo desarrollado ya habían sido abordados con anterioridad en el desarrollo del *Proyecto 1* del curso. En general, el manejo de archivos en **Java** puede llegar a resultar una experiencia tediosa en un inicio. Sin embargo, con la experiencia obtenida esta dificultad inicial se ha reducido notablemente y las operaciones solicitadas han sido implementadas con éxito. En general, pueden destacarse los siguiente puntos respecto al desarrollo de esta práctica:

- La programación de los ejercicios fue sencilla y no presentó mayores complicaciones.
- La consulta de la documentación para las clases involucradas con archivos es muy importante para evitar cometer errores de implementación.
- Aunque pueda parecer un detalle menor, resulta crítico en determinadas situaciones cerrar los flujos de I/O; lección aprendida en el *Proyecto 1* y que fue rescatada en el trabajo desarrollado.

El manejo adecuado de los archivos y de los procesos I/O es una herramienta indispensable al momento de realizar programas complejos que requieran información externa para su ejecución. Cabe señalar que es una herramienta, a su vez, potencialmente peligrosa en caso de estar implementada incorrectamente debido al potencial peligro de alterar o eliminar otros archivos ajenos al programa.

Finalmente, con base en el trabajo desarrollado y en el éxito de las soluciones implementadas, puedo afirmar que los objetivos de esta práctica han sido cumplidos con éxito. La experiencia obtenida en el manejo de archivos fue exitosamente aplicada en esta práctica y tengo certeza de que seguirá siendo de utilidad durante el resto del curso, en próximos cursos de la carrera o en un entorno laboral que requiera tales habilidades.



## Referencias

- [1] Class BufferedReader. Recuperado de: <https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>. Fecha de consulta: 02/05/2020.
- [2] Class BufferedWriter. Recuperado de: <https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html>. Fecha de consulta: 02/05/2020.
- [3] Class FileReader. Recuperado de: <https://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html>. Fecha de consulta: 02/05/2020.
- [4] Class FileWriter. Recuperado de: <https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html>. Fecha de consulta: 02/05/2020.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©