EXCEPCIONES

1.- Manejo de Excepciones básicas

Una de las características de Java, que facilita enormemente el tratamiento de errores, y contribuye a que se creen programas más robustos, más claros y que toleren mejor los fallos, con relativamente poco esfuerzo por parte del programador, es el manejo de excepciones, que controla la aparición de errores durante la ejecución de un programa.

Hay ocasiones en las que de alguna manera, podemos esperar un error inevitable y que al menos, debemos gestionarlo para evitar la ruptura del programa y orientar al usuario para que no vuelva a aparecer ese error. Es necesario comprobar determinadas circunstancias que pueden provocar fallos en el programa. Como ejemplos, puedes pensar en la necesidad de controlar que no se realicen divisiones por cero, o evitar que se acceda a una posición de un vector que no existe, o impedir que se intente comprobar un carácter de un String más allá del último (o anterior al primero), introduciendo un valor negativo para la posición dentro del String o el índice del vector.

También tenemos que comprobar cada vez que leemos un número desde teclado que lo que se teclea se puede considerar un número, y se puede convertir en número sin que el programa aborte cuando metemos caracteres extraños. Por ejemplo Xt56?e4 no se podrá convertir en número, por más que nos empeñemos, así que tenemos que asegurarnos de que cuando se espere un número y se teclee esa cadena, el error no va a suponer que nuestro programa termine bruscamente, abortando.

En general existen dos planteamientos posibles, a la hora de manejar los errores:

• Prevenirlos. Algo muy común. Supone incluir líneas de código adicionales, intercaladas con el código de la aplicación, en los puntos en los que pueden ocurrir los errores, de forma que prevengan y eviten esos errores. Esto tiene ventajas e inconvenientes:

Ventaja: El programador que lee el código puede ver claramente si en ese punto se procesó o no el error, y si se comprobó o no correctamente.

Inconveniente: El código se "complica" con el procesamiento de errores. Para el programador que lee la aplicación intentando comprender su funcionamiento, las líneas de control de errores le distraen de la lógica principal del sistema, dificultando la comprensión de la aplicación.

Esta es la forma elegida en los ejemplos sobre cadenas de caracteres y arrays, para impedir que se pueda acceder a posiciones que no existen. En todos esos casos hemos usado la función length o el método length() para impedir accesos más allá del último elemento del array o el String, respectivamente.

• Tratarlos adecuadamente una vez que ocurren. No se impide que se produzca el error, por que no se pueden tener en cuenta todas las circunstancias que lo pueden provocar, o sencillamente porque supone complicar excesivamente el código. Una vez que el error (o excepción) ha ocurrido, se "captura", y se pasa el control del flujo del programa a una zona concreta que llamamos manejador de la excepción, de forma que se corrige ese error, se avisa al usuario de lo que ha ocurrido, o sencillamente se termina el programa de forma ordenada impidiendo que aborte. El manejo de excepciones aparentemente rompe el flujo del programa, con un salto incondicional a otra zona de código. Pero si esto se hace como en Java, transfiriendo el control hacia delante, a una zona cercana y bien definida del código, no tiene porqué ser considerado como una violación de los principios de la programación estructurada.

Ventajas: Permite al programador quitar el código que maneja los errores del código principal, quedando un hilo de ejecución más limpio y claro, de forma que los programas así escritos son más fáciles de modificar y mantener. Además, es congruente con los criterios de modularidad. La tarea de tratar los errores es algo distinto al propósito principal del programa, y tiene sentido que se haga en un módulo distinto, que es el manejador de la excepción. Por otro lado, una vez

que se produce una excepción, no se puede ignorar, y debe haber un código que la maneje.

Inconveniente: Las líneas de código que gestionan los errores pueden estar físicamente bastante separadas de las líneas en las que se produjo el error, por lo que para saber cómo se maneja un error determinado puede obligarse al programador a revisar el código en su búsqueda. Pero este inconveniente es mínimo, ya que los manejadores de excepciones están bien identificados, y siempre detrás de los bloques try correspondientes que contienen el código potencialmente generador de errores.

La filosofía del manejo de excepciones en Java contempla la posibilidad de "capturar el error", y en lugar de abortar el programa, "manejar el error". ¿En qué consiste manejar el error?

- Si el error no es excesivamente grave, se podrá dar otra oportunidad al usuario para corregirlo, por ejemplo introduciendo un nuevo valor para el número que se solicita, o volviendo a llamar al método que provocó el error, pero con otros parámetros más adecuados.
- Si el error es un fallo realmente grave, un error irrecuperable, al menos tendremos la oportunidad de avisar convenientemente al usuario de lo que ha pasado, podremos salvar lo que sea aprovechable de nuestra aplicación, y terminar el programa ordenadamente, en vez de con un final brusco, como cuando el programa aborta.

Excepciones básicas

En general, por excepción entenderemos una situación anómala en la ejecución de un programa, que impide que se siga ejecutando el flujo normal del programa, sin que en el ámbito en que se produce esa situación de error tengamos información suficiente para corregir el error, por lo que debemos pasar el control del flujo del programa a otro ámbito en el que quizás sea posible manejar ese error disponiendo de más información. Así se propaga la excepción al método desde el que se invocó el código que generó la excepción, que si tampoco la sabe tratar la pasará a su vez al método que lo invocó, y así sucesivamente, hasta en el peor de los casos llegar al método main(), que le pasará la excepción a la máquina virtual Java, que escribirá un mensaje de error indicando el error ocurrido, escribirá la lista de invocaciones a métodos que han producido la excepción, y terminará la ejecución del programa. Esta es la situación menos deseable, pero en cualquier caso garantiza que cualquier excepción va a ser tratada, y no va a poder ignorarse sin más.

En Java existe la clase Throwable, y cualquier situación anómala en la ejecución de un programa genera directa o indirectamente un objeto de la clase Throwable.
La clase Throwable tiene dos subclases definidas en Java, cada una para un tipo de error o situación anómala en la ejecución:

- Error. Indica que se ha producido un fallo irrecuperable, para el que es imposible recuperar y continuar la ejecución del programa. La máquina Virtual Java presenta un mensaje en el dispositivo de salida, y concluye la ejecución del programa. Para este tipo de errores, no hay nada que el programador pueda hacer, por lo que no nos vamos a ocupar más de ellos.
- Exception. Indica una situación anormal, pero que puede corregirse y reconducirse para que el programa no tenga que terminar forzosamente. Java además nos proporciona cerca de 70 subclases directas de la clase Exception, ya predefinidas, cada una para un tipo de error concreto. Además, cada una de estas subclases suele tener a su vez bastantes otras subclases más, que nos definen cada vez errores más concretos y específicos.

Así por ejemplo, una de las subclases de la clase Exception es IOException, encargada de capturar errores de Entrada/Salida. Tiene a su vez 26 subclases, entre las que se encuentra EOFException, que se lanza cuando se alcanza inesperadamente el final de un fichero realizando una entrada de datos desde un fichero o flujo.

Además de la inmensa cantidad de subclases de Exception que ya nos da definidas el lenguaje Java, y que abarcan la práctica totalidad de situaciones problemáticas que puedas imaginar, el programador puede definir, lanzar y usar sus propias excepciones en Java, por lo que las posibilidades de capturar y tratar errores son casi infinitas.

En general, la sintaxis del manejo de excepciones en Java incluye el uso de las siguientes palabras reservadas:

try, catch, finally, throw, throws

Capturar una excepción (try - catch - finally)

¿Cómo identificar en el programa el código que puede generar excepciones?. De ello se encarga el bloque de código que comienza con la palabra reservada try, seguida de un bloque de sentencias entre llaves, que son el código que puede generar el error.

¿Cómo indicar el código al que se transfiere el control en el caso de que se produzca la excepción?. Es la misión del bloque iniciado por la palabra reservada *catch* seguida de un paréntesis que contiene la declaración del objeto de tipo Exception (o subclase de Exception) que se creará si se produce el error, y que recibe la cláusula catch como si de un parámetro se tratara. El paréntesis va seguido de un bloque de sentencias contenidas entre llaves, que constituyen el manejador de la excepción, indicando lo que debe hacerse en el caso de que se produzca el error. El bloque catch irá siempre después del bloque try.

Puede haber varios bloques catch para cada bloque try, cada uno encargado de indicar lo que debe hacerse en el caso de que se presente un determinado tipo distinto de excepción dentro de las sentencias del bloque try. Lo que no es posible es tener un bloque try sin ningún bloque catch, o viceversa. Sólo se permite un bloque try sin ningún bloque catch si se ha incluido un bloque finally. En definitiva, debe haber siempre un código que se ejecute cuando se lanza la excepción, que no puede quedar sin tratamiento.

Por último podemos colocar opcionalmente un bloque finally, que encerrará entre llaves un grupo de sentencias que se ejecutarán siempre, tanto si se produce una excepción como si no. Siempre que se incluya una cláusula finally, debe colocarse detrás del último bloque catch. Normalmente la cláusula finally no es muy utilizada, pero su principal utilidad es garantizar la liberación de recursos que el bloque try ha acaparado con uso exclusivo, y que podrían quedar definitivamente retenidos por él si no se ejecutan las sentencias que los liberan debido a que una excepción se produce antes de las sentencias encargadas de esa liberación del recurso, transfiriendo el control al bloque catch.

Por ejemplo de recurso que debería ser liberado una vez que no es necesario puede ser una conexión a una base de datos. La mayoría de las bases de datos tendrán establecido un número máximo de usuarios que pueden acceder simultáneamente a los datos. Si nuestra aplicación establece la conexión, y debido a un error, no la cierra, puede estar impidiendo que otro usuario pueda conectarse a la base de datos, aunque nuestra aplicación ya no esté usando esa conexión.

2.- Normas para el manejo de excepciones

A la hora de capturar excepciones, hay varias cuestiones que se deben tener en cuenta:

- Cuando hay una excepción que se lanza desde el bloque try, se transfiere el control al primer bloque catch cuyo parámetro coincida con el tipo de excepción que se ha lanzado.
- Si se coloca primero un bloque catch para una excepción más genérica, siempre se ejecutará ese bloque catch, y los que aparezcan detrás, correspondientes a subclases de excepciones de la anterior, no se alcanzarán ni se ejecutarán nunca.
- Cuando se ejecutan todas las sentencias del manejador de la excepción, se continúa con la sentencia siguiente, es decir, en el manejador de la excepción nunca se produce un salto hacia atrás, a la sentencia que produjo la excepción.
- Si la excepción que se lanza desde el bloque try no se captura por ningún catch adecuado, se relanza hacia el método anterior en la cadena de llamadas, hasta llegar a un lugar donde se capture y se trate adecuadamente.
- En el peor de los casos será la propia máquina virtual la que se encargue de tratar la excepción, escribiendo un mensaje de error y terminando la ejecución del programa. Es lo único que puede hacer la máquina virtual de forma genérica para cualquier excepción que le llegue sin haber sido capturada adecuadamente, ya que no tiene más información que le permita intentar una recuperación del error.
- Si se escribe el bloque opcional finally, irá después del último catch, y se ejecutará siempre, se produzca o no alguna excepción, y con independencia del tipo de excepción de la que se trate.

- Si en las sentencias del bloque catch se produjera una nueva excepción, sería esta última la única que se propagaría hacia los métodos llamantes.
- Aunque se puede transferir el control desde dentro del bloque try hacia otras zonas del código, usando break, continue o return, no es demasiado aconsejable hacerlo, y se debe tener presente que si se ha especificado una cláusula finally, primero se ejecutará la cláusula finally, y luego se transferirá el control a la zona del programa que corresponda.
- En el bloque catch se recibe como parámetro el objeto Exception del tipo de la excepción que se ha producido en el bloque try, y se le pueden enviar una serie de métodos para obtener información sobre la excepción que se ha producido, como por ejemplo getMessage() , que recoge el mensaje de error asociado al tipo de excepción que se ha producido.

En este ejemplo capturaremos la excepción *ArithmeticException*, que es una subclase de *RuntimeException*; aunque no es obligatorio tratar las excepciones que derivan de *RuntimeException*, en ciertos casos resulta conveniente.

La clase *Excepcion1* implementa un bucle sin fin (líneas 3 y 18) que recoge un valor entero suministrado por el usuario (línea 7) y lo utiliza como denominador en una división (línea 8). Todo ello dentro de un bloque *try* (líneas 5 y 10).

El bloque *catch* atiende únicamente a excepciones de tipo *ArithmeticException* (línea 12), imprimiendo el mensaje "Division por cero" en la consola (línea 13). La instrucción situada en la línea 16 nos muestra como se ejecuta el programa tras los bloques ligados *try-catch*.

```
1  public class Excepcion1 {
2     public static void main(String[] args) {
3      do {
4
5        try {
6           System.out.print("Valor: ");
7           int Valor = Teclado.Lee_int();
8           float Auxiliar = 8/Valor;
9           System.out.println(Auxiliar);
10       }
11
12       catch (ArithmeticException e) {
13          System.out.println("Division por cero");
14       }
15
16       System.out.println("Despues del catch");
17
18     } while (true);
19   }
20 }
```

Al ejecutar el programa e introducir diversos valores, obtenemos la siguiente salida:

3.- Creación de excepciones propias. Cláusulas throw y throws

A pesar de que Java nos proporciona una variedad suficientemente amplia de excepciones para todo tipo de situaciones posibles de error, también es interesante poder crear excepciones propias, de forma que los mensajes de error estén personalizados, o que podamos decidir las condiciones bajo las que deben lanzarse esas excepciones. Java una vez más nos ofrece esta posibilidad, aumentando la flexibilidad, y lo hace mediante las cláusulas throw y throws.

- Debemos definir una clase que declare la excepción que queremos crear como una subclase de alguna de las excepciones definidas por el lenguaje. Naturalmente puede ser una subclase de Exception.
- throw. Se usa para indicar en qué lugar del código de nuestro método se lanza esa excepción, y para lanzarla, de hecho.
- throws Se usa en la cabecera del método para avisar a los programadores usuarios de nuestro método de que lanza un determinado tipo de excepción, y que deberán preocuparse de capturarla y manejarla convenientemente en los programas que usen ese método.

Veamos una serie de ejemplos que pueden ayudar a entender mejor estos conceptos.

El primero consistirá en construir un bloque de código para controlar un error que se producirá si se intenta realizar una división por cero. Bastará con capturar la excepción y proporcionar el código que nos permita el tratamiento de ésta. La solución propuesta es la que sigue:

```
public class UnaExcepcionAritmetica {
      public static void main (String args[]){
          int valor; valor=100;
          try {
            for( int x=0; x < 100; x ++ )
                  valor /= x;
          }
      catch( ArithmeticException e ) {
          System.out.println( "Matemáticas locas!" );
          }
      catch( Exception e ) {
          System.out.println( "Se ha producido un error" );
          }
      }
}
```

El bloque de código que se encuentra dentro de esta cláusula try es el que corresponde a donde se prevé que se genere una excepción. Es como si dijésemos "intenta realizar estas sentencias y mira a ver si se produce una excepción". El bloque try tiene que ir seguido, al menos, por una cláusula catch o una cláusula finally.

La sintaxis general del bloque *try* consiste en la palabra clave **try** y una o más sentencias entre llaves.

```
try {
    // Sentencias Java
  }
```

Puede haber más de una sentencia que genere excepciones, en cuyo caso habría que proporcionar un bloque *try* para cada una de ellas.

En caso del ejemplo que nos ocupa, las sentencias que se encuentran dentro del try, intentarán realizar una serie de divisiones de la variable valor por el índice x. Lo que queremos que recoja la excepción será el caso de que intente realizar una división por cero. Por tanto ahora tendremos que colocar el controlador de la excepción. Los controladores de excepciones deben colocarse inmediatamente después del bloque try. Si se produce una excepción dentro del bloque try, esa excepción será manejada por el controlador que esté asociado con el bloque try.

Catch es el código que se ejecuta cuando se produce la excepción. Es como si dijésemos "controlo cualquier excepción que coincida con mi argumento". No hay código alguno entre un bloque try y un bloque catch, ni entre bloques catch.

La sintaxis general de la sentencia **catch** en Java es la siguiente:

```
catch( UnTipoTrhowable nombreVariable ) {
    // sentencias Java
    }
```

El argumento de la sentencia declara el tipo de excepción que el controlador, el bloque **catch**, va a manejar. La cláusula **catch** comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque.

En caso del ejemplo que nos ocupa la excepción que se captura es ArithmeticException, que se corresponde con una de las nueve predefinidas en la clase Exception. Las excepciones aritméticas son típicamente el resultado de división por 0. Lo que indicamos en nuestro caso en la captura de esta excepción es que si se produce nos muestre un mensaje tal y como este:

```
catch( ArithmeticException e ) {
System.out.println( "Las matemáticas se han vuelto locas!" );
}
```

Pero además, en el ejemplo, capturamos cualquier otra excepción que se pudiera producir distinta a la tratada anteriormente y mostramos otro mensaje. Esto lo realizamos con las siguientes líneas de código:

```
 catch( Exception e ) {
System.out.println ("Se ha producido un error");
}
```

Recuerda que la cláusula catch comprueba los argumentos en el mismo orden en que aparecen en el programa. Por tanto esta segunda excepción se ejecutará sólo si no se ha producido la primera.

Ejercicio: captura de excepciones

```
...
public class Circulo extends Figura {
...
public void leer (Scanner teclado) {
    do {
      try {
        boolean hayError=true;
        System.out.println("Introduzca el radio del Círculo\n");
        radio = teclado.nextDouble();
        if (radio<0) throw new ExcepcionRadioNegativo("Radio:"+radio);
        hayError=false;
      } catch (InputMismatchException e1) {
        System.out.println("No has tecleado un número. Vuelve a teclear");
      } catch (ExcepcionRadioNegativo e2) {
        System.out.println("No se admiten radios <0 ("+e2.getMessage()+").
Vuelve a teclear");                                                    -").
      }
    }while (hayError)
    ...
}

  }
```