Apunte 11 – Procesamiento de Archivos CSV

Introducción

En este apunte vamos a explorar cómo trabajar con archivos CSV en Java. Nuestro objetivo no es seguir un paso a paso de implementación, sino comprender los conceptos, herramientas y APIs involucradas. Queremos que, cuando después sigamos un tutorial práctico, tengamos claras las bases conceptuales que justifican cada decisión.

Los archivos CSV son un formato de texto simple, donde los datos se organizan en filas y columnas, separados por un delimitador (generalmente `,` o `;`). Esta simplicidad los hace muy usados para intercambio de datos entre sistemas, aunque también implica ciertos desafíos como el manejo de comillas, espacios en blanco y caracteres especiales.

Si bien este tema es más bien una aplicación de lo que hemos visto hasta aquí que un apunte conceptual, lo agregamos en este punto dada la importancia que tiene en el contenido de la asignatura. En el parcial individual tendremos que llevar a cabo el proceso de un archivo CSV para cargar en una base de datos las instancias asociadas a este archivo y por lo tanto será necesario revisar este mecanismo y tenerlo claro.

Lectura básica con `Scanner` y `split`

Empezamos con la forma más elemental de procesar un archivo CSV: leerlo línea por línea y dividir cada línea en columnas usando un delimitador.

Herramientas:

- `java.util.Scanner` para iterar sobre las líneas del archivo.
- `String.split(";")` para dividir las cadenas.

Ambas clases (`Scanner` y `String`) forman parte del JDK y nos permiten construir una primera solución sin dependencias externas.

```
Path csv = Path.of("personas.csv");
try (Scanner sc = new Scanner(Files.newBufferedReader(csv))) {
    while (sc.hasNextLine()) {
        String[] columnas = sc.nextLine().split(";");
        // columnas[0] → documento
        // columnas[1] → nombre
        // columnas[2] → apellido
        // columnas[3] → edad
    }
}
```

> Aquí recorremos línea por línea, separamos por `;` o `,` y obtenemos un array de `String` que luego podemos mapear a objetos.

Como vimos en el apunte anterior una forma elemental de realizar este mapeo es agregar a la clase del objeto a construir un constructor que reciba un array de strings y realice la asignación de los datos a los atributos, aunque también podemos optar por mecanismos más sofisticados como usar alguna librería de mapeo.

Lectura robusta con `BufferedReader`

Cuando necesitamos mayor control (charset, excepciones, cierres automáticos), `BufferedReader` es una alternativa más explícita.

Herramientas:

- `java.nio.file.Files.newBufferedReader`
- `java.io.BufferedReader`

```
Path archivo = Path.of("personas.csv");
try (BufferedReader br = Files.newBufferedReader(archivo,
StandardCharsets.UTF_8)) {
    String linea;
    while ((linea = br.readLine()) != null) {
        String[] columnas = linea.split(";");
        // procesar columnas...
    }
}
```

> El `try-with-resources` asegura que el archivo se cierre. Controlamos directamente el charset (`UTF-8`) y evitamos problemas con acentos o caracteres especiales.

Uso de OpenCSV con `CSVReader`

Llegado un punto, manejar manualmente los delimitadores, comillas y escapes se vuelve tedioso. Aquí entra en juego una librería externa: OpenCSV.

Herramientas:

- Dependencia externa: `com.opencsv.CSVReader`.
- Capacidad de manejar delimitadores, comillas y escapes automáticamente.

Maven:

En el archivo `pom.xml` deberemos agregar la dependencia a la librería OpenCSV

```
<dependency>
  <groupId>com.opencsv</groupId>
  <artifactId>opencsv</artifactId>
  <version>5.9</version>
</dependency>
```

Y luego utilizando la librería podemos trabajar de la siguiente manera:

```
try (CSVReader reader = new
CSVReader(Files.newBufferedReader(Path.of("personas.csv")))) {
     String\[] fila;
     while ((fila = reader.readNext()) != null) {
         // fila\[0] → documento
         // fila\[1] → nombre
         // fila\[2] → apellido
         // fila\[3] → edad
     }

}
```

> Usamos `CSVReader` para leer filas completas ya separadas y limpias, evitando lidiar con casos problemáticos manualmente.

OpenCSV con `CSVReaderHeaderAware`

A veces preferimos trabajar por nombres de columnas en lugar de índices, para hacer el código más legible y menos dependiente del orden. Este proceso depende que el archivo CSV tenga una primera línea que en general llamamos encabezado con los nombres de cada una de las columnas.

Herramientas:

- `CSVReaderHeaderAware` de OpenCSV.

```
try (CSVReaderHeaderAware reader = new
CSVReaderHeaderAware(Files.newBufferedReader(Path.of("personas.csv")))) {
    Map<String, String> fila;
    while ((fila = reader.readMap()) != null) {
        String nombre = fila.get("nombre");
        String apellido = fila.get("apellido");
    }
}
```

Cada fila se transforma en un `Map<String,String>` donde la clave es el nombre de la columna y el valor el dato correspondiente.

`CsvToBean` + POJOs con Lombok

Contexto: Para un diseño más orientado a objetos, podemos mapear cada fila a una clase Java. Esto nos permite trabajar con objetos fuertemente tipados.

Herramientas:

- `CsvToBean` de OpenCSV.

- Anotaciones de OpenCSV (`@CsvBindByName`).
- Lombok para reducir boilerplate (`@Data`, `@NoArgsConstructor`).

```
@Data
@NoArgsConstructor
public class Persona {
    @CsvBindByName(column = "documento") private Integer documento;
    @CsvBindByName(column = "nombre")    private String nombre;
    @CsvBindByName(column = "apellido")  private String apellido;
    @CsvBindByName(column = "edad")      private Integer edad;
}

try (Reader r = Files.newBufferedReader(Path.of("personas.csv"))) {
    List<Persona> personas = new CsvToBeanBuilder<Persona>(r)
        .withType(Persona.class)
        .withSeparator(';')
        .withIgnoreLeadingWhiteSpace(true)
        .build()
        .parse();
}
```

OpenCSV crea objetos `Persona` automáticamente a partir de los datos del archivo, vinculando columnas con atributos gracias a las anotaciones.

Anexo – Procesamiento con Streams y Collectors

Una vez que tenemos los objetos `Persona` cargados desde un CSV, podemos aprovechar la API de Streams de Java para realizar consultas y transformaciones de forma declarativa.

Ejemplos:

1. **Promedio de edad de las personas**

```
double promedioEdad = personas.stream()
    .mapToInt(Persona::getEdad)
    .average()
    .orElse(0);
```

2. **Conteo de personas por apellido**

```
Map<String, Long> conteoPorApellido = personas.stream()
    .collect(Collectors.groupingBy(Persona::getApellido,
Collectors.counting()));
```

3. **Top 3 personas más grandes por edad**

```
List<Persona> top3 = personas.stream()
    .sorted(Comparator.comparing(Persona::getEdad).reversed())
    .limit(3)
    .toList();
```

4. **Generación de un string con todos los nombres**

```
String nombres = personas.stream()
    .map(Persona::getNombre)
    .collect(Collectors.joining(", "));
```

> Con esto vemos cómo el procesamiento de CSV se integra naturalmente con la programación funcional: leemos los datos, los convertimos en objetos y luego usamos Streams y Collectors para obtener información útil de manera simple y expresiva.

Anexo – Procesamiento declarativo con Streams & Collectors sobre CSV

Contexto. Una vez que mapeamos el CSV a `List<Persona>`, queremos explotar la API de Streams para consultas y transformaciones **declarativas** (sin escribir bucles manuales). Nuestro objetivo aquí es entender **qué** resuelve cada operación y **por qué** la elegimos, más que hacer un paso a paso.

Herramientas. Usamos `java.util.stream.*` (Streams), `java.util.Comparator` (ordenamientos), y `java.util.stream.Collectors` (recolecciones a listas, mapas, estadísticas, etc.).

Todo lo que mostramos a continuación forma parte del JDK (no dependemos de librerías externas), y se apoya en el diseño visto en el apunte anterior de Programación Funcional y Streams.

Escenario base

Partimos de una lista ya construida con OpenCSV:

```
// Suponemos la clase Persona del apunte
@Data
@NoArgsConstructor
public class Persona {
    @CsvBindByName(column = "documento") private Integer documento;
    @CsvBindByName(column = "nombre")    private String  nombre;
    @CsvBindByName(column = "apellido")  private String  apellido;
    @CsvBindByName(column = "edad")      private Integer edad;
}

List<Persona> personas;
try (Reader r = Files.newBufferedReader(Path.of("personas.csv"))) {
    personas = new CsvToBeanBuilder<Persona>(r)
        .withType(Persona.class)
        .withSeparator(';')
        .withIgnoreLeadingWhiteSpace(true)
        .build()
```

```
            .parse();
    }
```

Estadísticas y métricas rápidas

Promedio de edades (usamos stream primitivo para evitar boxing):

```
double promedio = personas.stream()
    .mapToInt(Persona::getEdad)
    .average()
    .orElse(0);
```

Resumen completo (`count`, `min`, `max`, `sum`, `avg`) con `summaryStatistics`:

```
IntSummaryStatistics stats = personas.stream()
    .mapToInt(Persona::getEdad)
    .summaryStatistics();
// stats.getCount(), stats.getMin(), stats.getAverage(), stats.getMax(),
stats.getSum()
```

Selecciones y ordenamientos

Top 3 por edad (descendente):

```
List<Persona> top3 = personas.stream()
    .sorted(Comparator.comparing(Persona::getEdad).reversed())
    .limit(3)
    .toList();
```

Filtrar y proyectar solo nombres de mayores de 21:

```
List<String> nombresMayores = personas.stream()
    .filter(p -> p.getEdad() != null && p.getEdad() > 21)
    .map(Persona::getNombre)
    .toList();
```

Transformaciones a Map y manejo de duplicados

Documento → Persona (con estrategia de merge):

```
Map<Integer, Persona> porDocumento = personas.stream()
    .filter(p -> p.getDocumento() != null)
```

```
    .collect(Collectors.toMap(
        Persona::getDocumento,
        p -> p,
        // Si hay documentos duplicados, nos quedamos con la persona de
mayor edad
        (p1, p2) -> (p1.getEdad() >= p2.getEdad() ? p1 : p2),
        LinkedHashMap::new // preserva orden de aparición
    ));
```

Apellido → lista de nombres (downstream `mapping`):

```
Map<String, List<String>> nombresPorApellido = personas.stream()
    .collect(Collectors.groupingBy(
        Persona::getApellido,
        Collectors.mapping(Persona::getNombre, Collectors.toList())
    ));
```

Conteos y agregaciones por clave

Cantidad por apellido:

```
Map<String, Long> conteoPorApellido = personas.stream()
    .collect(Collectors.groupingBy(Persona::getApellido,
Collectors.counting()));
```

Promedio de edad por apellido:

```
Map<String, Double> promedioEdadPorApellido = personas.stream()
    .collect(Collectors.groupingBy(
        Persona::getApellido,
        Collectors.averagingInt(Persona::getEdad)
    ));
```

Join de nombres (string final):

```
String listado = personas.stream()
    .map(Persona::getNombre)
    .collect(Collectors.joining(", ", "[", "]"));
```

Buenas prácticas aplicadas al CSV

- **`try-with-resources`** para `Reader`/`BufferedReader`/`Lines`: evitamos dejar archivos abiertos.

- **Validaciones y trim**: antes de mapear/convertir, conviene `trim()` y controlar nulos para evitar `NumberFormatException`. -* **Evitar side-effects** dentro del pipeline: preferimos recolectar (`collect`) a estructuras nuevas en vez de mutar listas externas.
- **Elegir la estructura correcta**: `List`, `Set`, `LinkedHashMap` según orden/duplicados que necesitemos.
- **No sobreusar Streams**: si un `for` simple es más claro, lo elegimos. Streams suman cuando mejoran **legibilidad** y **expresividad**.

Reflexión final

En este recorrido vimos cómo podemos ir desde la solución más simple (leer líneas y dividir con `split`) hasta un enfoque más robusto y orientado a objetos usando OpenCSV y Lombok.

No pretendemos que cada alumno memorice todas las variantes, sino que comprendamos qué herramientas tenemos disponibles y cuándo conviene aplicar cada una. Más adelante, en un paso a paso práctico, podremos ejercitar estos conceptos en ejemplos concretos y ver cómo aprovechar Streams y Collectors para procesar la información leída.