PRG (E.T.S. de Ingeniería Informática) - Curso 2018-2019

Práctica 4. Tratamiento de excepciones y ficheros en un registro ordenado de accidentes. Primera parte

(2 sesiones)

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Contexto y objetivos

En el marco académico, esta práctica corresponde al "*Tema 3. Elementos de la POO: herencia y tratamiento de excepciones*" y al "*Tema 4. E/S: ficheros y flujos*". El objetivo principal que se pretende alcanzar con ella es reforzar y poner en práctica los conceptos introducidos en las clases de teoría sobre el tratamiento de excepciones y la gestión de la E/S mediante ficheros y flujos. En concreto:

- Lanzar, propagar y capturar excepciones local y remotamente.
- Leer/escribir desde/en un fichero de texto.
- Tratar las excepciones relacionadas con la E/S.

Para ello, durante las tres sesiones de prácticas, se va a desarrollar una pequeña aplicación en la que se procesarán los datos que se lean de ficheros de texto, guardando el resultado en otro fichero.

2. Planteamiento del problema

Se dispone de un registro del número de accidentes acaecidos a lo largo de un año. El registro puede provenir de una o más áreas (ciudades, provincias, ...), y los datos pueden encontrarse distribuidos en uno o más ficheros de texto, donde cada línea tiene el formato siguiente:

```
dia  mes  cantidad
```

en donde `cantidad` es un dato registrado para la fecha dada por `dia` y `mes`.

En un mismo fichero de datos pueden darse fechas repetidas, y las líneas no tienen por qué estar en orden cronológico, como podría darse si en un mismo fichero se hubiesen concatenado los datos procedentes de diversas áreas.

Se desea una aplicación que extraiga los datos de un año a partir de unos ficheros de texto, y genere un fichero de resultados en el que aparezcan registrados, y por orden cronológico, los datos acumulados de cada fecha para la que constan registros, a la manera que se muestra en la figura 1. Debe contemplarse también, la posibilidad de que los ficheros contengan anotaciones erróneas, bien porque los datos no correspondan a una fecha correcta, porque la cantidad no sea un entero positivo o porque la línea contenga más o menos de 3 valores.

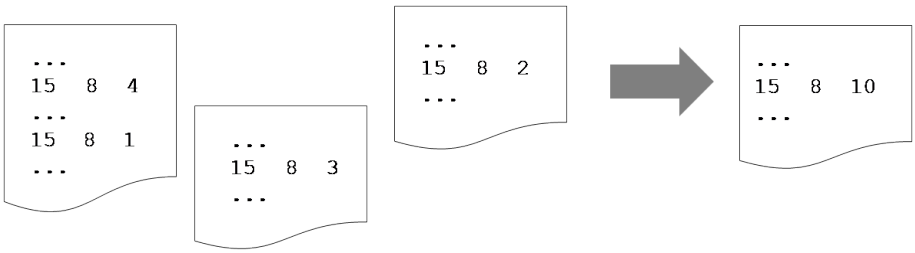

Figura 1: Agregación de datos procedentes de uno o varios ficheros.

Para resolver el problema es necesario usar una matriz, indexada por meses y por los días de cada mes, de forma que la cantidad que aparezca en cada línea de datos que se procese, se acumule directamente en la componente indexada por el mes y el día de la fecha. La estructura de los objetos de esta matriz se muestra en la figura 2. Notar que las componentes de la matriz con algún índice 0, en filas o en columnas, no se usan. De esta forma los datos de una fecha, mes y día, sirven directamente como índices de acceso a la componente correspondiente. Un recorrido de la matriz por filas y columnas permite obtener un listado ordenado por fechas de los datos que se hubieran acumulado.

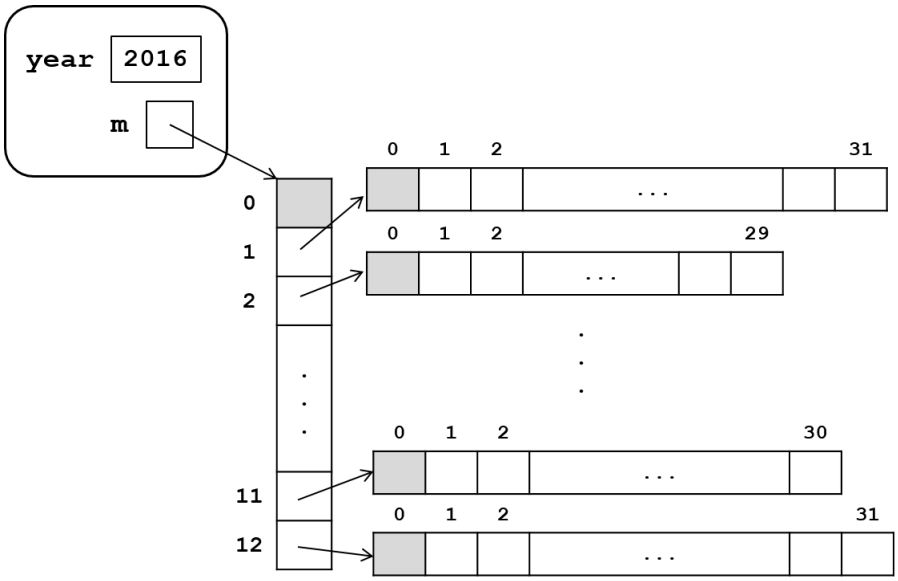

Figura 2: Estructura de un `SortedRegister`.

3. Las clases de la aplicación

Para la resolución del problema planteado se proporcionan las siguientes clases:

- La clase `SortedRegister` contiene un array bidimensional `m` en el que las filas son meses y las columnas los días de cada mes, de manera que `m[f][c]` corresponde a la cantidad de accidentes acumulados el día `c` del mes `f` (ver la figura 2). La clase contiene los métodos necesarios para volcar la información leída desde un `Scanner` sobre la matriz y, a su vez, volcar la información de la matriz en un `PrintWriter`.
- La clase `TestSortedRegister` con la que se van a hacer pruebas del comportamiento de `SortedRegister`.
- La clase de utilidades `CorrectReading` que permite realizar la lectura validada de datos de tipos primitivos desde la entrada estándar, y de la que se va a hacer uso en la clase anterior.

3.1. Actividad 1: preparar el paquete `pract4`

- Abrir con *BlueJ* el proyecto `prg` y crear un paquete `pract4` específico para esta práctica y, dentro de este, un subpaquete llamado `utilPRG`.
- Desde el explorador de archivos y dentro de la carpeta `DiscoW/prg/pract4` crear una nueva carpeta con el nombre `data` donde se dejarán y crearán los ficheros de datos y resultados.
- Descargar desde `Recursos/Laboratorio/Práctica 4` de *PoliformaT* de `PRG`, los ficheros `CorrectReading.java` y `TestCorrectReading.class` y situarlos en el subpaquete `utilPRG`.
- Descargar también los ficheros `SortedRegister.java` y `TestSortedRegister.java` y agregarlos al paquete `pract4`.
- Descargar del mismo sitio los ficheros `data.txt` y `badData.txt`, y copiarlos en la carpeta `DiscoW/prg/pract4/data`.

4. Detección de errores en la lectura desde teclado

El método `nextInt()` de la clase `Scanner`, se ha utilizado frecuentemente. Si se consulta la documentación de la clase (`https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`), se puede observar que si el valor introducido por el usuario no es un entero, puede lanzar la excepción `InputMismatchException` cuya documentación también se puede consultar. Esta es una excepción *unchecked* (derivada de `RuntimeException`), de tal manera que su situación en la jerarquía de clases coincide con la que se muestra en la figura 3, por lo que Java no obliga a prever su tratamiento, aunque puede ser interesante capturarla y tratarla.

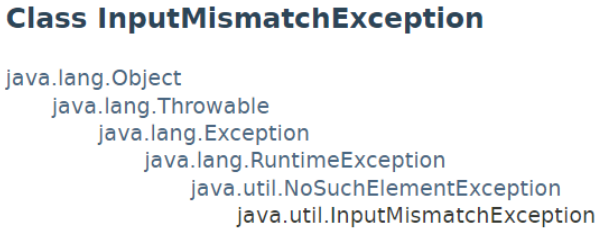

Figura 3: Jerarquía de clases de la excepción `InputMismatchException`.

El método `nextInt(Scanner, String)` de la clase `CorrectReading` permite realizar la lectura de un valor de tipo `int` desde teclado usando el método `nextInt()` de la clase `Scanner`, capturando la excepción en caso de que se produzca y mostrando un mensaje de error para indicar al usuario qué acción correctiva es necesaria.

Se puede probar este método en la *zona de código* (*Code Pad*) de *BlueJ*. Para ello, ejecutar las instrucciones siguientes:

```
import java.util.Scanner;
Scanner t = new Scanner(System.in);
int valor = CorrectReading.nextInt(t, "Valor: ");
```

Desde la *ventana del terminal* de *BlueJ*, introducir un valor no entero, por ejemplo *"hola"*. Se puede observar el mensaje mostrado indicando que el valor no es válido y que la ejecución del método no acaba hasta que se introduce un valor entero. Esto es así, porque este método utiliza un bloque `try-catch-finally` para tratar la excepción.

Observar la cláusula `finally` del método `nextInt(Scanner, String)`. Incluso cuando se produce un error en un método, puede haber instrucciones que se requieren antes de que el método o programa termine. La cláusula `finally` se ejecuta si todas las instrucciones del bloque `try` se ejecutan (y ningún bloque `catch`) o si se produce una excepción y uno de los bloques `catch` se ejecuta. En el método `nextInt(Scanner, String)`, para cualquier posible lectura, siempre se ejecuta la instrucción `tec.nextLine()` de la cláusula `finally`, permitiendo descartar el salto de línea que se almacena en el buffer de entrada cuando el usuario pulsa la tecla *Enter* o el token incorrecto que hace que se lance la excepción `InputMismatchException`, evitando que el método entre en un bucle infinito. Por ejemplo, supongamos que se introduce desde teclado el texto *"hola"* en lugar de un entero y que se pulsa la tecla *Enter*. En ese caso, el token que se encuentra disponible en el buffer del `Scanner` es el `String` *"hola\n"* pero, al producirse la excepción `InputMismatchException`, dicha lectura no se produce y el primer token disponible sigue siendo el mismo.

4.1. Actividad 2: completar `nextDoublePositive(Scanner, String)`

En la clase `CorrectReading`, el método `nextDoublePositive(Scanner, String)` debe capturar la excepción `InputMismatchException` si el valor introducido por el usuario no es un `double`, de manera similar al método `nextInt(Scanner, String)`, mostrando un mensaje de error apropiado en lugar de abortar la ejecución. En la documentación (comentarios) del método se encuentran ejemplos de los mensajes que deben mostrarse. Completar el método añadiendo el bloque `try-catch-finally` necesario. De este modo, se acaba de añadir un controlador de excepciones para detectar una excepción a nivel local, es decir, en el mismo método en donde se produce el fallo.

4.2. Actividad 3: completar `nextInt(Scanner, String, int, int)`

- En la clase `CorrectReading`, el método `nextInt(Scanner, String, int, int)` debe completarse para que capture la excepción `InputMismatchException` si el valor introducido por el usuario no es un `int`, de manera similar al método `nextInt(Scanner, String)`, mostrando un mensaje de error apropiado en lugar de abortar la ejecución.
- El mismo método, además, ha de controlar que el valor introducido está en el rango `[lowerBound,upperBound]`. Hay dos formas de realizar este control:
  - la primera consiste en añadir una condición apropiada en la guarda del bucle,
  - la segunda en lanzar una excepción.

  En este método se debe optar por esta última; para ello, aprovechando que se dispone de la instrucción `throw`, se debe añadir una instrucción condicional tal que, si el valor introducido no está en el rango anterior, se lance la excepción `IllegalArgumentException` con un mensaje que indique que el valor leído no está en dicho rango. A continuación, añadir una cláusula `catch` para capturar localmente dicha excepción, de forma similar a la captura de la excepción `InputMismatchException`, mostrando el mensaje de la excepción mediante el método `getMessage()` (heredado de la clase `Throwable`).

Se puede comprobar que los métodos de la librería `CorrectReading` funcionan correctamente con la clase `TestCorrectReading` que se proporciona; sin embargo, este test NO dará información de dónde está el fallo, sino sólo si todo es correcto o si no lo es. Para encontrar el error, se deben realizar las pruebas pertinentes, bien escribiendo y ejecutando un programa de prueba, bien con el *CodePad*.

5. Propagación de excepciones vs tratamiento in situ

En esta sección se va a completar el método `handleLine(String)` que extrae el día, mes y valor de una línea, para que propague las excepciones que se puedan producir parando la ejecución del método. A continuación se exploran dos formas de tratar las excepciones que produce el método `handleLine(String)`:

1. Usando el método `add(Scanner)` que, a su vez, propaga estas mismas excepciones *unchecked* y que, finalmente, serán tratadas en el método `testUnreportedSort` de la clase `TestSortedRegister` que habrá que completar para dar información al usuario de los errores encontrados.

2. Tratando localmente las excepciones en el método `add(Scanner, PrintWriter)` que habrá que completar para que las trate *in situ* y escriba, si procede, el correspondiente mensaje de error en el `PrintWriter` que tiene como parámetro en lugar de propagarlas.

5.1. Actividad 4: probar la clase `SortedRegister`

Examinar el código de la clase `SortedRegister.java` que se ha descargado de *PoliformaT*: estructura de los datos y métodos `add(Scanner)`, `handleLine(String)` y `save(PrintWriter)`.

Examinar también la clase `TestSortedRegister` que contiene un método `main` y un método `testUnreportedSort` que sirve para probar los métodos de `SortedRegister`. El método `main` se encarga de leer un año correcto dentro de un intervalo y el nombre de un fichero de datos, abrir un `Scanner` y un `PrintWriter` a partir de los ficheros de datos y de resultados (`result.out`), respectivamente, y usar el método `testUnreportedSort` para realizar el procesamiento de los datos.

Probar a ejecutar la clase introduciendo como datos el año 2016, y el fichero `data.txt`. Comparar el contenido del fichero introducido con el creado como resultado de la ejecución.

5.2. Actividad 5: completar el método `handleLine(String)`

En la ejecución anterior, el método `handleLine(String)`, invocado desde el método `add(Scanner)` de la clase `SortedRegister`, no ha detectado que el fichero contenía, entre las líneas que se muestran a continuación (correspondientes al 30 de abril), una línea con un dato negativo:

```
....
30  4  2
....
30  4  -1
....
```

produciéndose en `result.out` la línea resultado de sumar las cantidades `2` y `-1`:

```
30  4  1
```

Sin embargo, en un caso como este, el método debería haber rechazado el fichero de datos e interrumpido el proceso.

Para corregirlo, modificar el método `handleLine(String)` de la clase `SortedRegister` de modo que, cuando la cantidad leída sea negativa, no la acumule en la matriz sino que lance una excepción de la clase predefinida `IllegalArgumentException` (derivada de `RuntimeException`), con el mensaje `cantidad negativa`.

Volver a repetir la ejecución anterior y comprobar que ahora se produce el error correspondiente, quedando vacío `result.out`.

5.3. Actividad 6: ejecutar la clase programa `TestSortedRegister`

Además del error tratado en la actividad anterior, en el método `handleLine(String)` se producen otras excepciones cuando los datos no cumplen la precondición:

- Si se lee una línea que contiene un número de elementos distinto de 3.
- Si se intenta convertir uno de los elementos a entero cuando no lo es, el método `parseInt(String)` de `Integer` lanza la excepción `NumberFormatException`, como sucede por ejemplo con la línea

  ```
  30  abril  2
  ```

- Si después de haber leído `month` y `day`, los rangos de estos índices no corresponden a una fecha correcta, se lanza una excepción `ArrayIndexOutOfBoundsException` primero, si son valores iguales a `0` ya que no se está usando la posición `0` de los arrays, o de forma automática al intentar acceder a la componente correspondiente de la matriz si están, efectivamente, fuera de rango, como sucede por ejemplo si se lee la línea

  ```
  29  2  1
  ```

  y el año no es bisiesto.

El método `add(Scanner)` no trata estas excepciones sino que, a su vez, las propaga y, de hecho, en sus comentarios de cabecera y en los del mètodo `handleLine(String)`, aparecen documentadas estas excepciones. Antes que nada, para completar la documentación de estos métodos, añadir la siguiente información:

```
@throws IllegalArgumentException si se lee alguna cantidad negativa.
```

Notar que al tratarse de excepciones no comprobadas, no ha sido preciso añadir ninguna cláusula de propagación en el perfil del método. Para hacer pruebas de estas excepciones, se tiene el fichero `badData.txt` que contiene, entre otras, las líneas que se muestran:

```
....
29  2  1
....
30 abril 2
....
```

Probar ahora a ejecutar `TestSortedRegister` en los siguientes casos:

- Introducir como datos el año 2016 y el fichero `badData.txt`, que debe producir una excepción al alcanzar la línea conteniendo el token `abril`. Comparar el contenido del fichero introducido con el fichero creado como resultado de la ejecución.
- Repetir la ejecución con el mismo fichero pero introduciendo el año 2018, para comprobar que el método se interrumpe tan pronto como se intenta procesar la línea del 29 de febrero.

En resumen, tan pronto como en el test de `add(Scanner)` se detecta que el fichero de datos contiene una línea con alguno de los tipos de errores contemplados, el proceso se interrumpe y no se alcanza la instrucción de escritura en el fichero de resultados, con lo que en este no se llega a escribir nada.

5.4. Actividad 7: tratar remotamente excepciones de `add(Scanner)`

Como se ha visto en las actividades anteriores, el método `add(Scanner)` propaga las excepciones que se producen, provocando que los métodos `testUnreportedSort` y `main` de `TestSortedRegister` las propaguen, a su vez, terminando abruptamente la ejecución del programa.

Para que el usuario del programa reciba mayor información acerca de lo que ha sucedido, se deberá modificar el método `testUnreportedSort` para que capture las excepciones producidas por `add(Scanner)` y, según el caso, escriba en la salida uno de los siguientes mensajes:

```
Fichero incorrecto: cantidad negativa.
Fichero incorrecto: linea incorrecta.
Fichero incorrecto: dato con formato no entero.
Fichero incorrecto: dato con fecha incorrecta.
```

Es importante destacar que las excepciones que propaga el método `add(Scanner)` son de tres tipos, mientras que los casos son cuatro. Para distinguir las dos primeras, en lugar de escribir directamente un mensaje nuevo en pantalla, se escribirá el mensaje incluido en la excepción invocando al método consultor `getMessage()` sobre la misma.

Además, se tiene que tener en cuenta que la excepción `NumberFormatException` es una subclase de `IllegalArgumentException` por lo que, para que sea tratada de forma independiente, el `catch` de la excepción derivada debe estar situado **antes** que el de su clase base.

Comprobar que se obtiene el mensaje correspondiente a cada caso repitiendo las siguientes ejecuciones del programa:

- Introduciendo el año 2016 y el fichero `data.txt`.
- Introduciendo el año 2016 y el fichero `badData.txt`.
- Introduciendo el año 2018 y el fichero `badData.txt`.

5.5. Actividad 8: completar `add(Scanner, PrintWriter)`

En esta actividad se va a sobrecargar el método `add(Scanner)`, añadiendo un nuevo método que, en lugar de rechazar la lectura desde una entrada que contenga alguna línea defectuosa, la lleve a término rechazando únicamente el procesamiento de dichoas líneas erróneas y produciendo un informe de errores.

```
/** Clasifica ordenadamente los datos leídos del Scanner s. Se filtran
 *  los datos que tuvieran algún defecto de formato, emitiendo un informe
 *  de errores.
 *  Precondición: El formato de línea reconocible es
 *        día mes cantidad
 *  en donde día y mes deben ser enteros correspondientes a una fecha válida,
 *  y cantidad debe ser un entero > 0.
 *  La cantidad leída se acumula en el registro que se lleva para el día del mes.
 *  En err se escriben las líneas defectuosas, indicando el número de línea.
 */
public void add(Scanner sc, PrintWriter err)
```

Para ello, este método deberá de ser una modificación del anterior, de forma que:

- Lleve la cuenta del número de línea leída de `sc`.
- Para cada línea de `sc`, en un bloque `try` intente obtener los datos de la línea y acumular la cantidad leída en la componente correspondiente de la fecha leída.
- Capture las excepciones producidas por `handleLine(String)`, escribiendo en `err` una de las siguientes frases según el caso:

  ```
  Linea n: cantidad negativa.
  Linea n: linea incorrecta.
  Linea n: numero incorrecto.
  Linea n: fecha incorrecta.
  ```

  siendo `n` el número de línea en la que se produce la excepción. Se puede ver que las dos primeras corresponden a la misma excepción y, como se ha comentado, para distinguirlas se ha utilizado la orden `e.getMessage()` siendo `e` la excepción capturada.

Para probarlo, añadir a la clase `TestSortedRegisted` un método con perfil:

```
public static void testReportedSort(int year, Scanner in, PrintWriter out,
       PrintWriter err)
```

que cree un `SortedRegister` para `year` y le añada los datos de `in`, escribiendo el resultado ordenado en `out` y los errores encontrados en `err`.

El método `main` de la clase, después de leer el año y abrir el fichero de lectura y el fichero `result.out`, pedirá al usuario una opción válida con el mensaje:

```
Opciones de ordenación:
  1.- Rechazar el fichero si tiene errores.
  2.- Filtrar las líneas erróneas del fichero.
```

En el caso de la opción 1 se probará la ejecución de `testUnreportedSort`. En el caso de la opción 2, se creará un fichero de errores `result.log` y se probará la ejecución de `testReportedSort` que escriba el listado de errores en `result.log`. Recordar que, al terminar, se debe cerrar el `PrintWriter` de escritura de dicho fichero. Se probará a ejecutar el test para dicha opción 2, con los siguientes datos:

- Año 2016 y fichero `data.txt`.
- Año 2018 y fichero `badData.txt`.

Y, finalmente, se comprobará el fichero `result.out` y `result.log` obtenido en cada caso.

6. Evaluación

Esta práctica forma parte del segundo bloque de prácticas de la asignatura que será evaluado en el segundo parcial de la misma. El valor de dicho bloque es de un 60 % con respecto al total de las prácticas. El valor porcentual de las prácticas en la asignatura es de un 20 % de su nota final.