1. Refuerzo de Java

Parte II. Colecciones, I/O y control de excepciones

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

4. Colecciones

Una colección es un grupo de objetos que se quieren almacenar y operar con ellos de una manera particular. Según la forma en la que se almacenen (ordenados, únicos,….) y como se opere con ellos (comparar, iterar,…), habrá un tipo de colección más adecuado que otro.

4.1. Listas

Una lista (*List*) es una colección ordenada (por un índice → 0, 1, 2,…., N) y una colección secuencial (iterable). Puede tener duplicados (el mismo objeto en diferentes posiciones). En Java, una lista es una **interfaz**, por lo que se debe instanciar siempre un subtipo del mismo (*ArrayList*, *LinkedList*, *Stack*, *Vector*, …) según las necesidades.

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

4.1.1. ArrayList

Esta clase es una implementación de una lista que usa una matriz interna para almacenar elementos. Su principal ventaja es la complejidad de acceder a un índice para obtener un elemento, que es O(1), pero no es la mejor implementación cuando se quieren hacer muchas operaciones de adición y eliminación, que tienen una complejidad de O(n).

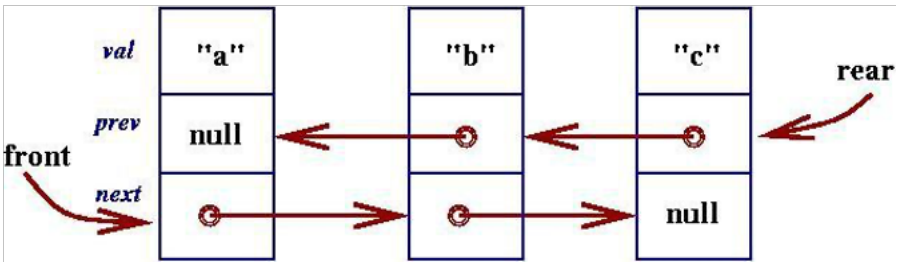
Complejidades:

```
get(int index) es O(1), principal beneficio de ArrayList<E>.
add(E element) cuando la matriz interna es lo suficientemente grande es O(1), pero O(n) en el peor de los casos, ya que la matriz debe redimensionarse y copiarse.
add(int index, E element) es O(n - índice) si no es necesario cambiar el tamaño, pero O(n) en el peor de los casos (como la anterior).
remove(int index) es O(n - índice) (por ejemplo, eliminar el último sería O(1)).
Iterator.remove() es O(n - índice)
ListIterator.add(E element) es O(n - índice)
```

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

4.1.2. LinkedList

Este tipo de lista se implementa como una lista doblemente enlazada. Los elementos están vinculados por una referencia al elemento siguiente y anterior de la lista (el primer elemento y el último elemento estarán enlazados a nulo).



Internamente no es un *array*, por lo que no es necesario cambiar el tamaño de nada, los elementos se insertan y eliminan solo modificando las referencias en el objeto anterior y siguiente. Ésta es la principal ventaja de este tipo de lista. La principal desventaja es acceder a una posición aleatoria porque debes viajar a través de todos los elementos para llegar al índice especificado (no hay un índice interno para acceder a una posición directamente).

Complejidades:

`get(int index)` es O(n).
`add(E element)` es `O(1)`.
`add(int index, E element)` es `O(n)`.
`remove(int index)` es O(n).
`Iterator.remove()` es `O(1)`, principal beneficio de `LinkedList<E>`.
`ListIterator.add(E element)` es `O(1)`, principal beneficio de `LinkedList<E>`.

Esta implementación es más adecuada para ser explorada con un *Iterator*, o *ListIterator*, que con el clásico bucle *for* (funcionará, pero será menos eficiente). Si deseas acceder a posiciones aleatorias con frecuencia, es mejor utilizar *ArrayList*.

```
List<String> strings = new LinkedList<>();

strings.add("Hello");
strings.add("world!");
strings.add("How");
strings.add("are");
strings.add("you?");

ListIterator<String> strIt = strings.listIterator();

while (strIt.hasNext()) {
    System.out.print(strIt.next() + " ");
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Ejercicio 3

Creta un proyecto llamado ***ListBenchmark***. La idea es probar en qué casos es mejor utilizar un *ArrayList* o una *LinkedList*. Para medir el tiempo que tarda una operación, puedes utilizar el siguiente fragmento de código:

```
Instant start = Instant.now();
// Some operation with ArrayList or LinkedList
Instant end = Instant.now();

Duration dur = Duration.between(start, end);
System.out.printf("ArrayList: The operation ... takes: %dms\n", dur.toMillis());
```

Tienes que comparar estas situaciones (*ArrayList<Double>* vs *LinkedList<Double>*). Crea cada uno de ellos vacío y reutilizalos en cada comparación:

1. Añade 100.000 (*doubles*) elementos aleatorios siempre desde la posición 0. Compara los tiempos.
2. Elimina los primeros 50.000 elementos (elimina siempre el primero).

3. Añade 50.000 elementos aleatorios en posiciones aleatorias.
4. Elimina 50.000 elementos de posiciones aleatorias.
5. Elimina los elementos que están en posiciones pares (divisibles por 2) usando un *Iterator*.

Observarás que cuando usa muchos accesos aleatorios (índice), el *ArrayList* es mucho más rápido (*LinkedList* necesita contar desde el principio). Al agregar o eliminar elementos al principio, o al usar un *Iterator*, la situación es la opuesta (*ArrayList* tiene que reordenar los índices internos cada vez, mientras que en el *LinkedList* no es necesario).

4.2. Mapas

Los ***Maps*** son un tipo de colección que mapea una clave (representada con un objeto) con un valor (que también es un objeto). Las claves deben ser únicas, y la principal diferencia con otros tipos de colecciones, como las listas, es que una clave no tiene que ser un número entero.

https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Existen muchas implementaciones de esta interfaz, cada una adecuada para diferentes necesidades.

4.2.1. *HashMap* y *Hashtable*

HashMap y ***Hashtable*** son implementaciones de mapas que utilizan una función *hash* para distribuir claves. La principal diferencia entre ellas es que una *Hashtable* está sincronizada, por lo que es segura para sub-procesos. Otra diferencia es que con *HashMap* puedes usar nulos como clave.

La principal ventaja de este tipo de estructuras es que, cuando se quiere buscar un valor, es rápido de encontrar (comparado con una lista por ejemplo), ya que solo tiene que calcular una función *hash* para averiguar dónde se encuentra el objeto deseado. Cuando se necesita insertar muchas entradas, o buscar, en lugar de iterar, la implementación de un mapa suele ser más adecuada que una lista.

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html

4.3. Conjuntos (*Sets*)

Un conjunto (***set***) define una colección de objetos cuya característica principal es no contener duplicados, además no mantiene dichos objetos ordenados mediante un índice. Es solo un repositorio de objetos que le permite saber si un objeto está en el conjunto o no.

Puedes recorrer los objetos contenidos con un *Iterator*, o generar una matriz con el método *toArray()*. Como siempre, hay muchas implementaciones de esta interfaz, cada una con sus pros y sus contras.

https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

4.3.1. HashSet

La implementación de ***HashSet*** utiliza un *HashMap* interno para almacenar valores (como claves que no contienen nada). Tiene sus ventajas: rápida inserción, eliminación y búsqueda. Y sus desventajas: función *hash*, sin orden establecido, iteración costosa.

https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

4.4. Clase Collections

En Java, la clase ***Collections*** tiene métodos estáticos diseñados para operar o generar nuevas colecciones. Esos métodos incluyen la generación de colecciones sincronizadas (para una operación segura con múltiples sub-procesos), ordenar una lista (con algún tipo de comparador), invertir una lista, rotar una lista, obtener el elemento mínimo, máximo, …

https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

5. Excepciones

Las excepciones son una forma común y, generalmente recomendada, de tratar los errores en las aplicaciones, especialmente los graves. Cuando se utiliza una API de Java (*arrays*, *strings*, archivos, fechas,….), cualquier error, esperado o no, producirá una excepción. ***Exception*** (y sus subclases) es una clase especial de Java (y de otros lenguajes). Lo que realmente se obtiene cuando se produce un error es un objeto *Exception* (o un subtipo de *Exception*).

Si no se captura (*catch*) el objeto de *Exception*, o se lanza para que pueda ser capturado más tarde, se generará un *fatal error* y la aplicación se bloqueará inmediatamente. Esta política se puede definir como; *“Si hay un error que no estamos esperando (try… catch), puede causar una corrupción en nuestros valiosos datos, por lo que es mejor detener la aplicación que dejarla correr y solo informar (el ordenador no puede saber si un error es grave o no, por eso detiene la operación en ejecución para que el programador pueda decidir qué hacer con ese error en el futuro)”*.

Una excepción se captura utilizando la declaración ***try… catch***. Puedes capturar tantos tipos de excepciones como desees dentro de un bloque de prueba (captura múltiple). Se recomienda especificar tantos bloques de captura como sean necesarios para cada tipo de excepción en lugar de capturar una *“Exception”* genérica, especialmente si desea proporcionar una respuesta diferente a cada tipo de posible excepción. También puedes especificar un bloque ***finally*** al final de cada captura para aquellas operaciones que se deban realizar tanto si hay un error como si no (como cerrar una secuencia).

```
String line;
BufferedReader bfRead = null;

try {
    bfRead = new BufferedReader(new FileReader("/home/pepe/file.txt"));
    line = bfRead.readLine();
    while(line != null) {
        line = bfRead.readLine();
    }
} catch (FileNotFoundException e) {
    System.err.println("File /home/pepe/file.txt doesn't exist!");
```

```
} catch (IOException e) {
    System.err.println("Error reading /home/pepe/file.txt.");
} finally {
    if(bfRead != null)
        try {
            bfRead.close();
        } catch (IOException e) {
            System.err.println("Couldn't close C:\file.txt stream.");
        }
}
```

Por fortuna, desde Java 7, se pueden crear variables (separadas por punto y coma ';') como argumentos después del ***try***. Este tipo de variables deben implementar la interfaz ***AutoCloseable*** (como *StreamReader*). Existirán localmente solo en el bloque *try*, y Java los cerrará automáticamente después de que finalice ese bloque. Esta práctica se conoce como *try-with-resources*, y ahorra bastante código adicional feo:

```
String line;

try (BufferedReader bfRead = new BufferedReader(new FileReader("/home/pepe/file.txt"))) {
    line = bfRead.readLine();

    while(line != null) {
        line = bfRead.readLine();
    }
} catch (FileNotFoundException e) {
    System.err.println("File /home/pepe/file.txt doesn't exist!");
} catch (IOException e) {
    System.err.println("Error reading /home/pepe/file.txt.");
}
```

Si consideras que una excepción no debe administrarse dentro del método en el que se está produciendo, sino que prefieres delegar la administración al método padre en la pila de llamadas, puedes utilizar la instrucción ***throws*** en la definición del método. Después de la cláusula *throws*, puedes escribir todos los tipos de excepciones que necesites (separados por comas) que se envíen. Cuando se genera una excepción de este tipo, el método se interrumpirá devolviendo el objeto *Exception* generado que debe ser capturado en el método que realizó la llamada. Puedes hacerlo tantas veces como desees hasta el método ***Main***. Sin embargo, no es muy recomendable lanzar excepciones que no sean creadas por nosotros mismos.

Siempre podrás crear una nueva excepción (generalmente genérica) y lanzarla con la instrucción especial ***throw***:

```
public void welcome() throws Exception {
    throw new Exception("Error, nobody can pass!");
}
```

5.1. Excepciones personalizadas

Se pueden crear nuestras propias excepciones creando clases que hereden de la clase *Exception*. Después se puede lanzar una excepción personalizada cuando sea necesario y administrarla en el método que la lanzó o enviarla al método al que volverá.

```java
public class CustomException extends Exception {
    public CustomException(String msg) {
        super(msg);
    }
}

public class Store {
    public void welcome() throws CustomException {
        throw new CustomException("Error, nobody can pass!");
    }
}

public static void main(String[] args) {
    Store store = new Store();
    try {
        // This method can throw a CustomException
        store.welcome();
    } catch (CustomException e) {
        System.err.println(e.getMessage());
    }
}
```

Ejercicio 4

Crea un proyecto llamado ***CustomException***. En este proyecto vas a implementar:

- Una clase llamada ***NegativeSubtractException***. Esta clase heredará de *Exception* y se creará cuando el resultado de una resta sea negativo. El constructor recibirá 2 parámetros (los dos números que causaron una resta negativa debido al orden). El mensaje generado será: *"NegativeSubstractException: el resultado 'N1 - N2' es negativo"*.
- En la clase principal crea un método estático que lance este tipo de excepción personalizada. Este método se llamará ***int positiveSubtract(int n1, int n2)***, y generará, y lanzará, este tipo de excepción si el resultado es negativo. En el método ***main*** llama a este método con parámetros que dará un resultado negativo y detectará la excepción correspondiente, mostrando el mensaje por consola.

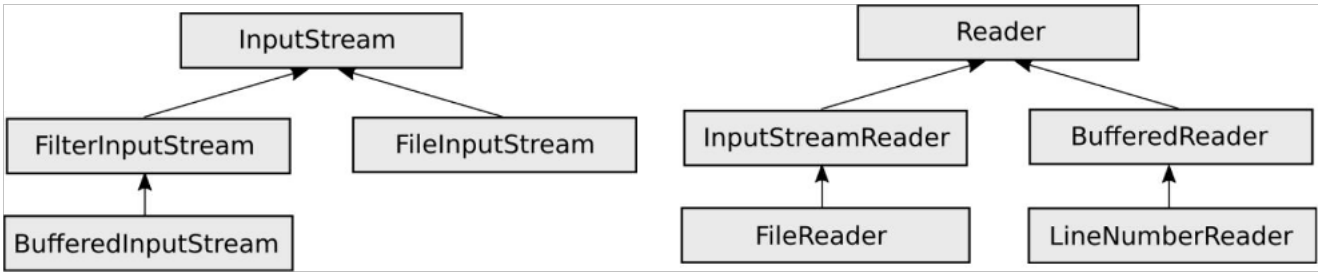
6. Java I/O

Cuando se trabaja con archivos o accedemos a servicios remotos como servicios web, hay dos operaciones que se pueden hacer, escribir (*out*) y leer (*in*) en el flujo de datos que comunica nuestra aplicación con el archivo o servicio.

Las clases básicas para gestionar este tipo de operaciones de entrada/salida son:

- ***InputStream***: Representa el flujo de datos (*bytes*) que llega a la aplicación para su lectura (*input*).
- ***OutputStream***: Representa el flujo de datos (*bytes*) que sale de la aplicación (*write* – *output*).
- ***Reader***: Representa el flujo de caracteres que llega a la aplicación (*input*).
- ***Writer***: Representa el flujo de caracteres que sale de la aplicación (*output*).

Estas clases proporcionan una funcionalidad que suele ser demasiado básica para las operaciones más habituales, como la lectura o escritura de archivos. Hay subclases de estas que implementan una funcionalidad más avanzada. A esto se le llama *Decorator Pattern* en el diseño de *software* (cada capa añade funcionalidades más específicas).



Se puede utilizar una combinación *Readers*, por ejemplo:

```
try(BufferedReader buffer = new BufferedReader(new FileReader("/home/pepe/file.txt"))) {
    String line;
    while((line = buffer.readLine()) != null) {
        System.out.println(line);
    }
} catch (FileNotFoundException e) {
    System.err.println("File /home/pepe/file.txt doesn't exist!");
} catch (IOException e) {
    System.err.println("Error reading /home/pepe/file.txt.");
}
```

O *Writers*:

```
try (PrintWriter print = new PrintWriter(new FileWriter("/home/pepe/file.txt"))) {
    print.println("Hello world!");
} catch (IOException e) {
    System.err.println("Error writing: " + e.getMessage());
}
```

También puedes escribir tu propia implementación de una de estas clases utilizando herencia y añadir métodos nuevos, o modificar, alguna funcionalidad. Por ejemplo, esta clase hereda de la clase *BufferedReader* para devolver cada línea en mayúsculas:

```
public class UpperCaseReader extends BufferedReader {
    public UpperCaseReader(Reader reader) {
        super(reader);
     }

    @Override
    public String readLine() throws IOException {
        String line = super.readLine();
        return line != null?line.toUpperCase():null;
    }
}
```

Esta clase personalizada se utilizaría en el programa principal de esta manera:

```
try(UpperCaseReader buffer = new UpperCaseReader(new FileReader("/home/pepe/file.txt"))) {
    String line;
    while((line = buffer.readLine()) != null) {
        System.out.println(line);
    }
} catch (FileNotFoundException e) {
    …
```

```
} catch (IOException e1) {
    …
}
```

Ejercicio 5

Crea un proyecto llamado ***RegexReader*** que contenga lo siguiente:

- Una clase llamada ***RegexReader*** que herede de ***BufferedReader*** (echa un vistazo al ejemplo anterior para ayudarte a implementar esta clase). Recibirá un *Reader* en el constructor y un *String* que contendrá una expresión regular para buscar coincidencias al leer las líneas. Tendrás que sobrecargar el método *readLine()*. Utilizará el método *readLine()* del padre y devuelve la primera línea que coincida con la expresión regular recibida en el constructor (utiliza el método ***matches(String)*** de la clase *String*), ignorando el resto de líneas (hasta que se nulo).
- En la clase principal, crea un objeto ***RegexReader*** que recibirá solo líneas que contienen una fecha (en formato dd/mm/yy o dd/mm/yyyy), mezcladas con el texto. Una expresión regular en Java siempre intentará hacer coincidir todas las cadenas para que sea válida, por lo que tendrás que añadir .* al inicio y final de la expresión, además, deberás tener en cuenta que **\d** no es válido, tendrás que utilizar **\\d**. Crea un fichero que contenga algunas fechas en varias líneas y pruébalo.

6.1. Operaciones con el sistema de archivos

Desde Java 7, se han añadido nuevas clases para facilitar las operaciones con el sistema de archivos del SO. Hablamos principalmente de las clases ***Paths*** y ***Files***, que proporcionan métodos estáticos para este propósito. La mayoría de las veces, estas clases devuelven un ***Path***. Se utilizará esta interfaz para trabajar con un fichero en el sistema de archivos:

```
Path file = Paths.get("/home/pepe/file.txt");
System.out.println("Path: " + file.getParent());
System.out.println("Separator: " + file.getFileSystem().getSeparator());
System.out.println("File name: " + file.getFileName());
```

Se pueden hacer muchas operaciones básicas con archivos y directorios haciendo uso de los métodos ***Files***:

```
Path file = Paths.get("/home/pepe/file.txt");
Path parent = file.getParent();
Path destiny = Paths.get("/home/pepe/docs");

// Create
if(Files.isDirectory(parent) && Files.isWritable(parent) && !Files.exists(file)) {
    try {
        Files.createFile(file);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Move
if(Files.isDirectory(destiny) && Files.isWritable(destiny)) {
    try {
        Files.move(file, destiny);
    } catch (IOException e) {
```

```
        e.printStackTrace();
    }
}

// Delete
try {
    Files.deleteIfExists(file);
} catch (IOException e) {
    e.printStackTrace();
}
```

Más información:

https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html

https://docs.oracle.com/javase/8/docs/api/java/nio/file/Paths.html

https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html