PRG (E.T.S. de Ingeniería Informática) - Curso 2019-2020

Práctica 4. Tratamiento de excepciones y ficheros

Segunda parte: Obtención de un registro ordenado de accidentes a partir de un fichero de datos.

(Dos sesiones)

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Planteamiento del problema

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

```
dia  mes  cantidad
```

siendo `dia` y `mes` unos enteros mayores que 0 correspondientes a una fecha del año, y `cantidad` es un entero no negativo correspondiente a un dato de accidentes registrados en esa fecha.

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 uno o más 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 una línea contenga más o menos de tres valores o estos no sean enteros, porque `dia` y `mes` no sean una fecha correcta, o porque `cantidad` sea negativa.

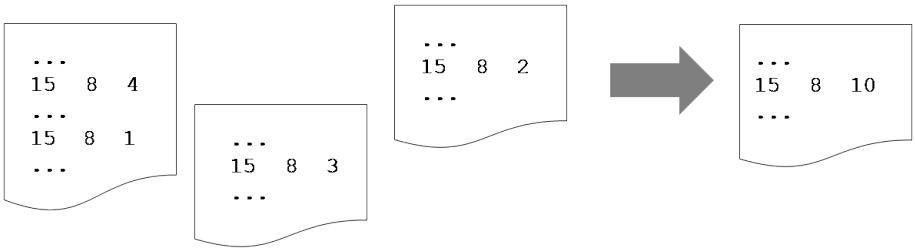


Figura 1: Agregación de datos procedentes de uno o varios ficheros.

Para resolver el problema se va a usar una matriz cuyas filas vengan indexadas por meses, y las columnas vengan indexadas 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. Una vez volcados los datos en la matriz, un recorrido por filas y columnas permite obtener un listado, ordenado por fechas, de los datos que se hubieran acumulado.

2. Las clases de la aplicación

En el material de la práctica se proporciona la clase tipo de datos `SortedRegister`, cuya funcionalidad permite el procesamiento de datos como los anteriores.

Los objetos de la clase `SortedRegister` (ver figura 2) contienen 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]` se usará para almacenar los accidentes acumulados el día `c` del mes `f`. Las filas 1 a 12 se hacen corresponder a los meses del año (la fila 0 no se va a utilizar). Para cada fila, las columnas de la 1 en adelante corresponderán a los días del mes (la columna 0 no se va a utilizar). Notar que la última columna del mes de febrero deberá ser la 28 o la 29, dependiendo de que el año sea o no bisiesto.

Los principales métodos que vienen implementados en la clase son:

- El constructor

  ```
  public SortedRegister(int year)
  ```

  que crea la matriz `this.m` en consonancia con el año dado por `year`.

- Método de perfil

  ```
  public int add(Scanner sc)
  ```

  que, siendo `sc` un `Scanner` abierto a partir de la fuente de texto de los datos, procesa las líneas de `sc` para volcar los datos en `this.m`. Si el proceso termina normalmente (sin

producirse excepciones por formato incorrecto en los datos), devuelve el número de líneas procesadas.

- Método de perfil

```
public void save(PrintWriter pw)
```

que escribe en `pw` los datos acumulados en `this.m`, por orden cronológico.

Para hacer pruebas del comportamiento de `SortedRegister`, leyendo los datos de unos ficheros concretos, se dispone de la clase programa `TestSortedRegister`. En esta clase se usa también la clase de utilidades `CorrectReading` desarrollada en la primera parte de la práctica.

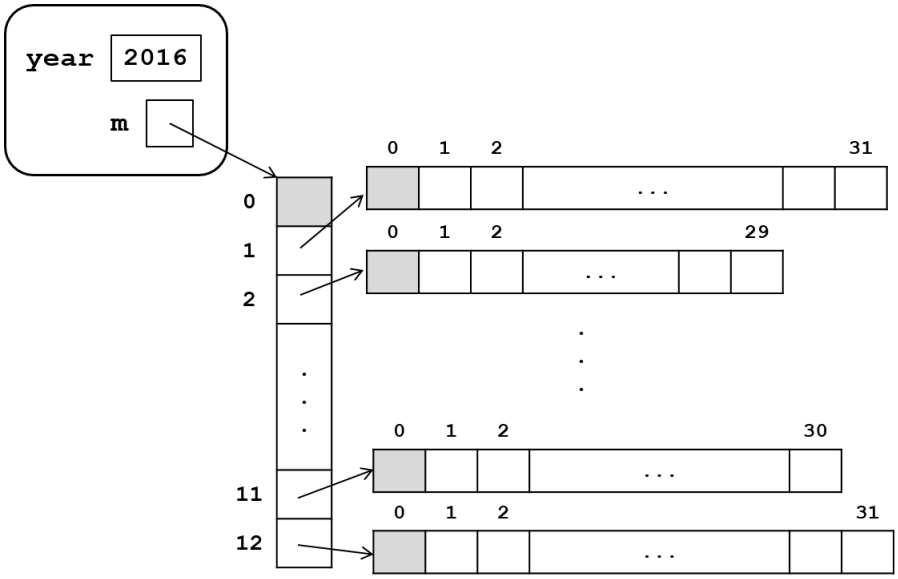


Figura 2: Estructura de un `SortedRegister`.

2.1. Actividad 1: instalación de las clases y ficheros de prueba

- Descargar de la carpeta de material de la práctica 4, los ficheros `SortedRegister.java` y `TestSortedRegister.java` y agregarlos al paquete `pract4`.
- Desde el explorador de archivos y dentro de la carpeta `prg/pract4` crear una nueva carpeta con el nombre `data` donde se dejarán y crearán los ficheros de datos y resultados.
- Descargar del mismo sitio los ficheros de texto que se usarán para hacer pruebas: `data.txt`, `badData1.txt`, `badData2.txt`, `badData3.txt`, `badData4.txt`. Copiarlos en la carpeta `prg/pract4/data`.

Es importante que los ficheros de datos se sitúen en la ubicación correcta: como se podrá comprobar en la siguiente actividad, el código del programa `TestSortedRegister` busca los ficheros con los que hacer las pruebas en la subcarpeta `pract4/data` del proyecto `prg`. Los ficheros de resultados los deja en la misma carpeta `data`.

2.2. Actividad 2: examen y prueba de la clase `SortedRegister`

Examinar el código de la clase `SortedRegister` que se ha descargado de *PoliformaT*: estructura de la matriz de datos, método `add` (que usa un método auxiliar `handleLine` para obtener los datos de cada línea y actualizar con ellos la matriz `this.m`), y método `save`.

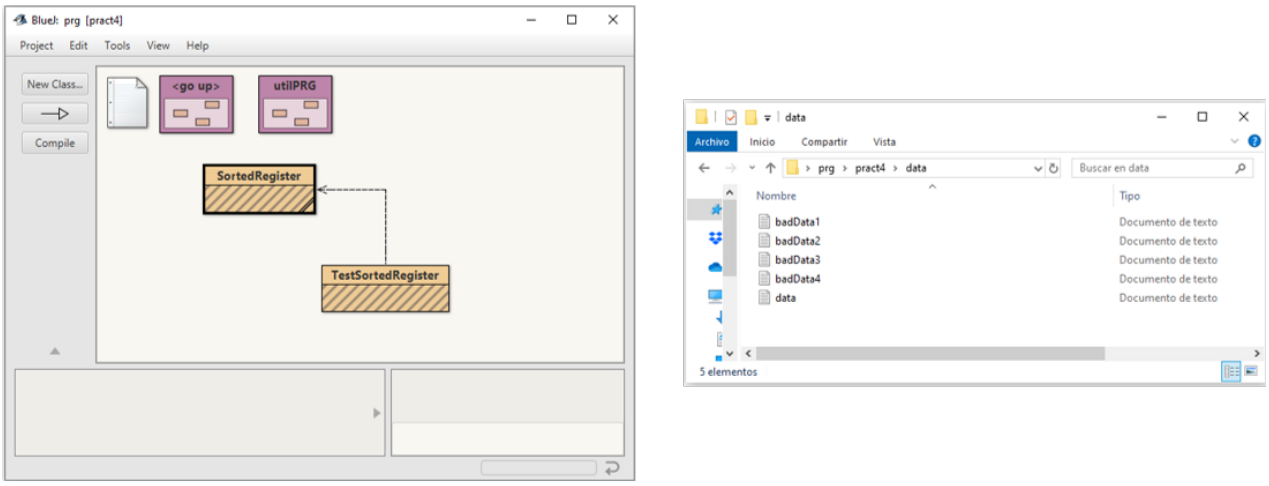


Figura 3: Paquete `pract4` con las clases de la aplicación y carpeta `data` con los ficheros de datos.

Examinar también la clase `TestSortedRegister`, que contiene un método `main` y un método `test1` 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 `test1` para realizar el procesamiento de los datos.

Probar a ejecutar la clase introduciendo como datos el año 2016 (bisiesto), el fichero `data.txt`, y seleccionando la opción 1 del menú (`test1`). Comparar el contenido del fichero introducido con el creado como resultado de la ejecución. Como se muestra en la figura 4, se puede observar que en `result.out` aparecen los datos de `data.txt`, convenientemente acumulados, y en orden cronológico.

data.txt

```
 2   1   1
29   2   1
23   3   1
30   4   2
12   5   2
25   6   2
15   8   3
30   8   1
23   9   2
30   9   1
23  10   6
 1  11   3
 4  12   1
31  12   3
 2   1   1
31   1   2
28   2   3
24   3   1
30   4  -1
12   5   2
15   6   4
30   6   1
23   7   6
31   7   1
15   8   7
31   8   3
30   9   1
23  10   6
 4  12   1
31  12   3
```

result.out

```
 2   1   2
31   1   2
28   2   3
29   2   1
23   3   1
24   3   1
30   4   1
12   5   4
15   6   4
25   6   2
30   6   1
23   7   6
31   7   1
15   8  10
30   8   1
31   8   3
23   9   2
30   9   2
23  10  12
 1  11   3
 4  12   2
31  12   6
```

Figura 4: Contenido de los ficheros de texto `data.txt` y `result.out`.

Atención: al ejecutar el método `main` de `TestSortedRegister` se pueden producir excepciones de entrada/salida si se escribe mal el nombre del fichero, o no se encuentra en la ubicación esperada; estas excepciones se tratarán más adelante, en la sección 4.

3. Gestión de excepciones `RunTimeException`

3.1. Actividad 3: examen y prueba del método `handleLine(String)`

Para procesar cada una de las líneas que el método `add` lee con un `nextLine()`, invoca al método privado auxiliar `handleLine(String)` pasándole la línea como parámetro.

Si examinamos el código de este método auxiliar, vemos que lo primero que hace es aplicar a `line` el método `split` de `String`; esto se hace para "dividir" la línea en los substring que aparezcan separados entre sí por blancos: `split` devuelve un array cuyas componentes son los sucesivos substrings en los que se divide la línea, de modo que si `line` se ajusta al formato el array resultante tendrá exactamente tres componentes.

Si se han podido extraer correctamente los enteros `day`, `month`, `amount`, la cantidad leída se almacena en la componente `this.m[month][day]` de la matriz.

Notar que el método propaga las excepciones correspondientes a las siguientes situaciones de error:

1. Si la línea dividida mediante el método `split` no tiene exactamente tres datos, se crea y se lanza una excepción de clase `IllegalArgumentException` con el mensaje `"La linea no contiene tres datos."`.

2. Si alguno de los tres datos de la línea no se puede transformar a `int`, se propaga la excepción `NumberFormatException` debida al método `Integer.parseInt`.

3. Si `day` y `month` no son índices ≥ 1 correspondientes a una componente de la matriz `this.m`, se crea y se lanza una excepción `IllegalArgumentException` con el mensaje `"Fecha incorrecta."`. Cabe recordar que el constructor de `SortedRegister` crea la matriz `this.m` con las filas 1 a 12 correspondientes a los meses del año, y para cada una de estas filas, sus columnas desde la 1 en adelante, correspondientes a los días del mes.

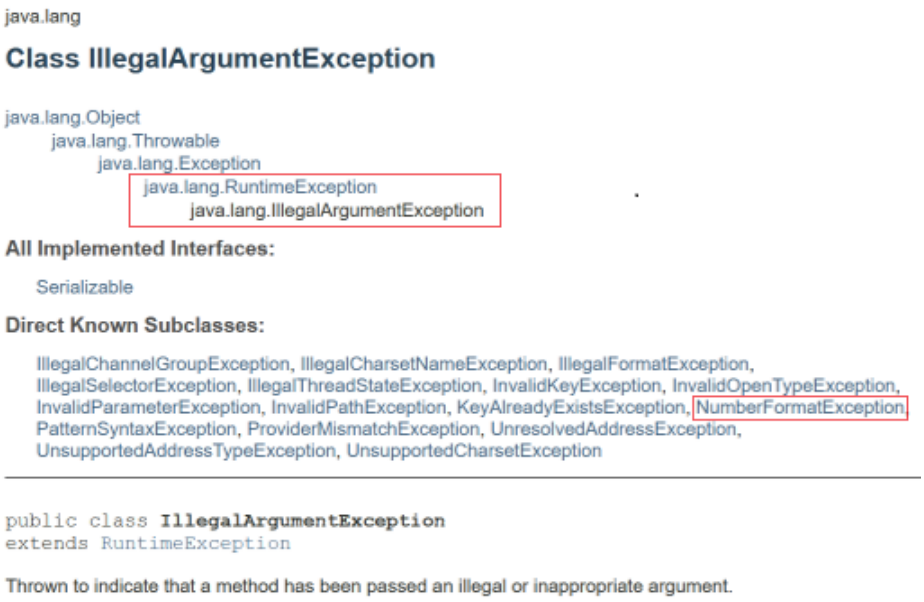
Para las situaciones 1 y 3 anteriores, se ha escogido crear una excepción no comprobada de clase `IllegalArgumentException` cuyo uso, como se ve en la documentación de la figura 5, está indicado cuando a un método se le pasa un argumento inapropiado. El mensaje con que se crea permite distinguir entre estas dos situaciones.

Para comprobar la propagación de las excepciones a `add`, y de ahí al programa, se hará la siguiente prueba:

Volver a ejecutar el `main` de la clase `TestSortedRegister`, opción 1, pero pasando como datos el año 2016 y el fichero `badData1.txt`. Observar cuál es la excepción que se produce (propagada por `handleLine`).

A continuación, examinar el contenido de `badData1.txt`, y comprobar que la primera línea errónea que contiene el fichero se corresponde con dicha excepción.

Comprobar además que `result.out` ha quedado vacío, dado que `test1` se interrumpe sin alcanzar la instrucción de escritura en el fichero de resultados.



java.lang

Class IllegalArgumentException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IllegalArgumentException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

IllegalChannelGroupException, IllegalCharsetNameException, IllegalFormatException, IllegalSelectorException, IllegalThreadStateException, InvalidKeyException, InvalidOpenTypeException, InvalidParameterException, InvalidPathException, KeyAlreadyExistsException, NumberFormatException, PatternSyntaxException, ProviderMismatchException, UnresolvedAddressException, UnsupportedAddressTypeException, UnsupportedCharsetException

---

```
public class IllegalArgumentException
extends RuntimeException
```

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Figura 5: Fragmento de la documentación de `IllegalArgumentException`.

3.2. Actividad 4: finalización del método `handleLine(String)`

El método `handleLine` que se ha examinado y probado en la actividad anterior está incompleto. En efecto, reconsiderar la ejecución de prueba de la actividad 3 anterior, y cuyo resultado se mostraba en la figura 4. Se aprecia que en `data.txt` aparecen las siguientes dos líneas para el 30 de abril:

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

El procesamiento de ambas ha producido en `result.out` la línea resultante de sumar las cantidades `2` y `-1`:

```
30  4  1
```

Es decir, el método `handleLine` ha olvidado la detección de una cantidad errónea (el número de accidentes acaecidos en una fecha no puede ser menor que 0).

Así pues, para que `handleLine` contemple todos los errores posibles, añadir en el método una instrucción que, antes de pasar a almacenar en la matriz la cantidad leída, compruebe si es negativa, y en ese caso cree y lance una excepción `IllegalArgumentException` con el mensaje `"Cantidad negativa."`.

Volver a ejecutar el `main` de la clase `TestSortedRegister`, opción 1, pasando de nuevo como datos el año 2016 y el fichero `data.txt`. Comprobar que, en cuanto se procese la línea con la cantidad negativa, se produzca la excepción correspondiente (figura 6).

3.3. Actividad 5: captura de excepciones en el método `add(Scanner)`

El método `add` implementado en la clase tiene como precondición que las líneas que se leen del `Scanner` se ajusten al formato establecido. Como se ha visto en las actividades anteriores,

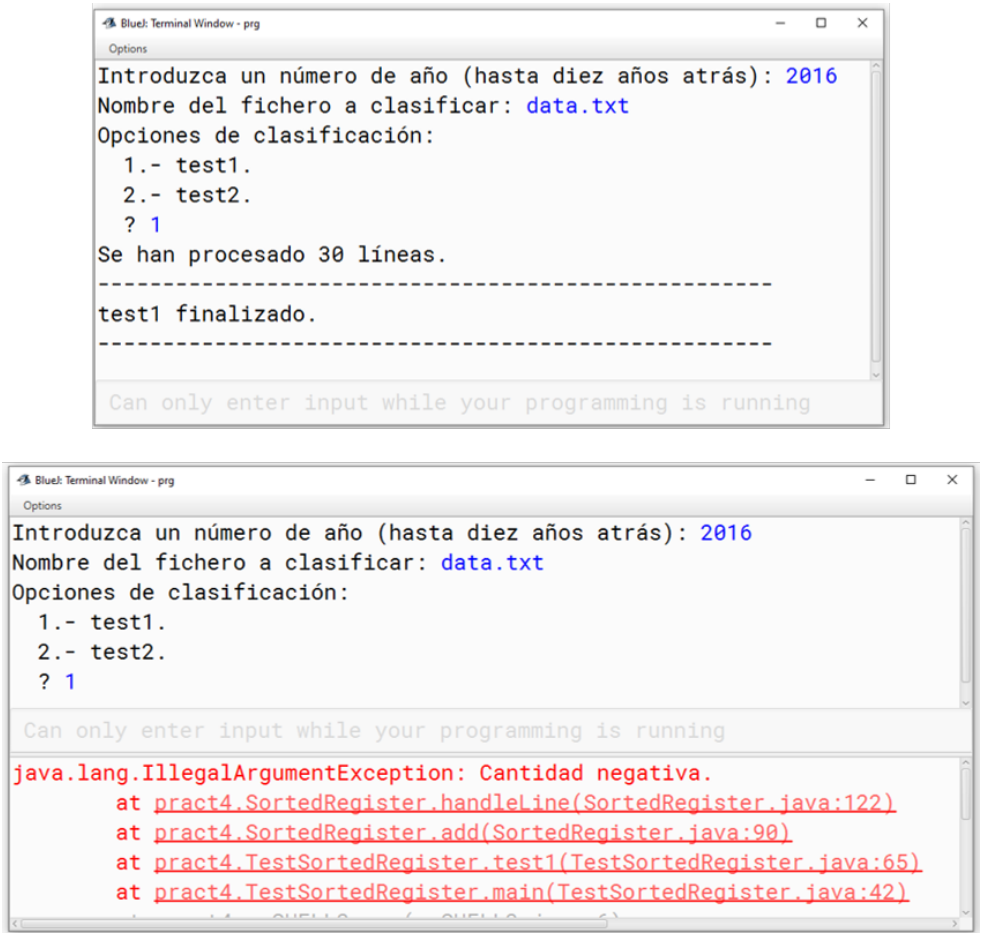


Figura 6: Creación y propagación de la excepción debida a una cantidad < 0: la ejecución de arriba es previa a completar `handleLine` según la actividad 4, y la de abajo es posterior.

si se lee una línea incorrecta, el método se limita a propagar la excepción que le llega de `handleLine`; en el programa de prueba `TestSortedRegister`, ello provoca que los métodos `test1` y `main` las propaguen a su vez, con lo que se termina abruptamente la ejecución.

Deseamos refinar el comportamiento del método para que termine normalmente (sin propagar ninguna excepción) en cualquier caso, de manera que:

- Si todas las líneas son correctas, el método devolverá el número de líneas procesadas.
- En caso contrario, en cuanto detecte que la fuente del `Scanner` tiene alguna línea con defecto de formato, interrumpirá el procesamiento de los datos y terminará devolviendo -1. Antes de terminar, deberá escribir en la salida estándar, uno de los siguientes mensajes, según el error detectado:

```
ERROR. Linea n: La linea no contiene tres datos.
ERROR. Linea n: Dato no entero.
ERROR. Linea n: Fecha incorrecta.
ERROR. Linea n: Cantidad negativa.
```

  siendo `n` el número de línea en el que se ha detectado el error.

Para ello, el cuerpo del método deberá seguir una estructura como la que se muestra a continuación.

```
int count = 0;
try {
    ...
} catch (NumberFormatException e) {
    System.out.println("ERROR. Linea " + count + ": Dato no entero.");
    count = -1;
} catch (IllegalArgumentException e) {
    System.out.println("ERROR. Linea " + count + ": " + e.getMessage());
    count = -1;
}
return count;
```

Es importante tener en cuenta que la excepción `NumberFormatException` es una derivada de `IllegalArgumentException` (ver figura 5) por lo que el compilador exige que el `catch` de la excepción derivada esté situado **antes** que el de su clase base.

Los comentarios de documentación del método deberán reflejar estos cambios (indicar que el método interrumpe la lectura de datos si detecta una línea errónea y termina devolviendo -1, y eliminar las frases etiquetadas con `@throws`).

Con esta modificación, la repetición de la ejecución del `test1` con el fichero `data.txt` termina normalmente, como sucede en el ejemplo de la figura 7. Observar que en el código de `test1`, si `c.add(sc)` devuelve un valor negativo, entonces no se vuelcan los datos de `c` en el `PrintWriter out` (queda vacío).

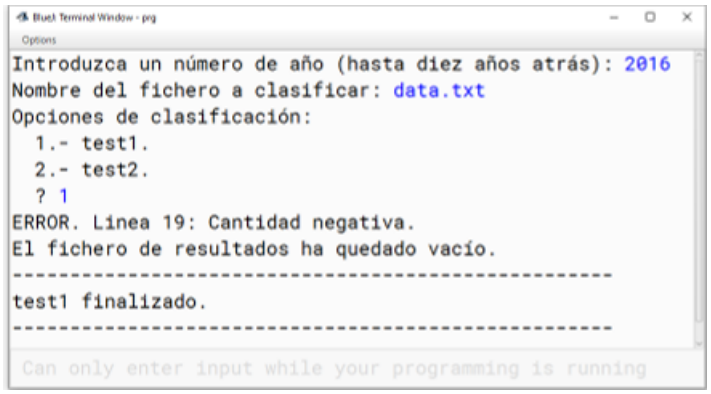


Figura 7: Captura de excepciones en `add(Scanner)`: si aparece una línea errónea, `add` no propaga la excepción correspondiente.

Para comprobar el funcionamiento del método se han dejado los ficheros `badData1.txt`, `badData2.txt`, `badData3.txt` y `badData4.txt`. Los cuatro contienen la misma información, con las mismas cuatro líneas incorrectas (líneas número 4, 5, 6 y 7), aunque en diferente orden. De este modo, la primera línea errónea de cada uno se ha hecho corresponder con uno de los cuatro casos de error. Se deberá probar la opción 1 del `main` de `TestSorteRegister`, y observar el mensaje de error que se escribe:

- Introduciendo el año 2016 y el fichero `badData1.txt`.
- Introduciendo el año 2016 y el fichero `badData2.txt`.
- Introduciendo el año 2016 y el fichero `badData3.txt`.
- Introduciendo el año 2016 y el fichero `badData4.txt`.

Se puede editar `data.txt` para eliminar la línea incorrecta con una cantidad negativa y obtener un fichero sin errores. Si se repite la prueba, el método debe procesar correctamente el fichero con las líneas restantes.

3.4. Actividad 6: desarrollo del método `add(Scanner, PrintWriter)`

El método `add` de `SortedRegister` está sobrecargado por el método de perfil

```
public int add(Scanner sc, PrintWriter err)
```

a falta de que se complete su código. Como antes, `sc` es la fuente de los datos, pero ahora se procesarán todas sus líneas, filtrando las erróneas, de las que se dejará registro en `err`.

En concreto, este método deberá de ser una modificación del anterior, de forma que, para todas y cada una de las líneas de `sc`:

- Intente obtener los datos de la línea y acumular la cantidad leída en la matriz, valiéndose del método `handleLine`.
- Capture las excepciones producidas por `handleLine`, escribiendo en `err` una de las siguientes frases según el caso:

  ```
  ERROR. Linea n: La linea no contiene tres datos.
  ERROR. Linea n: Dato no entero.
  ERROR. Linea n: Fecha incorrecta.
  ERROR. Linea n: Cantidad negativa.
  ```

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

De esta forma, en `this.m` habrán quedado los datos de las líneas que se ajustan al formato, y en `err` se habrán escrito los mensajes de error. El método terminará devolviendo el número total de líneas procesadas, correctas e incorrectas.

Para probarlo, completar el método `test2` de la clase `TestSortedRegisted`. El método deberá ser análogo a `test1`, excepto que deberá usar el método de perfil `add(Scanner, PrintWriter)` completado en esta actividad, y la escritura en `out` de los datos ya clasificados se deberá ejecutar siempre. Una vez completado `test2`, se podrán hacer pruebas como la de la figura 8.

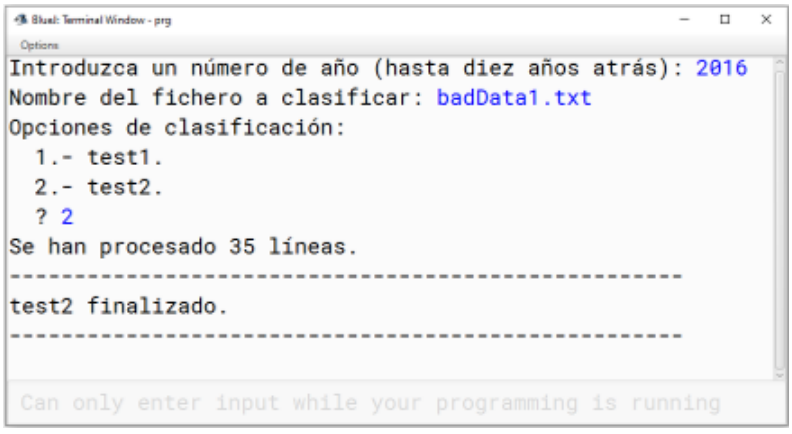


Figura 8: Prueba de la ejecución de `add(Scanner, PrintWriter)`.

Para esta ejecución debería obtenerse unos ficheros `result.out` y `result.log` como los de la figura 9.

badData1.txt

```
 2   1   1
 1   6   1
23   3   1
30   1   4   5
30  abril   2
31   9   2
30   4  -1
12   5   2
25   6   2
15   8   3
30   8   1
23   9   2
30   9   1
23  10   6
 1  11   3
 4  12   1
31  12   3
 2   1   1
31   1   2
28   2   3
24   3   1
29   2   1
12   5   2
15   6   4
30   6   1
23   7   6
31   7   1
15   8   7
31   8   3
30   9   1
23  10   6
 4  12   1
31  12   3
16   5   1
 2   1   2
```

result.out

```
 2    1    4
31    1    2
28    2    3
29    2    1
23    3    1
24    3    1
12    5    4
16    5    1
 1    6    1
15    6    4
25    6    2
30    6    1
23    7    6
31    7    1
15    8   10
30    8    1
31    8    3
23    9    2
30    9    2
23   10   12
 1   11    3
 4   12    2
31   12    6
```

result.log

```
ERROR. Linea 4: La linea no contiene tres datos.
ERROR. Linea 5: Dato no entero.
ERROR. Linea 6: Fecha incorrecta.
ERROR. Linea 7: Cantidad negativa.
```

Figura 9: Resultado de la prueba de la figura 8.

4. Gestión de excepciones `IOException`

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 (`RuntimeException` y sus derivadas). 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.

En el método `main` del programa `TestSortedRegister`, puede suceder la excepción comprobada `FileNotFoundException` si se da algún problema al abrir el `Scanner` y los `PrintWriter` que usarán los tests (no se tiene permiso para hacerlo, no se introduce bien el nombre del fichero de datos o no está en la ubicación correcta, no queda suficiente memoria para grabarlos, etc.).

En esta sección se desea capturar estas excepciones para evitar una finalización abrupta del programa.

4.1. Actividad 7: captura de excepciones `FileNotFoundException` en el programa `TestSortedRegister`

Examinar el código del método `main` de `TestSortedRegister`. Se puede comprobar que en dicho método no se realiza ninguna captura de las posibles excepciones `FileNotFoundException`, por lo que el compilador obliga a que en el perfil del `main` aparezca la cláusula `throws FileNotFoundException`.

En esta actividad se debe cambiar el código de `main` para no propagar la excepción que se pueda producir. Para ello, una vez leídos de teclado el año y el nombre del fichero de los datos, se deberá incorporar la gestión de excepciones que viene a continuación.

```
...
Scanner in = null; PrintWriter out = null, err = null;
File f = new File("pract4/data/" + nameIn);
try {
    in = new Scanner(f);
    f = new File("pract4/data/" + "result.out");
    out = new PrintWriter(f);
    f = new File("pract4/data/" + "result.log");
    err = new PrintWriter(f);
    ...
    // Selección y ejecución del test de prueba
    ...
} catch (FileNotFoundException e) {
    System.out.println("Error al abrir el fichero " + f);
} finally {
    if (in != null) { in.close(); }
    if (out != null) { out.close(); }
    if (err != null) { err.close(); }
}
```

Es importante remarcar que la máquina virtual ejecuta siempre el código del bloque `finally`. Así aseguramos que, en cualquier circunstancia, se cierran todos los ficheros que hubieran sido abiertos en el bloque `try`.

Una vez incorporada en el método esta captura de excepciones, se puede eliminar la cláusula `throws` de la cabecera de `main`.

En la figura 10 se muestra cuál es el comportamiento de `TestSortedRegister` sin gestión de la excepción `FileNotFoundException`, y cuál debe ser después de introducir dicha gestión.

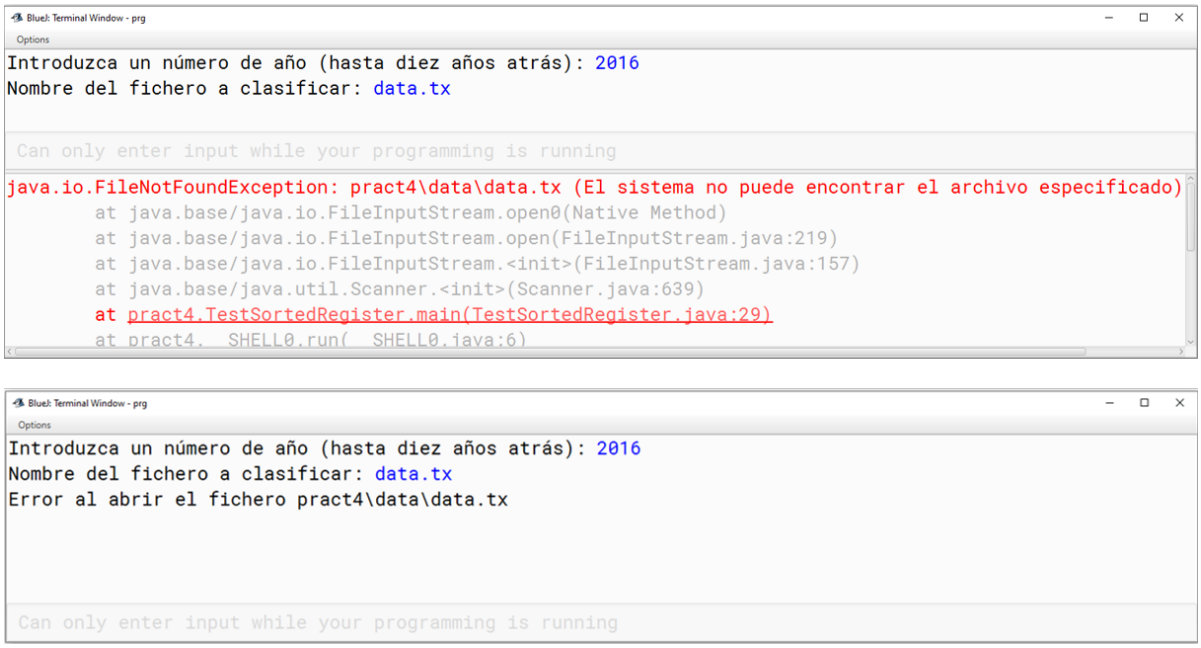


Figura 10: Gestión de la excepción `FileNotFoundException`: la ejecución de arriba es previa a completar el `main` de `TestSortedRegister` según la actividad 7, y la de abajo es posterior.