1. Refuerzo de Java

Anexo III. Más características y ejemplos de Java

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

1. Introducción

En este anexo se entrará en detalle sobre algunas características interesantes de Java, especialmente de las últimas versiones, como:

- Trabajar con fechas con *LocalDate* y algunas otras clases útiles.
- Nuevas interfaces funcionales añadidas en Java 8 que, junto con las expresiones lambda, permiten realizar algunas operaciones en unas pocas líneas de código.
- Nuevas funciones añadidas para trabajar con ficheros.
- Nuevas funciones para al trabajo con colecciones (reemplazo de elementos, fusión, etc).
- …

2. Trabajar con fechas

Antes de Java 7, la mayoría de las operaciones con fechas, se manejaban con la clase *Date*. Java 7 marcó como obsoletos (*deprecated*) la mayoría de sus métodos y obligó a trabajar con la clase ***Calendar*** (o ***GregorianCalendar***).

```
Date d = new Date(); // Represents NOW. All other constructors are deprecated
```

Todavía pueden utilizarse objetos *Date*, pero casi todos sus métodos están en desuso, por lo que la mayoría de las operaciones deberán realizarse con un objeto *Calendar*:

```
Calendar cal = Calendar.getInstance(); // NOW -> default locale and timezone
cal.set(2014, 1, 10); // 10th Feb. 2014 (January -> 0)
cal.add(Calendar.DAY_OF_MONTH, 7); // Add 7 days
Date weekLater = cal.getTime(); // Returns Date object
```

Desde Java 8, las clase del paquete ***java.time*** reemplazan las funciones de *Date*, *Time*, *TimeStamp* y *Calendar* (todavía admitidas por compatibilidad). El procesamiento de fechas y horas es mucho más flexible y se adapta mejor a las zonas horarias internacionales.

2.1. Tipos de fecha y hora

LocalDate es la nueva clase que reemplaza la antigua clase *Date* para trabajar con fechas. Se pueden crear fechas a partir de una marca de tiempo actual, o de un año, mes y día determinados:

```
LocalDate now = LocalDate.now();
LocalDate birth = LocalDate.of(1950, Month.JULY, 13);
```

La clase ***LocalTime*** representa la hora del día:

```
LocalTime now = LocalTime.now();
LocalTime bedTime = LocalTime.of(23, 0);
LocalTime wakeUp = bedTime.plusHours(8);
```

```
System.out.printf("I will wake up at %02d:%02d",
wakeUp.getHour(), wakeUp.getMinute());
```

La clase ***LocalDateTime*** tiene información sobre la fecha y la hora:

```
LocalDateTime dateTime = LocalDateTime.now();
```

2.1.1. Fechas y horas por zonas

También se puede trabajar con zonas horarias, es decir, marcas temporales que no están expresadas en nuestra ubicación actual, sino en cualquier otra ubicación del mundo. Por ejemplo, si estás en España, los objetos *LocalTime* obtendrán la hora según la zona horaria de España. Pero si utilizas objetos ***ZonedDateTime*** se puede obtener la fecha y/o hora de cualquier lugar del mundo.

La clase ***ZoneId*** proporciona identificadores para cada zona horaria admitida (https://www.iana.org/time-zones).

```
Set<String> zonesIds = ZoneId.getAvailableZoneIds();
ZoneId spainZone = ZoneId.of("Europe/Madrid");
```

Así es como se crearía una fecha y hora dividida por zonas:

```
ZonedDateTime zDate = ZonedDateTime.of(
        1969, Month.AUGUST.getValue(), 14, // year, month, day
        14, 25, 0, 0, // hour, min, sec, nanos
        ZoneId.of("Europe/Madrid"));
```

Se puede sumar, o restar, unidades a esta fecha, cambiar la zona horaria, etc.

```
ZonedDateTime date2 = zDate.plus(Period.ofMonths(5)); // Add 5 months
ZonedDateTime date3 = zDate.plus(15, ChronoUnit.DAYS);
ZonedDateTime dateUk = zDate.withZoneSameInstant(ZoneId.of("Europe/London"));
```

2.2. Periodos entre fechas y horas

Instant representa un momento (fecha y hora) con una precisión de nano segundos, y es inmutable. ***Duration*** es la cantidad de tiempo entre dos objetos *Instant*. Puedes obtener su valor en diferentes unidades, sumar o restar tiempo, etc.

```
Instant start = Instant.now();
// Some code
Instant end = Instant.now();
Duration dur = Duration.between(start, end);
long milliseconds = dur.toMillis();
```

Period es la cantidad de tiempo entre dos *LocalDate*. Es el mismo concepto que *Duration*.

```
LocalDate now = LocalDate.now();
LocalDate birth = LocalDate.of(1950, Month.JULY, 13);
Period age = birth.until(now);

System.out.println("Lived: " + age.getYears() + " years, " +
        age.getMonths() + " months, " + age.getDays() + " days.");
System.out.println("Total days lived: " + birth.until(now, ChronoUnit.DAYS));
```

2.3. Formato de fechas

Si quieres formatear un objeto fecha con un patrón dado, se puede hacer de la siguiente manera:

- Utilizando la clase ***DateTimeFormatter*** con un patrón personalizado:

```
// Will print 14/25/1969
System.out.println(DateTimeFormatter.ofPattern("d/m/Y").format(zonedDate));
```

- Utilizando formatos predefinidos de la clase ***DateTimeFormatter***:

```
// Will print Thu, 14 Aug 1969 14:25:00 +0100
System.out.println(DateTimeFormatter.RFC_1123_DATE_TIME.format(zonedDate));
```

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

Ejercicio 1

Crea un proyecto llamado ***OrderedComments*** y un fichero llamado ***comments.txt*** que contendrá comentarios de usuario (no los ordenes por fecha) separados por punto y coma ***';'*** en este formato:

```
username;comment;dd/mm/yyyy;hour:min:sec;timezone
```

Por ejemplo: `peter;Hello everybody;23/09/2013;10:04:54;Europe/London`

Crea una clase llamada ***Comment*** que contendrá el nombre de usuario, el comentario y la fecha *ZonedDateTime*.

En el método ***main***, deberás leer el fichero y crear un objeto *Comment* por cada comentario (puedes utilizar el método split() de *String* para dividir campos), cambia la zona horaria de la fecha de cada comentario a "Europa/Madrid" (utiliza el método ***withZoneSameInstant()***), y añade los comentarios a una lista. Después, ordena la lista por la fecha de los comentarios utilizando un *Comparator*. Para comparar una fecha con otra (en segundos), puede usar:

```
date1.until(date2, ChronoUnit.SECONDS) // "date2 – date1" in seconds
```

Finalmente, crea un nuevo fichero llamado ***Order_comments.txt*** y guarda todos los comentarios en el mismo formato que antes, pero esta vez ordenados por fecha y en la misma zona horaria (Madrid). Para convertir un *ZonedDateTime* al formato de fecha en el fichero utiliza el método explicado en la sección 2.3, utiliza este patrón.

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern("dd/MM/yyy;HH:mm:ss;VV");
```

2.4. Más sobre fechas

2.4.1. TemporalAdjuster

Hay más características de Java que pueden resultar útiles cuando se trabaja con fechas. Por ejemplo, ***TemporalAdjuster*** y ***TemporalAdjusters*** son útiles para sumar o restar una cantidad de tiempo a un *Instant* o *LocalDate*.

```
LocalDate now = LocalDate.now();
LocalDate nextMonday = now.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
```

TemporalAdjusters proporciona 14 métodos estáticos para este tipo de operaciones, como *firstDayOfMonth()*, *lastDayOfMonth()*, *firstDayOfYear()*, …

2.4.2. Puentes entre API

Cómo interoperar con *legacy Date API* (antes de Java 8):

- *Instant* ↔ *Date*

```
Date date = Date.from(Instant.now()); // Instant → Date
Instant instant = date.toInstant(); // Date → Instant
```

- *Instant* ↔ *TimeStamp*

```
Timestamp time = Timestamp.from(Instant.now()); // Instant → TimeStamp
Instant instant = time.toInstant(); // TimeStamp → Instant
```

3. StringBuilder and StringJoiner

Antes de Java 7, la concatenación de *Strings* '+' no era muy eficiente. Internamente, se creaba un nuevo objeto *String* para cada concatenación, y si había más de una concatenación, los objetos intermedios también se eliminaban al mismo tiempo, consumiendo muchos recursos. La forma recomendada para concatenar cadenas era utilizar ***StringBuilder***:

```
StringBuilder strBuild = new StringBuilder("Hello");
strBuild.append(" ").append("world!").append(" Yes!");
String str = strBuild.toString();
```

Esto era más eficiente que hacer `String str = "Hello" + " " + "world!" + " Yes!"`. Desde Java 7, el compilador usa internamente *StringBuilder* para procesar concatenaciones de cadenas, por lo que ya no hay problema (si todavía utilizas Java 6, es mejor usar *StringBuilder* manualmente).

Desde Java 8, puedes utilizar ***StringJoiner***, una versión más flexible de *StringBuilder*. Por ejemplo, puedes especificar un separador predeterminado para cada cadena que se añada:

```
StringJoiner joiner = new StringJoiner(", ");
joiner.add("Peter").add("John").add("Mary");
System.out.println(joiner.toString());  // Will print -> Peter, John, Mary
```

También puede especificarse un prefijo y un sufijo:

```
StringJoiner joiner = new StringJoiner(", ", "[", "]");
joiner.add("Peter").add("John").add("Mary");
System.out.println(joiner.toString());  // Will print -> [Peter, John, Mary]
```

Puedes usarlo desde el método estático ***join()*** de la clase *String*. El primer argumento será el separador y el segundo un *array* o una colección iterable.

```
String[] names = {"Peter", "John", "Mary"};
System.out.println(String.join(", ", names));  // Will print -> Peter, John, Mary
```

4. Nuevas funciones de interfaz

4.1. Métodos predeterminados y estáticos

Hasta Java 8, una interfaz solo podía tener métodos abstractos y constantes (*final static*). Ahora, una interfaz también puede tener métodos estáticos (implementados) y métodos predeterminados (también implementados) que serán heredados por las clases que implementan la interfaz.

Por ejemplo, la interfaz ***Iterable*** de Java (***java.lang***), aparte del método ***iterator()***, que es abstracto, ha introducido dos nuevos métodos predeterminados. Uno de ellos es ***forEach()***, que coge la interfaz implementada *Consumer* (pronto se verá qué es un *Consumer*) y la aplica a todos los elementos de una colección, por ejemplo.

```
public interface Iterable<E> {
    // Other methods
    default void forEach(Consumer<E> consumer) {
        Objects.requireNonNull(consumer);
        for(E e: this) {
            consumer.accept(e);
        }
    }
}
```

Un método *default* se heredará como si fuese un método de clase normal. Los métodos estáticos también se pueden definir si fuese necesario.

4.2. Nuevas interfaces funcionales

Recuerda que una interfaz funcional es una interfaz que solo tiene un método abstracto, por lo que puede implementarse usando expresiones lambda. En Java 8, se han definido muchas interfaces funcionales nuevas para operaciones genéricas dentro del paquete ***java.util.function***, tal como:

4.2.1. Consumer

La interfaz ***Consumer*** toma un objeto (o más) como argumento y no devuelve nada (*System::println*).

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
}

// MAIN
List<String> strings = Arrays.asList("one", "two", "three", "four", "five");
List<String> result = new ArrayList<>();
Consumer<String> cPrint = System.out::println;
Consumer<String> cAdd = result::add;

// Will print and then add to the other list (chaining consumers)
strings.forEach(cPrint.andThen(cAdd));
```

4.2.2. Predicate

Esta interfaz recibe un objeto (o más) y devuelve un booleano. Se usa con colecciones, por ejemplo, cuando se quiere filtrar aquellas que coinciden con una condición determinada.

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
}

// MAIN
Predicate<String> p1 = s -> s.length() < 20;
Predicate<String> p2 = s -> s.length() > 10;
Predicate<String> p3 = p1.and(p2); // Applies both p1 and p2
// You can also create new predicates with .or(Predicate) and .negate()
```

4.2.3. Function

La interfaz ***Function*** toma un objeto (o más) y devuelve otro objeto.

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Hay varios tipos, como:

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}

@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

Fíjate en el siguiente ejemplo:

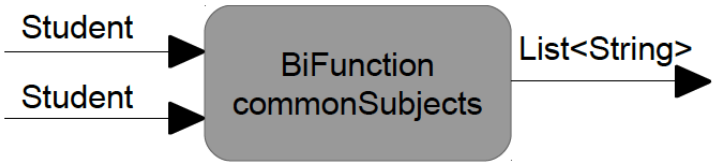
```
/* This BiFunction takes a string s and an integer i and returns a
substring of s containing the first i characters */

BiFunction<String, Integer, String> biFunc = (s,i) -> s.substring(i);
// This Function takes a string and converts it to uppercase
Function<String, String> func = s -> s.toUpperCase();
// This BiFunction combines both previous functions in a third one
BiFunction<String, Integer, String> biFunc2 = biFunc.andThen(func);
```

4.2.4. Ejemplo

Observa cómo funcionan estas interfaces funcionales con un ejemplo. En particular, se trabajará con *Functions*. Supón que tienes una clase llamada *Student*, con los siguientes atributos: *name* (*String*), *age* (*integer*) y una lista de asignaturas, *subjects* (*Strings*).

Se podría definir una *BiFunction* que reciba dos objetos *Student* como parámetros y devuelve una lista de *Strings* que contenga las materias (*subjects*) que ambos estudiantes tienen en común. Podría ser así:

```
BiFunction<Student, Student, List<String>> commonSubjects = (s1,s2) -> {
    List<String> list = new ArrayList<>(s1.getSubjects()); // Copy
    list.retainAll(s2.getSubjects()); // Only keep elements in common
    return list;
};
```



También se podría definir una *Function* que tome una lista de asignaturas y devuelva un *String* separando las asignaturas por comas. Para hacer esto, se pueden usar las funciones vistas antes con *StringJoiner*:

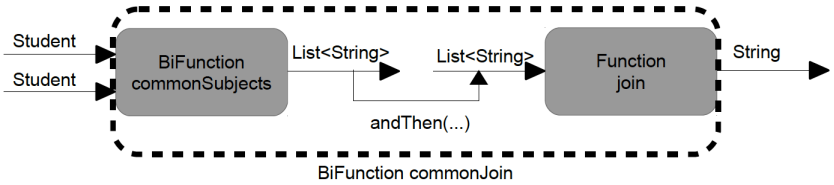
```
Function<List<String>, String> join = set -> String.join(", ", set);
```

Por último, se podría combinar ambas funciones en una tercera:

```
BiFunction<Student, Student, String> commonJoin = commonSubjects.andThen(join);
```

En esta tercera función combinada, se debe prestar atención al orden en que se llaman a las funciones. Primero se llama a la *BiFunction commonSubjects*, por lo que se debe definir esta tercera función como un *BiFunción*, con los mismos parámetros de entrada que *commonSubjects*. Después, se llama a la segunda función *join()*, por lo que el tipo de retorno de la tercera función debe ser el mismo que el devuelto por la función *join(String)*.

Finalmente, se llama a la función *commonSubjects* y vincular su salida con la entrada de la función *join()* con el método *andThen()*.



Puede probarse esta función de la siguiente manera:

```
String test = commonJoin.apply(
    new Student("Trevor Swanson", 19,
        Arrays.asList("English","Spanish","Drawing")),
    new Student("Sue Smith", 20,
        Arrays.asList("English","Literature","Drawing", "Philosophy"))
);
System.out.println(test);
```

5. Programación funcional y I/O

A continuación, se verán algunas características nuevas añadidas a Java con respecto a la programación funcional y las tareas I/O, como la lectura o escritura de archivos.

- El método ***lines()*** se añadió a ***BufferedReader***. Devuelve un ***Stream<String>***:

  ```
  try(BufferedReader reader = new BufferedReader(
      new FileReader("/home/javier/file.txt"))) {
          reader.lines()
              .filter(line -> line.contains("Login: "))
              .forEach(System.out::println);
  } catch …
  ```

  El método ***Files.lines()*** crea el *Stream* sin tener que crear un ***BufferedReader***, y es ***AutoCloseable***, para que puedas crearlo dentro de un *try*:

  ```
  try(Stream<String> stream = Files.lines(
      Paths.get("/home/javier", "file.txt"))) {
          stream
              .filter(line -> line.contains("Login: "))
              .forEach(System.out::println);
  } catch …
  ```

- ***Files.list()*** devuelve un ***Stream<Path>*** que contiene una lista de todos los archivos y directorios presentes en el directorio actual (pasado como parámetro):

  ```
  try(Stream<Path> stream = Files.list(Paths.get("/home/javier"))) {
      stream // Prints subdirectories
          .filter(path -> path.toFile().isDirectory())
          .forEach(System.out::println);
  } catch …
  ```

- ***Files.walk()*** es similar, pero también explora subdirectorios. El segundo parámetro es la profundidad máxima a la que se desea ir en el árbol de directorios:

```
try(Stream<Path> stream = Files.walk(Paths.get("/home/javier"), 2)) {
    stream // Prints subdirectories
        .filter(path -> path.toFile().isDirectory())
        .forEach(System.out::println);
} catch …
```

6. Más funciones para colecciones

En cuanto a las colecciones, también se añadieron características nuevas desde Java 8. A continuación tienes algunas de ellas:

- Existe un nuevo método, ***forEach()***, en las colecciones ***Iterable***:

```
List<Integer> list = Arrays.asList(4,6,7,9);
list.forEach(elem -> {
    if(elem % 2 == 0) {
        System.out.println(elem);
    }
});
```

- ***List → replaceAll(UnaryOperator)***: Aplica una *Function* (*UnaryOperator*) a todos sus elementos, modificándolos:

```
List<Integer> list = Arrays.asList(4,6,7,9,12,15);
list.replaceAll(Math::incrementExact); // Increments all by 1
System.out.println(list.stream().map(i -> i.toString())
    .collect(Collectors.joining(", ")));
```

- Con respecto al interfaz ***Comparator***, también hay nuevos métodos. Por ejemplo, ¿qué se puede hacer con los nulos?:

```
list.sort(Comparator.nullsFirst(Comparator.naturalOrder()));
list.sort(Comparator.nullsLast(Comparator.naturalOrder()));
```

  Como puedes ver, también se pueden encadenar criterios de comparación. Observa un comparador para ordenar a las personas por su apellido, y si es igual, use el nombre (Java 7 vs Java 8):

```
List<Person> list = new ArrayList<>();

// JAVA 7
list.sort(new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        int last = p1.getLastName().compareTo(p2.getLastName());
        if(last == 0) { // Equal last name
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
        return last;
    }
});

// JAVA 8
list.sort(Comparator.comparing(Person::getLastName)
.thenComparing(Person::getFirstName));
```

- ***Maps*** → Nuevos métodos:
  - ***forEach(BiConsumer)***. Igual que List::forEach pero con *BiConsumer(key, value)*:

```
Map<String, Person> map = new HashMap<>();
…
map.forEach((key,person) ->
    System.out.println("Key: " + key +
        ". Person: " + person.getFullName()));
```

- ***getOrDefault(key, defaultValue)***. Si la clave no tiene un valor, devuelve el valor predeterminado (recibido como parámetro).

```
Map<String, Person> map = new HashMap<>();
Person defaultPerson = …;
Person p = map.getOrDefault(key, defaultPerson);
```

- ***putIfAbsent(key, value)***. Con *put()*, si una clave ya tenía un valor, este se modificó sin preguntar. Con este nuevo método, el nuevo valor solo se pone si la clave no tenía un valor anterior.

```
Map<String, Person> map = new HashMap<>();
map.putIfAbsent(key, person);
```

- ***replaceAll(BiFunction)***. Aplica la función pasada como parámetro a todos los elementos transformándolos.

```
Map<String, Person> map = new HashMap<>();
map.replaceAll((key, p) ->
    new Person(p.getFirstName().toUpperCase(),
        p.getLastName().toUpperCase()));
```

- …

7. Un ejemplo de lambdas y streams

Ahora se verá un ejemplo completo usando expresiones lambda y *streams*, para que puedas reforzar lo que has aprendido al leer la parte 3 de esta unidad. En este caso, se tratará con una clase llamada *Book* con un conjunto de atributos, *getters*, *setters*, etc.

```
public class Book {
    String title;
    String author;
    int pages;
    …
}
```

7.1. Ordenar libros

Se puede, por ejemplo, definir un *TreeSet* que ordene un conjunto de libros por su título. Para hacer esto, se necesita definir una instancia de la interfaz *Comparator*. Esta interfaz obtiene dos objetos (libros en este caso) y devuelve cuál de ellos es mayor o menor según el atributo o combinación de atributos (en este caso, se ordenarán los libros

alfabéticamente, por título). Si devuelve un número positivo, entonces el primer objeto es mayor. Si devuelve un número negativo, entonces el segundo objeto es mayor. Si devuelve 0, ambos objetos son iguales. Se puede hacer esto con una clase anónima:

```
Comparator<Book> comp = new Comparator<Book>() {
    @Override
    public int compare(Book b1, Book b2) {
        return b1.getTitle().compareTo(b2.getTitle());
    }
};
```

O con una expresión lambda:

```
Comparator<Book> comp = (b1, b2) -> b1.getTitle().compareTo(b2.getTitle());
```

Después, se crea un *TreeSet* con este comparador, y a medida que se le agreguen libros, se ordenan automáticamente:

```
TreeSet<Book> tree = new TreeSet<>(comp);
tree.add(new Book("Alice in Wonderland", "Lewis Carroll", 196));
tree.add(new Book("Hamlet", "William Shakespeare", 523));
tree.add(new Book("Ender's game", "Orson Scott Card", 213));
```

Finalmente, se puede iterar sobre el *TreeSet* y mostrar los libros en orden:

```
for (Book b: tree) {
    System.out.println(b);
}
```

7.2. Aplicar algunos filtros básicos

Se definirá una lista de libros:

```
List<Book> books = new ArrayList<>();
books.add(new Book("Alice in Wonderland", "Lewis Carroll", 196));
books.add(new Book("Hamlet", "William Shakespeare", 623));
books.add(new Book("Ender's game", "Orson Scott Card", 213));
books.add(new Book("The Holy Bible", "Anonymous", 596));
…
```

Se pueden aplicar algunos filtros a la lista y obtener algunas listas parciales como resultado. Por ejemplo, si se quiere obtener una lista de todos los libros con más de 500 páginas e imprimirlos, se podría aplicar este filtro (operación intermedia):

```
Stream<Book> moreThan500 = stream.filter(b -> b.getPages() > 500);
```

y luego se utilizará esta operación final para imprimirlos:

```
moreThan500.forEach(System.out::println);
```

7.3. Combinar filtros

Se puede combinar algunos filtros uniéndolos con el operador ***“.”***. Por ejemplo, se puede calcular la media del número de páginas de aquellos libros con más de 500 páginas:

```
OptionalDouble avg = stream.filter(b -> b.getPages() > 500)
                                                .mapToInt(b -> b.getPages()).average();
```

Se utiliza el objeto *OptionalDouble* porque la secuencia puede estar vacía (si no hay libros con más de 500 páginas). Después, se verifica si este objeto tiene un valor o no, y se imprime el mensaje con el resultado correspondiente:

```
System.out.println(avg.isPresent()?avg.getAsDouble():"No matches");
```

7.4. Mapeo de streams

Cuando se mapea un *stream*, se centrará la atención en un solo atributo de los objetos del *stream*. Por ejemplo, con este filtro y mapeo se imprimen solo los títulos de los libros con más de 500 páginas:

```
books.stream().filter(b -> b.getPages() > 500)
        .map(b -> b.getTitle())
        .forEach(System.out::println);
```

7.5. Recolección de streams

Las operaciones de recolección pueden devolver una colección, un subconjunto de la original. Por ejemplo, se puede devolver una lista que contenga solo los libros con más de 500 páginas:

```
List<Book> listMoreThan500 =
        books.stream().filter(b -> b.getPages() > 500)
                .collect(Collectors.toList());
```

También se puede devolver, por ejemplo, una lista de los títulos de estos libros, separados por comas:

```
String resultMoreThan500 =
        books.stream().filter(b -> b.getPages() > 500)
                .map(b -> b.getTitle())
                .collect(Collectors.joining(", "));
```

Ejercicio 2

Se partirá de un proyecto de muestra para realizar los siguientes ejercicios. Hay una clase ***Product*** con algunos atributos, constructores, *getters* y *setters*, y un *main* en el que se creará una lista de productos. A partir de este punto, se definirán algunos *streams*, expresiones lambda y *functions* para obtener resultados de esta colección.

- Define un Comparator usando una expresión lambda para ordenar la lista de productos por categoría y nombre (en este orden). Ordena la lista usando el método *Collections.sort* y muéstrela para verificar que el orden sea correcto.
- Crea un *stream* que muestre los productos cuya categoría es *"Tablets"*. El *stream* devolverá una lista y se iterará para mostrar la información de los productos. ¿Cómo harías esto sin generar ninguna lista secundaria?

- Crea un *stream* que calcule la media de precios para una categoría determinada. Prueba este *stream* con la aplicación principal y muestra los resultados de la categoría *“Videogames”*.
- Crea una *BiFunction* que reciba dos parámetros: una lista de objetos *Product* y un precio, y devuelva un *String* que contenga todos los nombres de productos (separados por comas) cuyo precio sea más alto que el proporcionado como parámetro. Utiliza esta función en *main* con un precio de 100 y muestra los resultados.
- Crea un *stream* que cuente cuántos productos hay agrupados por categoría. Explora el mapa resultante y muéstralo por consola.