

Programación Orientada a Objetos

Archivos, E/S y persistencia

`java.nio.file`, Decorator, CSV y serialización

Gerson Garrido

Semana 15

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Rutas y archivos con `Path` y `Files`; errores de E/S
2. Codificación: bytes y caracteres
3. Streams de `java.io` y el patrón **Decorator**
4. Persistencia: un repositorio, CSV a mano y pruebas de ida y vuelta
5. Serialización de Java, y por qué no con datos de fuera
6. Un menú de línea de comandos con `Scanner`: el arranque de P5

> **Lectura**
>
> Eck, secs. 11.1, 11.2.1, 11.3.2 y 11.3.3. dev.java, *The Java I/O API*: *Working with Paths*, *Reading and Writing Small Files*, *Decorating I/O Streams*.

SECCIÓN 01

Archivos

01



Programas que recuerdan

- Hasta ahora todo vivía en el heap y desaparecía al terminar el programa.
- Un programa de verdad tiene que recordar entre una ejecución y otra: persistir.
- La semana pasada `LineImporter` leía de una `List<String>`. ¿Y si las líneas vienen de un archivo?

Dos APIs

`java.nio.file` (`Path`, `Files`, Java 7): la moderna, para el día a día. `java.io` (streams): la de siempre, que está por debajo y organiza todo con un patrón.

Rutas: `Path`

FilesDemo.java

```java
Path dir = Path.of("build", "demo");
Path file = dir.resolve("notas.txt");
IO.println("ruta: " + file
    + "  nombre: " + file.getFileName()
    + "  padre: " + file.getParent());
IO.println("¿absoluta? " + file.isAbsolute()
    + "  ¿existe? " + Files.exists(file));
```

bash

```
ruta: build/demo/notas.txt  nombre: notas.
    txt  padre: build/demo
¿absoluta? false  ¿existe? false
```

- Un `Path` es un nombre: crearlo no abre ni crea nada.
- `Path.of` usa el separador del sistema (`/` o `\`).
- Una ruta **relativa** se interpreta desde el directorio de trabajo: donde se lanzó `java`. En el contenedor, `/work/code/week-15`.

Error común: “el archivo desapareció”

Error común

Probar desde el editor, que lanza el programa en la carpeta del proyecto, y después desde la terminal en otra carpeta. Con rutas relativas, el archivo **depende de dónde se ejecuta** el programa.

Cuando importa, imprimir la ruta completa: `file.toAbsolutePath()`.

Archivos pequeños: todo de una vez

Método	Qué hace
`Files.readString(path)`	Todo el archivo en un `String`
`Files.readAllLines(path)`	Una `List<String>`, una línea por elemento
`Files.writeString(path, texto)`	Escribe un `String`: crea o reemplaza
`Files.write(path, líneas)`	Escribe una lista de líneas
`Files.exists`, `Files.size`	Preguntas sobre el archivo
`Files.createDirectories(path)`	Crea la carpeta y las que falten

Con `StandardOpenOption.APPEND`, `write` agrega al final en lugar de reemplazar.

Ejemplo: escribir, agregar, leer

FilesDemo.java

```java
Files.createDirectories(dir);
Files.write(file, List.of("Lima, 18 °C",
        "Cusco, 9 °C", "Iquitos, 31 °C"));
List<String> lines = Files.readAllLines(file);
IO.println("leídas " + lines.size()
        + " líneas; la segunda es \""
        + lines.get(1) + "\"");
Files.writeString(file, "Puno, 4 °C\n",
        StandardOpenOption.APPEND);
IO.print(Files.readString(file));
```

bash

```
leídas 3 líneas; la
    segunda es "Cusco, 9
    °C"
Lima, 18 °C
Cusco, 9 °C
Iquitos, 31 °C
Puno, 4 °C
```

Archivos grandes: línea por línea

FilesDemo.java

```java
try (var in = Files.newBufferedReader(file)) {
    String line;
    int n = 0;
    while ((line = in.readLine()) != null) {
        if (line.contains("°")) { n++; }
    }
}
try (Stream<String> s = Files.lines(file)) {
    IO.println(s.filter(l -> l.startsWith("C"))
                .toList());
}
```

- No se carga todo en memoria: se lee de a poco.
- `readLine()` devuelve `null` al final del archivo.
- `Files.lines`: un stream de la semana 12.
- Los dos dejan el archivo abierto: van en un `try`-with-resources (semana 9).

Errores de E/S

- Casi todo `Files` declara `throws IOException`: checked.
- Es justo: un disco lleno o un archivo borrado pasan aunque el programa esté bien escrito.
- ¿Dónde se capturan? En el borde del programa (el `main`, el menú), donde se le puede hablar al usuario. No en el modelo.

Excepción	Cuándo
`NoSuchFile-Exception`	No existe
`AccessDenied-Exception`	Sin permiso
`FileAlreadyExists-Exception`	Ya existe
`MalformedInput-Exception`	Mala codificación

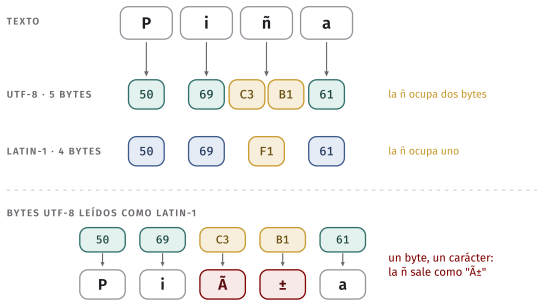
SECCIÓN 02

Codificación

02

Un archivo guarda bytes, no letras

- La **codificación** *(charset)* dice cómo se convierten las letras en bytes.
- **UTF-8**: la de la web. ñ, é, ° ocupan dos bytes.
- **Latin-1** y Windows-1252: un byte por letra, solo 256. Aún aparecen en archivos de Excel en Windows.



Leer con la codificación equivocada

FilesDemo.java · fragmento

```java
Files.writeString(latin, "Piña, café", StandardCharsets.ISO_8859_1);
Files.readString(latin);                        // Files usa UTF-8 por defecto
Files.readString(latin, StandardCharsets.ISO_8859_1);
Files.readString(utf8, StandardCharsets.ISO_8859_1);  // escrito en UTF-8
```

bash · salida, con los textos de FilesDemo

```
bytes en Latin-1: 10
bytes en UTF-8:   12
leído como UTF-8: java.nio.charset.MalformedInputException: Input length = 4
leído como Latin-1: Piña, café
UTF-8 leído como Latin-1: PiÃ±a, cafÃ©
```

En Java: UTF-8 por defecto

- Desde **Java 18** (JEP 400) la codificación por defecto es UTF-8 en todos los sistemas. La usan las API que no reciben codificación, como `FileReader` o `InputStreamReader`.
- Antes dependía del sistema: un programa que andaba en Linux rompía las tildes en Windows.
- Los métodos de texto de `Files` son otra regla, más antigua: sin codificación, su documentación dice UTF-8, desde que existen.
- Igual conviene escribir `StandardCharsets.UTF_8` cuando el archivo se comparte: la decisión queda a la vista.

SECCIÓN 03

Streams y Decorator

03



Streams de `java.io`

	Bytes	Caracteres
Entrada	`InputStream`	`Reader`
Salida	`OutputStream`	`Writer`

No son los streams de la semana 12: mismo nombre, otra idea.

- Un **stream** de E/S: datos que se leen o escriben en orden, de a poco.
- Cada clase concreta agrega una capacidad.
- Para tener varias, se envuelve un objeto dentro de otro.

Tres capas para leer líneas

DecoratorDemo.java

```java
try (InputStream bytes =
         Files.newInputStream(file);
     InputStreamReader chars =
         new InputStreamReader(bytes, UTF_8);
     BufferedReader lines =
         new BufferedReader(chars)) {
    String line;
    while ((line = lines.readLine()) != null) {
        IO.println("> " + line);
    }
}
```

- `InputStream`: bytes del archivo.
- `InputStreamReader`: bytes a caracteres, con una codificación.
- `BufferedReader`: lee en bloques y ofrece `readLine()`.
- Es lo que hace por dentro `Files.newBufferedReader`. Y es un patrón con nombre.

Decorator: el problema

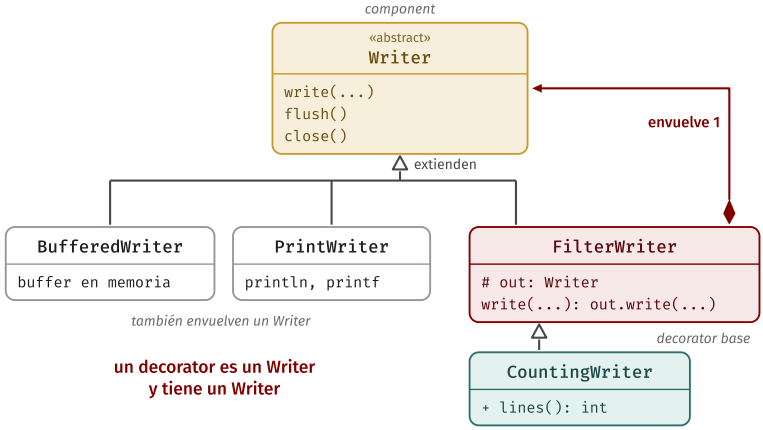
¿Y si `java.io` usara herencia?

- `FileReader`, `BufferedFileReader`, `CountingBufferedFileReader`...
- Y lo mismo para la red, la memoria, los archivos comprimidos...
- Cuatro orígenes y tres capacidades opcionales: una clase por combinación.

Decorator

Un objeto que implementa la misma interfaz que el que envuelve, hace lo suyo y le delega el resto. Como también es un `Writer`, se puede volver a envolver: las capacidades se combinan en tiempo de ejecución.

Estructura de Decorator en `java.io`



Un decorator propio: `CountingWriter`

CountingWriter.java · fragmento

```java
class CountingWriter extends FilterWriter {
  private int lines = 0;
  CountingWriter(Writer w) { super(w); }
  public void write(String s, int off, int len)
      throws IOException {
    for (int i = off; i < off + len; i++) {
      if (s.charAt(i) == '\n') { lines++; }
    }
    super.write(s, off, len);   // delega
  }
}
```

- `FilterWriter`: la base de `java.io` para decorators. Guarda el `Writer` envuelto y le delega todo.
- Se hace override solo de lo necesario (aquí, también `write(int)` y `write(char[],...)`).
- Agrega `lines()`.

Una cadena de tres decorators

DecoratorDemo.java

```java
var counter = new CountingWriter(
        Files.newBufferedWriter(file));
try (var out = new PrintWriter(counter)) {
    out.println("Informe del día");
    out.printf("%-10s %5d%n", "Lima", 18);
    out.printf("%-10s %5d%n", "Cusco", 9);
}
IO.println("líneas escritas: "
        + counter.lines());
```

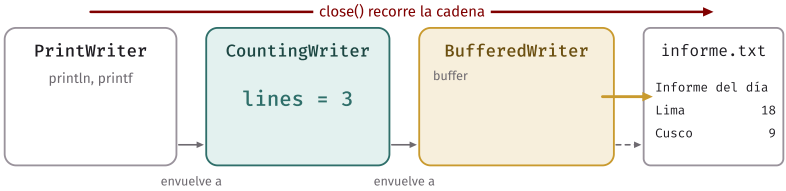
bash

```
líneas escritas: 3
```

- `PrintWriter` da `println` y `printf`.
- Se cierra solo el de afuera: cerrar un decorator cierra lo que envuelve.

Cada capa hace lo suyo

```
out.println("Informe del día");
out.printf("%-10s %5d%n", "Lima", 18);
out.printf("%-10s %5d%n", "Cusco", 9);
out.close();
```



4. close() vacía el buffer: recién ahora el archivo tiene el texto

Error común: no cerrar

DecoratorDemo.java

```java
var leaky = Files.newBufferedWriter(other);
leaky.write("esto todavía no está "
        + "en el disco\n");
IO.println("antes de close: "
        + Files.size(other) + " bytes");
leaky.close();
IO.println("después de close: "
        + Files.size(other) + " bytes");
```

bash

```
antes de close: 0 bytes
después de close: 35 bytes
```

Error común

Olvidar cerrar un archivo, o leerlo antes de cerrarlo. El buffer guarda lo escrito en memoria; sin `close`, el archivo queda **vacío o incompleto**.

Solución: `try`-with-resources.

Decorator, herencia y Strategy

- Con herencia: `CountingBufferedWriter`, `CountingPrintWriter`… Con Decorator, `CountingWriter` sirve con cualquier `Writer`, también con los que se escriban después.
- Es la composición de la semana 7: tener un objeto y delegarle, en lugar de ser su subclase.
- ¿Y Strategy? Una strategy es **una pieza** que el context usa. Un decorator **se hace pasar** por el objeto que envuelve: quien lo usa no nota que está ahí.

Cuándo no usar Decorator

- **Una capacidad para un solo caso**: un método o un objeto auxiliar es más claro.
- **Interfaces grandes**: el decorator delega cada método, aunque no le interese. Por eso existe `FilterWriter`.
- **El orden importa y nadie lo controla**: cifrar y comprimir dan resultados distintos según quién envuelve a quién. Que la cadena la arme un solo lugar.
- **Depurar**: el stack trace pasa por muchas capas antes de llegar a la que trabaja.

SECCIÓN 04

Persistencia en CSV

04

¿Dónde va el código que guarda?

- Tentación: `song.toCsvLine()`. Pero `Song` tendría dos razones para cambiar: las reglas de una canción y el formato del archivo (semana 13).
- Solución: un **repositorio** *(repository)*, detrás de una interfaz.
- El resto del programa no sabe si es CSV, binario o memoria. En las pruebas, un fake.

SongRepository.java

```java
interface SongRepository {
    List<Song> load()
            throws IOException;

    void save(List<Song> songs)
            throws IOException;
}
```

CSV: fácil hasta que no

- Una línea por objeto, campos separados por comas, una línea de **encabezado**.
- Escribir: `String.join(",", campos)`. Leer: `line.split(",")`.
- Hasta que una canción se llama `Lima, de noche`: cuatro campos.

un CSV ingenuo

```
Lima, de noche,Los Ensayos,205
```

Nuestra regla (la de RFC 4180)

- Un campo con coma o comillas va entre comillas.
- Las comillas de adentro se escriben dobles: `""`.
- Sin saltos de línea dentro de un campo (el estándar sí los permite).

Csv.join

Csv.java

```java
static String join(List<String> fields) {
  List<String> quoted = new ArrayList<>();
  for (String f : fields) {
    if (f.contains("\n") || f.contains("\r"))
      throw new IllegalArgumentException(...);
    if (f.contains(",") || f.contains("\"")) {
      f = "\"" + f.replace("\"", "\"\"") + "\"";
    }
    quoted.add(f);
  }
  return String.join(",", quoted);
}
```

- `Csv.split` hace lo contrario: recorre la línea recordando si está dentro de comillas.
- `CsvTest` fija los casos difíciles: comas, comillas, campos vacíos (`"a„"` son tres), saltos de línea, comillas sin cerrar.

`CsvSongRepository`: guardar

CsvSongRepository.java

```java
public void save(List<Song> songs)
        throws IOException {
    List<String> lines = new ArrayList<>();
    lines.add(HEADER); // title,artist,seconds
    for (Song s : songs) {
        lines.add(Csv.join(List.of(s.title(),
                s.artist(), "" + s.seconds())));
    }
    Files.write(file, lines, UTF_8);
}
```

- Se arman todas las líneas y se escriben de una vez.
- La codificación, a la vista.
- `Song` no sabe nada de CSV.

`CsvSongRepository`: cargar

CsvSongRepository.java · parse

```java
try {
  List<String> f = Csv.split(line);
  if (f.size() != 3) {
    throw new MalformedCsvException(n, ...);
  }
  return new Song(f.get(0), f.get(1),
          Integer.parseInt(f.get(2).strip()));
} catch (NumberFormatException e) {
  throw new MalformedCsvException(n,
          "la duración no es un número");
} // luego: catch (IllegalArgumentException e)
```

- `load` verifica el encabezado y lleva la cuenta del número de línea.
- `NumberFormatException` es subclase de `IllegalArgument-Exception`: va primero (semana 9).
- Las reglas de la canción siguen en `Song`; el repositorio solo agrega la línea.

Ejemplo: guardar, mirar, cargar

`RepositoryDemo` guarda tres canciones elegidas para probar la regla:

- una con comillas en el artista,
- una con una coma en el título.

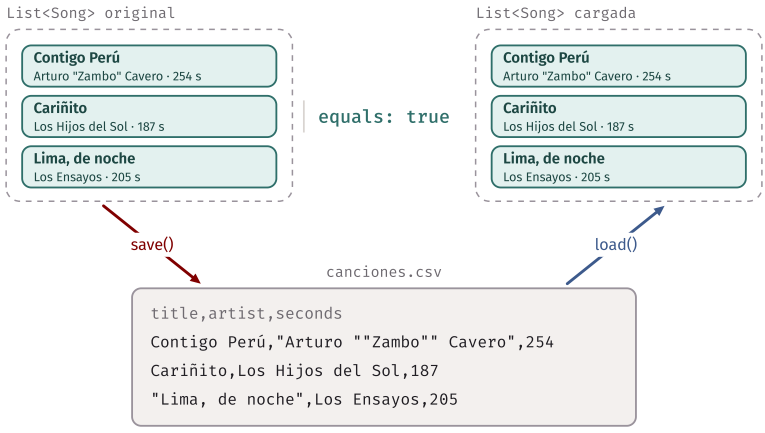
Muestra el archivo, lo vuelve a cargar y compara con `equals`.

Las líneas `[log]`: un decorator, en un momento.

bash

```
$ java -cp build RepositoryDemo
[log] save: 3 canciones
title,artist,seconds
Contigo Perú,"Arturo ""Zambo"" Cavero",254
Cariñito,Los Hijos del Sol,187
"Lima, de noche",Los Ensayos,205
[log] load: 3 canciones
¿iguales? true
```

Ida y vuelta



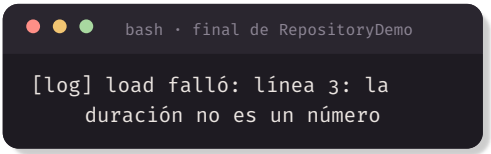
las comillas dobles existen solo en el archivo; los objetos no las tienen

Una línea mal formada

Dos políticas razonables:

1. **Rechazar el archivo** con una excepción que diga la línea. La nuestra: no se pierden datos sin que nadie se entere.
2. **Saltar la línea** y reportarla al final. Para importar datos de otros.

Lo que no es razonable: no decidir.



- `MalformedCsvException extends IOException`, con `line()`.
- "No pude leer" y "no lo entiendo": dos formas de que falle la carga.

Error común: la coma decimal

Error común

En el Perú se escribe `3,50`; en el código, `3.50`. Un CSV con precios con coma decimal confunde la coma del número con la del separador.

Regla del curso: los archivos guardan los números con punto, o mejor en céntimos, como enteros. La coma es solo para mostrar en pantalla.

Decorator otra vez: un repositorio que registra

LoggingSongRepository.java · fragmento

```java
class LoggingSongRepository
        implements SongRepository {
  private final SongRepository inner;
  private final Consumer<String> log;
  public void save(List<Song> songs)
      throws IOException {
    inner.save(songs);
    log.accept("save: " + songs.size()
        + " canciones");
  }
}
```

- Misma interfaz, guarda otro repositorio, le delega todo y registra.
- `load` registra también los errores y los relanza.
- El destino es un `Consumer<String>`: imprime en la demostración, guarda en una lista en las pruebas.
- Se prueba con un fake: un repositorio en memoria.

Pruebas de ida y vuelta con `@TempDir`

CsvSongRepositoryTest.java

```java
@TempDir Path dir;
@Test
void roundTripKeepsEverySong()
        throws IOException {
  List<Song> songs = List.of(...);
  var repo = new CsvSongRepository(
          dir.resolve("s.csv"));
  repo.save(songs);
  assertEquals(songs, repo.load());
}
```

- Guardar, cargar, comparar con `equals` (semana 5). Con el de `Object`, nunca serían iguales.
- JUnit crea una carpeta temporal nueva por prueba y la borra al final.
- Y pruebas de error: archivo inexistente, encabezado, campos, número.

SECCIÓN 05

Serialización

05



Serialización de Java

Playlist.java

```java
class Playlist implements Serializable {
  private static final long
      serialVersionUID = 1L;
  private final String name;
  private final ArrayList<Song> songs = ...;
  private transient int totalSeconds;
  // ...
}
```

- `Serializable` no tiene métodos: solo marca la clase.
- Se guarda el objeto con todo lo que alcanza.
- `serialVersionUID`: la versión. Si no coincide, `InvalidClassException`. Si no se declara, `-Xlint:all` avisa.
- `transient`: no se guarda; vuelve en su valor por defecto.

Ejemplo: guardar y cargar una lista

SerializationDemo.java · fragmento

```java
try (var out = new ObjectOutputStream(
        Files.newOutputStream(file))) {
    out.writeObject(list);
}
try (var in = new ObjectInputStream(
        Files.newInputStream(file))) {
    var back = (Playlist) in.readObject();
}
```

bash

```
antes:   2 canciones, 441 s
archivo: 290 bytes, empieza
    con ac ed 00 05
después: 2 canciones, 0 s (
    totalSeconds es
    transient)
¿mismas canciones? true
```

Cómodo: dos líneas para un grafo de objetos. Pero frágil ante cambios de la clase, binario, y peligroso con datos de fuera.

Bytes alterados

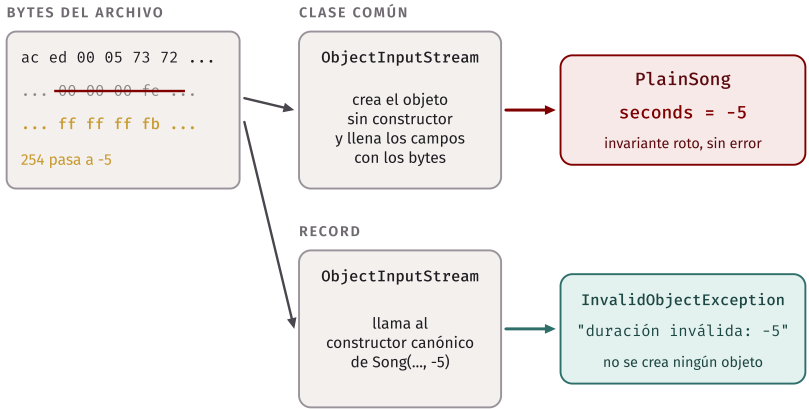
- Al deserializar, la JVM crea los objetos sin llamar a sus constructores: los campos salen de los bytes.
- La demostración cambia la duración 254 (`00 00 00 fe`) por −5 (`ff ff ff fb`) y deserializa:

```
bash · final de SerializationDemo

clase común: PlainSong[Contigo Perú, -5]
record:      java.io.InvalidObjectException: duración inválida: -5
```

- La clase común quedó con el invariante roto, sin error.
- Un **record** se reconstruye llamando a su constructor canónico: la validación corre.

Dos caminos para los mismos bytes



No deserializar datos que no son de confianza

- Un archivo de serialización puede nombrar cualquier clase serializable que el programa tenga a mano, y al reconstruirla se ejecuta código de esas clases.
- En programas con muchas bibliotecas se encontraron combinaciones que permiten a un atacante ejecutar código con solo enviar un archivo.
- Guía de Oracle, *Secure Coding Guidelines for Java SE*, sección 8: evitarlo.

En Java: `ObjectInputFilter` (Java 9)

El filtro `"Song;Playlist;java.util.ArrayList;!*"` acepta esas clases y rechaza todas las demás: `InvalidClassException` antes de crear nada.

CSV o serialización

	CSV	Serialización de Java
Código que hay que escribir	El formato, a mano	Casi nada
Se lee y edita a mano	Sí	No (binario)
Otros programas lo entienden	Sí	Solo Java
Si la clase cambia	Se adapta el lector	Puede quedar ilegible
Con datos de fuera	Se valida cada campo	Peligroso
Referencias compartidas	Difícil	Las guarda tal cual

Para el proyecto: CSV (o un formato de texto como JSON). La serialización, para casos internos.

SECCIÓN 06

Un menú con `Scanner`

06



Un menú de línea de comandos

PlaylistCli.java · run

```
1  while (true) {
2    out.print("> ");
3    if (!in.hasNextLine()) { break; }
4    String line = in.nextLine().strip();
5    // command: primera palabra; rest: lo demás
6    switch (command) {
7      case "listar" -> list();
8      case "agregar" -> add(rest);
9      case "guardar" -> save();
10     case "salir" -> { ...; return; }
11     default -> out.println("comando ...");
```

- Leer una línea, despachar, repetir.
- `Scanner` (Eck 2.4.6) y `switch` (Eck 3.6.2).
- `hasNextLine()` es `false` al final de la entrada (Ctrl+D, o un archivo que se acabó).
- El archivo llega en `args[0]` (Eck 4.3.6).

Pensado para probarse

- Entrada, salida y repositorio llegan por el constructor.
- En las pruebas: un `Scanner` sobre un `String`, un `PrintStream` sobre bytes, un fake del repositorio.
- Un comando mal escrito no detiene el programa.
- `make test` le da `menu-input.txt` como entrada.

bash · salida recortada

```
$ java -cp build PlaylistCli \
    build/demo/menu.csv < menu-
    input.txt
cargadas 3 canciones
...
> agregada: Valicha
> no se agregó: la duración no es
    un número
> comando desconocido: borrar
...
> guardadas 4 canciones
> hasta luego
```

Error común: la lógica en el `case`

Error común

Validar, calcular y guardar **dentro** de los `case` del menú. No se puede probar sin simular un usuario, ni reutilizar cuando aparezca otra interfaz.

Cada `case`: una llamada corta a un método del modelo o del repositorio. La semana 16 lleva esta idea más lejos: programación orientada a eventos.

SECCIÓN 07

Laboratorio

07

Laboratorio: guardar y cargar el estado

1. Un **repositorio** que guarde y cargue el inventario en CSV con `Files`, detrás de una interfaz.
2. Pruebas de **ida y vuelta** en `@TempDir`, sin tocar archivos reales.
3. Pruebas de **errores**: archivo inexistente, línea mal formada, número inválido.
4. Un **decorator** del repositorio que registre cada operación; probar que delega.
5. Arranque de P5: el **menú** con `Scanner` y `switch`.

`code/week-15`: `make test` corre cuatro demostraciones, el menú con `menu-input.txt` y 23 pruebas. **P4** se entrega hoy; **P5** se publica.

Resumen

- `Path` nombra; `Files` lee y escribe. Todo de una vez para lo pequeño; línea por línea, con `try`-with-resources, para lo grande. `IOException` se captura en el borde.
- Un archivo guarda bytes: la **codificación** los vuelve caracteres. UTF-8 por defecto desde Java 18; `Files`, siempre.
- Los streams de `java.io` se envuelven: **Decorator**. Misma interfaz, delega, se combina en tiempo de ejecución. Cerrar el de afuera cierra la cadena.
- Persistir va en un **repositorio** detrás de una interfaz. CSV: decidir la regla, probarla, y la prueba de **ida y vuelta**.
- **Serialización**: poco código, pero frágil, binaria e insegura con datos de fuera. Los records pasan por su constructor.
- Un menú: leer, despachar, repetir, con entrada y salida inyectadas.

Para la próxima clase

- Semana 16: programación orientada a eventos (sin interfaz gráfica) y el cierre del curso.
- Lecturas: Eck, sec. 1.2 *Asynchronous Events*; edición Swing, subsecs. 6.3.1 *Event Handling* y 6.4.4 *State Machines*.
- **P5**: patrones, persistencia e interfaz de línea de comandos. Se presenta en el laboratorio de la semana 16.

Pregunta para pensar

El menú es un bucle que espera una línea y reacciona. ¿En qué se parece a un programa con ventanas que espera un clic?