PRG (E.T.S. de Ingeniería Informática) - Curso 2017-2018

Práctica 4. Tratamiento de excepciones y ficheros. Primera parte

(2 sesiones)

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Contexto y trabajo previo

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. Dicho registro puede provenir de una o más áreas (ciudades, provincias, ...), y los datos pueden encontrarse distribuidos en uno o más ficheros de texto, con el formato línea a línea:

```
dia  mes  cantidad
```

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

En un 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.

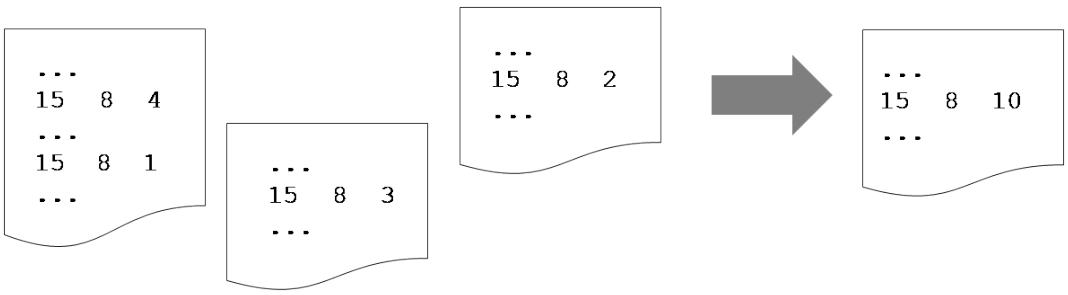

Figura 1: Agregación de datos procedentes de uno o varios ficheros.

Para resolver el problema se proporcionan las siguientes clases:

- La clase `SortedRegister`, que permite abordar el problema usando 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 clase 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.

- 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.

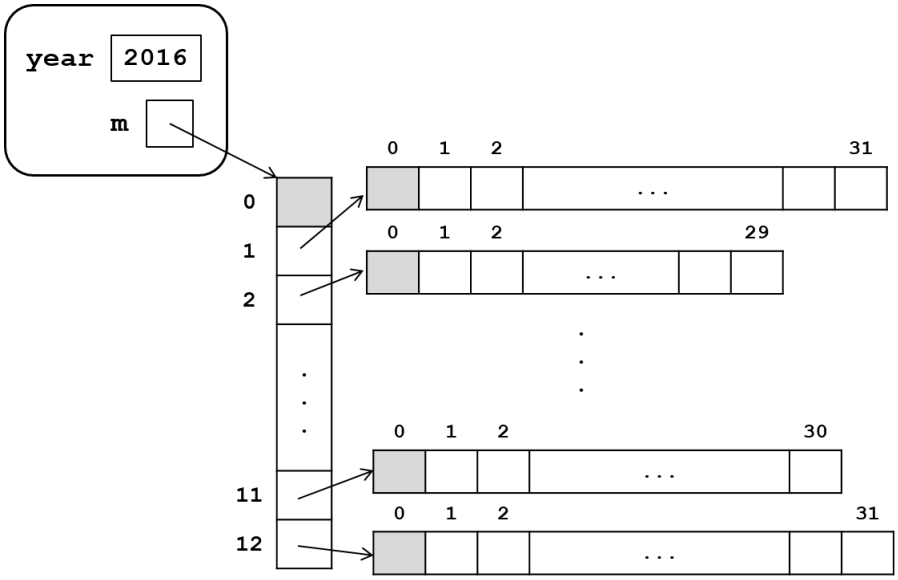

Figura 2: Estructura de un `SortedRegister`.

3. Excepciones comprobadas y no comprobadas

En Java se distingue entre excepciones *checked* o comprobadas, que son de tratamiento obligado (mediante su captura o propagación), y excepciones *unchecked* o no comprobadas.

Las excepciones no comprobadas suelen darse en situaciones muy variadas, y con frecuencia se pueden prever y evitar, por lo que su tratamiento no es obligatorio. No obstante, también es posible tratar estas excepciones no comprobadas de forma que, si acaban sucediendo, el programa se pueda recuperar del fallo o, al menos, terminar de una manera aceptable.

Las excepciones comprobadas surgen en situaciones en las que normalmente no es posible prever y eludir el fallo, como sucede típicamente en problemas de acceso a ficheros.

Esta práctica se ha organizado en dos partes, en la que se tratarán excepciones no comprobadas y comprobadas respectivamente:

- En la primera parte, a lo largo de las actividades planteadas, se irán completando las clases `CorrectReading` y `SortedRegister`. En estas clases pueden aparecer excepciones no comprobadas, principalmente de lectura de datos incorrectos. Dichas excepciones se tratarán bien propagándolas por defecto, o mediante una instrucción `try-catch`.

  En el programa de test que se proporciona, `TestSortedRegister`, el método `main` accede a unos ficheros con los datos a probar en el test y, por simplicidad, las excepciones asociadas al acceso a los ficheros (comprobadas) no se capturan, dejando en la cabecera de `main` la cláusula de propagación obligatoria para tales excepciones.

- En la segunda parte de la práctica se plantea un problema de aplicación de la clase `SortedRegister`, en la que se deberán tratar adecuadamente las excepciones de acceso a ficheros.

4. Detección de errores de ejecución

Actividad 1: preparación del paquete `pract4`

- Crear en `prg` un paquete `pract4` específico para esta práctica.
- Descargar desde `Recursos/Laboratorio/Práctica 4` de `PoliformaT` de `PRG`, los ficheros de código `CorrectReading.java`, `SortedRegister.java` y `TestSortedRegister.java`, y agregarlos al paquete `pract4`.
- Descargar del mismo sitio los ficheros `data2016.txt` y `badData.txt`, y copiarlos en la carpeta del proyecto `prg`.

Actividad 2: detección de errores en tiempo de ejecución en la clase `CorrectReading`

- El método `nextDoublePositive(Scanner, String)` permite realizar la lectura de un valor de tipo `double` ≥ 0, con un bucle `do-while` que se repite mientras el valor leído sea < 0. El segundo argumento de tipo `String` es el mensaje de petición del valor.
- Probar este método en la *zona de código* (*Code Pad*) de *BlueJ*. Para ello, ejecutar las instrucciones siguientes:

```
import java.util.*;
Scanner t = new Scanner(System.in).useLocale(Locale.US);
double realPos = CorrectReading.nextDoublePositive(t, "Valor: ");
```

- Desde la *ventana del terminal* de *BlueJ*, introducir valores reales negativos. Observar que la ejecución del método no acaba hasta que se introduce un valor positivo o cero.
- Probar una ejecución del método en la que erróneamente no se introduzca un token con el formato de un `double`, por ejemplo, tecleando la secuencia de caracteres `23.r4`.

La ejecución se detiene y se muestra un mensaje indicando qué ha ocurrido y en qué instrucción del código. Se trata de un error en tiempo de ejecución o *excepción*. Las secciones que siguen se dedican a cómo gestionar este tipo de errores.

5. Captura de excepciones. Un primer ejemplo

En esta sección se proponen una serie de actividades en las que se tratarán unas excepciones predefinidas en Java mediante su captura utilizando un bloque `try-catch-finally`. Mediante dicho tratamiento, los métodos que se presentan a continuación permiten responder de forma cómoda para el usuario a ciertos errores que este comete cuando teclea datos numéricos.

Actividad 3: análisis del método `nextInt(Scanner, String)`

- Considerar el método `nextInt(Scanner, String)` de la clase `CorrectReading` que permite realizar la lectura de un valor de tipo `int`. La lectura se realiza con el método `nextInt()` de la clase `Scanner`, que puede lanzar la excepción `InputMismatchException` si el valor introducido por el usuario no es un entero.

- Consultar la documentación del método `nextInt()` (y las excepciones que puede generar) en el API de Java de la clase `Scanner`:
`http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`

- Consultar también la documentación de la clase `InputMismatchException`, comprobando que es una excepción *unchecked* (derivada de `RuntimeException`) y que su situación en la jerarquía de clases coincide con la que se muestra a continuación, por lo que Java no obliga a prever su tratamiento.

```
java.lang.Object
 |
 +--java.lang.Throwable
     |
     +--java.lang.Exception
         |
         +--java.lang.RuntimeException
             |
             +--java.util.NoSuchElementException
                 |
                 +--java.util.InputMismatchException
```

- No obstante, el método `nextInt(Scanner, String)` captura (`catch`) este tipo de excepción, mostrando un mensaje de error para indicar al usuario qué acción correctiva es necesaria. En la *zona de código* de *BlueJ*, ejecutar el método `nextInt(Scanner, String)` y, desde la *ventana del terminal*, introducir un valor no entero, por ejemplo, un valor real. Observar el mensaje mostrado y que la ejecución no acaba hasta que no se haya introducido un valor entero.

- 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.

Actividad 4: tratamiento de excepciones en `nextDoublePositive(Scanner, String)`

- Completar el método `nextDoublePositive(Scanner, String)` de la clase `CorrectReading` para que capture 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.

Con ello 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.

Actividad 5: tratamiento de excepciones en `nextInt(Scanner, String, int, int)`

- Completar el método `nextInt(Scanner, String, int, int)` de la clase `CorrectReading` 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.
- Este método, además, ha de controlar que el valor introducido está en el rango `[lInferior,lSuperior]`. Hay dos formas de realizar este control: la primera consiste en añadir la condición apropiada en la guarda del bucle, como en el caso del método `nextDoublePositive(Scanner, String)`, y la segunda en lanzar una excepción. Optamos por esta última. Añadir una instrucción condicional tal que si el valor introducido no está en el rango anterior, lance la excepción `IllegalArgumentException`, usando una instrucción `throw`, 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`).

6. Propagación de excepciones vs tratamiento in situ

En la clase `SortedRegister` descrita en el apartado 2 nos vamos a encontrar cómo diferentes versiones de un método sobrecargado propagan o tratan localmente las excepciones según el comportamiento que se espere de cada una de ellas.

Actividad 6: prueba de 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)` y `save(PrintWriter)`.

La clase `TestSortedRegister` contiene 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 respectivamente, y usa dicho método para realizar el procesamiento de los datos.

Probar a ejecutar la clase introduciendo como datos el año 2016, y el fichero `data2016.txt`. Comparar el contenido del fichero introducido con el fichero creado como resultado de la ejecución.

Actividad 7: lanzamiento de una excepción en el método `add`

En la ejecución anterior, el método `add` no ha detectado que el fichero contenía las líneas que se muestran a continuación, entre las que hay una con un dato negativo

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

produciéndose en `result.out` la línea

```
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 `add` de la clase `SortedRegister`, de modo que cuando lea una línea de `s`, si la cantidad leída es 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`.

Actividad 8: propagación de excepciones

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

- Si se intenta leer un token que no es un entero, el método `nextInt()` de `Scanner` lanza la excepción `InputMismatchException`, 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` al intentar acceder a la componente correpondiente de la matriz, como sucede por ejemplo si se lee la línea

  ```
  29  2  1
  ```

  y el año no es bisiesto.

El método `add` no las trata, sino que a su vez las propaga, y de hecho, en sus comentarios de cabecera aparecen documentadas estas excepciones.

Antes que nada, para completar la documentación del método, añadir la siguiente información:

```
@throws IllegalArgumentException si se lee de s 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 que el fichero `badData.txt` 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` 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.

Actividad 9: tratamiento remoto de las excepciones de `add`

Como se ha visto en las actividades anteriores, el método `add` 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`, y según el caso, escriba en la salida uno de los siguientes mensajes:

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

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

- Introduciendo el año 2016 y el fichero `data2016.txt`.
- Introduciendo el año 2016 y el fichero `badData.txt`.
- Introduciendo el año 2018 y el fichero `badData.txt`.

Actividad 10: tratamiento in situ de las excepciones

En esta actividad se va a sobrecargar el método `add`, 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 dichos 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 s, 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 `s`.
- Para cada línea de `s`, 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, escribiendo en `err` una de las siguientes frases según el caso:

  ```
  Linea n: Cantidad negativa
  Linea n: Formato incorrecto
  Linea n: Fecha incorrecta
  ```

  siendo `n` el número de línea en la que se produce la excepción.

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 en dicho fichero.

Se probará a ejecutar el test para dicha opción 2, con los siguientes datos:

- Año 2016 y fichero `data2016.txt`.
- Año 2018 y fichero `badData.txt`.

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

7. 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.