Ejercicios de Excepciones e hilos

Índice

1. Captura de excepciones

El fichero `Ej1.java` es un programa que toma un número como parámetro, y como salida muestra el logaritmo de dicho número. Sin embargo, en ningún momento comprueba si se ha proporcionado algún parámetro, ni si ese parámetro es un número. Se pide:

a) Compilar el programa y ejecutadlo de tres formas distintas:

- Sin parámetros

```
java Ej1
```

- Poniendo un parámetro no numérico

```
java Ej1 pepe
```

- Poniendo un parámetro numérico

```
java Ej1 30
```

Anotad las excepciones que se lanzan en cada caso (si se lanzan)

b) Modificar el código de `main` para que capture las excepciones producidas y muestre los errores correspondientes en cada caso:

- Para comprobar si no hay parámetros se capturará una excepción de tipo `ArrayIndexOutOfBoundsException` (para ver si el *array* de `String` que se pasa en el `main` tiene algún elemento).
- Para comprobar si el parámetro es numérico, se capturará una excepción de tipo `NumberFormatException`.

Así, tendremos en el `main` algo como:

```
try
{
        // Tomar parámetro y asignarlo a un double
} catch (ArrayIndexOutOfBoundsException e1) {
        // Codigo a realizar si no hay parametros
} catch (NumberFormatException e2) {
        // Codigo a realizar con parametro no numerico
}
```

Probad de nuevo el programa igual que en el caso anterior comprobando que las excepciones son capturadas y tratadas.

2. Lanzamiento de excepciones (*)

El fichero `Ej2.java` es similar al anterior, aunque ahora no vamos a tratar las excepciones del `main`, sino las del método `logaritmo`: en la función que calcula el logaritmo se comprueba si el valor introducido es menor o igual que 0, ya que para estos

valores la función logaritmo no está definida. Se pide:

a) Buscar entre las excepciones de Java la más adecuada para lanzar en este caso, que indique que a un método se le ha pasado un argumento ilegal. (Pista: Buscar entre las clases derivadas de `Exception`. En este caso la más adecuada se encuentra entre las derivadas de `RuntimeException`).

b) Una vez elegida la excepción adecuada, añadir código (en el método `logaritmo`) para que en el caso de haber introducido un parámetro incorrecto se lance dicha excepción.

```
throw new ... // excepcion elegida
```

Probar el programa para comprobar el efecto que tiene el lanzamiento de la excepción.

c) Al no ser una excepción del tipo *checked* no hará falta que la capturemos ni que declaremos que puede ser lanzada. Vamos a crear nuestro propio tipo de excepción derivada de `Exception` (de tipo *checked*) para ser lanzada en caso de introducir un valor no válido como parámetro. La excepción se llamará `WrongParameterException` y tendrá la siguiente forma:

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

Deberemos lanzarla en lugar de la escogida en el punto anterior.

```
throw new WrongParameterException(...);
```

Intentar compilar el programa y observar los errores que aparecen. ¿Por qué ocurre esto? Añadir los elementos necesarios al código para que compile y probarlo.

d) Por el momento controlamos que no se pase un número negativo como entrada. ¿Pero qué ocurre si la entrada no es un número válido? En ese caso se producirá una excepción al convertir el valor de entrada y esa excepción se propagará automáticamente al nivel superior. Ya que tenemos una excepción que indica cuando el parámetro de entrada de nuestra función es incorrecto, sería conveniente que siempre que esto ocurra se lance dicha excepción, independientemente de si ha sido causada por un número negativo o por algo que no es un número, pero siempre conservando la información sobre la causa que produjo el error. Utilizar *nested exceptions* para realizar esto.

Ayuda

Deberemos añadir un nuevo constructor a `WrongParameterException` en el que se proporcione la excepción que causó el error. En la función `logaritmo` capturaremos cualquier excepción que se produzca al convertir la cadena a número, y lanzaremos una excepción `WrongParameterException` que incluya la excepción causante.

3. Terminación de hilos

En la clase `Ej3` se crean hilos utilizando la interfaz `Runnable`. De esta forma podremos crear múltiples hilos que ejecuten el mismo método `run` de esta clase accediendo al mismo espacio de memoria de este objeto. Ejecutar el programa, ver lo que hace, consultar el código fuente y contestar a las siguientes preguntas:

a) Explica que es lo que hace la condición de terminación del bucle `while` en el método `run`. ¿Qué utilidad le ves?

b) ¿Puede haber en algún momento dos hilos ejecutando simultáneamente el método `run`?

c) ¿Si hubiese dos hilos ejecutando el `run`, podría haber conflictos en el acceso a la variable `ini`? Es decir, que un hilo sobrescriba el valor que había escrito en ella el otro hilo ¿Por qué?

d) ¿Cómo podemos parar este hilo sin crear uno nuevo?

e) Escribir un programa `Ej3b` que realice una función similar pero utilizando temporizadores en lugar de hilos. No es necesario que sea exactamente igual al anterior programa, simplemente debe ser similar.

Ayuda

Se deberá crear una `TimerTask` que guarde en un campo privado el instante de tiempo en el que fue creada. Cada vez que se ejecute esta tarea deberá imprimir ese instante. Desde el programa principal crearemos la tarea, y la programaremos en un temporizador que la ejecute periódicamente. Transcurridos 5 segundos, el programa principal deberá cancelar el temporizador.

4. Carrera de hilos (*)

En la clase `Ej4` tenemos un programa que muestre una carrera entre tres hilos de distintas prioridades. Se pide:

a) Cada hilo tiene su propio contador que se va incrementando cada iteración de dicho hilo. En este caso, ¿podrá haber conflicto en el acceso al contador del hilo entre los distintos hilos? ¿Por qué?

b) Fíjate en el bucle principal en el que se imprime el contador de cada hilo, ¿cuál es su condición de terminación?

c) En cada iteración los hilos hacen una operación costosa. En este caso estamos forzando a que llamen al colector de basura (*garbage collector*) con la instrucción `System.gc()`. Sustituir esta instrucción por dormir durante 100ms y probar. ¿Qué ocurre en este caso? ¿Por qué? Volver a dejar el programa como antes, con la llamada al colector de basura.

d) En el método `run` de `Hilo`, añade la instrucción necesaria para que en el caso de que el hilo `t` sea distinto de `null` se quede bloqueado hasta que dicho hilo termine su ejecución. Probar el programa y ver lo que ocurre en este caso.

e) Prueba cambiando las prioridades de los hilos. Cuando tenemos hilos de alta prioridad e hilos de baja prioridad, cuando los de alta prioridad terminan, ¿qué ocurre con los de baja prioridad? ¿por qué?

5. Productor y consumidor (*)

En este ejercicio vamos a resolver el problema de los productores y los consumidores. Vamos a definir 3 clases: el hilo `Productor`, el hilo `Consumidor`, y el objeto `Recipiente` donde el productor deposita el valor producido, y de donde el consumidor extrae los datos.

- El productor se ejecuta durante 10 iteraciones y en cada una de ellas deposita en el recipiente el número de la iteración actual. Entre iteración e iteración se quedará durmiendo durante un tiempo aleatorio entre 1 y 2 segundos.
- El consumidor se ejecuta el mismo número de iteraciones que el productor, pero en cada una de ellas saca el valor almacenado en el recipiente y lo muestra por pantalla. Entre cada iteración duerme también un tiempo aleatorio entre 1 y 2 segundos.
- El recipiente proporciona los métodos `produce` y `consume` para depositar un dato en él y para sacarlo de él respectivamente.

El programa mostrará cuando el productor produce un valor y cuando el consumidor lo consume. El funcionamiento correcto debería ser que el consumidor consuma exactamente los mismos valores que el productor ha producido, sin saltarse ninguno ni repetirlos. Se pide:

a) Compilar y probar el programa. ¿Funciona correctamente? ¿Por qué? Ejecutar varias veces y explicar lo que pasa. ¿Qué tendremos que hacer para que funcione correctamente?

b) Vamos a añadir el código necesario en los métodos `produce` y `consume` para sincronizar el acceso a ellos. El comportamiento debería ser el siguiente:

- Si queremos producir y todavía hay datos disponibles en el recipiente, esperaremos hasta que se saquen, si no produciremos y avisamos a posibles consumidores que estén a la espera.
- Si queremos consumir y no hay datos disponibles en el recipiente, esperaremos hasta que se produzcan, si no consumimos el valor disponible y avisamos a posibles productores que estén a la espera.

¿Qué métodos utilizaremos para la sincronización? Insertar el código necesario y compilar. Probar el programa, ¿da alguna excepción? En caso afirmativo, ¿por qué? ¿será necesario añadir algo más en el encabezado de los métodos `produce` y `consume`? Hacer las modificaciones necesarias.

Compilar y comprobar que el programa funciona correctamente.