# GESTIÓN DE EXCEPCIONES

Hay diferentes tipos de excepciones, al menos en cuanto su tratamiento. Hay excepciones activas (cuando un método la lanza, es obligado capturarla, suceda o no suceda su excepción) y pasivas (o declaradas, que no es obligada su captura, y por lo tanto queda a mi criterio el meter una estructura de captura o no). Pero qué es una excepción?. Hay que desterrar la idea de que una excepción es un error. En muchos casos sí es así, pero en otros muchos casos no es así.

Una excepción es un mecanismo que tiene JAVA para comunicarnos que un hecho ha sucedido. En el fondo, una excepción es prácticamente lo mismo que un evento, la diferencia fundamental es que el evento siempre es a partir de una acción de un usuario, mientras que en una excepción el usuario no interviene, sucede al ejecutar el código. Cualquier método puede arrojar una excepción, o no.

Su tratamiento es siempre el mismo: **estructura try-catch**

Try {

Obj\_metodo ();

} Catch (tipo de excepción a tratar e) {

Tratamiento de la excepción, órdenes a realizar, o nada

}

Cuando la excepción es un error de ejecución, una vez realizado lo que hay en el catch el programa podrá continuar o no, dependiendo del tipo de excepción. Hay algunas que provienen del fataerror, y ahí no podemos continuar el programa. En el resto de los casos el programa continúa. Con la mayor parte de las excepciones, el programa continuará. Hay muchos tipos de excepciones (+ de 100), tienen una estructura jerárquica de herencia, siendo la superclase Exception. Por lo tanto, cuando ponga en el catch tipo de excepción, tengo que tener cuidado con los tipos con los que voy a jugar. Yo puedo poner el tipo concreto de la excepción, o capturar cualquier tipo de excepción poniendo “Exception”. Un método puede lanzar más de un tipo de excepción, tantas como queramos. Cuando tenemos un método así, que lanza 2 tipos de excepciones, se le ponen tantos catch como excepciones lance. Un try puede tener tantos catch como necesite. O puedo tener un solo catch que capture “exception” que me serviría para todas.

Si por ejemplo:

Catch (Exception)

Catch (IoException)

El segundo catch sobra, ya que el primero lo coge. Tendríamos que cambiar el orden para capturar específicamente la IoException. Funciona como un switch. Dentro del try, se puede colocar más de un método, lancen o no excepciones. No sólo se pueden poner dentro métodos que lancen excepciones. Ahora bien, si yo en el try tengo 4 métodos, y el que lanza la excepción es el segundo, de ahí hacia abajo no se ejecuta ninguna orden más, no es de ida y vuelta, va al catch y sigue ejecutando código por debajo, pero no vuelve. El hecho de que un método PUEDA producir una excepción no quiere decir que esa excepción se lance siempre, el compilador sólo nos obliga a su captura “por si acaso”. El catch tiene una cláusula adicional que nos puede ayudar, aunque no se suele usar mucho. Es la cláusula finalize. Cuando un try catch termina con finalize, el código que ponga ahí se ejecutará siempre tanto si se lanza la excepción como si no. Es una solución para poner código que se ejecute siempre.

Se pueden anidar try-catch. Puede haber el número de try-catch que yo necesite, no hay límite:

Metodo () {

Try

MétodoQueLanzaLaExcepcion

Catch

Try

MétodoQueLanzaLaExcepcion

Catch

Try

MétodoQueLanzaLaExcepcion

Catch

Try

MétodoQueLanzaLaExcepcion

Catch

}

Prácticamente todos los métodos del API de base de datos (JDBC) lanzan excepciones activas, así que éste escenario es normal. Cuando sucede esto, se puede **delegar** el tratamiento a un nivel superior. Esto quiere decir que en el escenario anterior, se le pone al final del nombre del método la palabra reservada “throws tipo de excepción”. Ese método a partir de ese momento, está lanzando excepciones activas. Cuando yo hago eso, puedo eliminar todos los try-catch de ese método, siempre y cuando todas las excepciones lanzadas sean del mismo tipo, y el tratamiento sea el mismo. Desde ese momento, cuando se llame a ese método, hay que ponerle dentro del try-catch:

Metodo () Throws SQLException {

MétodoQueLanzaLaExcepcion

MétodoQueLanzaLaExcepcion

MétodoQueLanzaLaExcepcion

MétodoQueLanzaLaExcepcion

}

*Y lo tratamos en otra parte, que puede ser incluso otra clase:*

Try {

Obj\_metodo();

} Catch (SQLException e) {

}

Se trata la excepción cuando se le llame al método, mediante un try catch. A su vez, podría ir rebotando hacia arriba las excepciones producidas para que fueran tratadas por una clase superior, aunque no se suele hacer, la primera vez que se rebota se trata.

En el tratamiento de la excepción, normalmente no va más allá de registrar y/o informar, registrar que ha sucedido la excepción (en un log) y/o informar al usuario de que puede o no puede hacer determinada opción.

Para tratar las excepciones, al final muchas veces terminarás dándoles el mismo tratamiento a todos, entonces lo mejor es crearse un objeto que los trate siempre para centralizar el tratamiento de excepciones y no repetir código.

Try {

Obj\_metodo();

} Catch (SQLException e) {

miExcepcion xx= new miExcepcion(e);

}

Lo mejor es centralizar el tratamiento, así cualquier ampliación supone que tengo que cambiar pocas clases y no tengo que andar recorriendo montones de try-catch.

CREACIÓN DE EXCEPCIONES

Lo más avanzado que se puede hacer con una excepción es crearte tus propias excepciones, es un mecanismo muy sencillo:

Me declaro mi clase

Public class xxxx extends Exception

Al heredar de exception, ya es una excepción, ya puede ser tratada, capturada,…

También puedo especializarla metiéndole los métodos para tratar la excepción concreta que quiero crear.

En cualquier método de cualquier clase que yo quiero que lance miexcepcion (para mis propias clases que yo desarrolle):

Metodo() throws miexcepcion {

If (\_\_\_\_\_\_) {

Throw new miexcepcion();

}

}

En algún momento del cuerpo del método, pondré un if, y pondré las condiciones para que lance dicha excepción, con la instrucción throw new miexcepcion();

Esto hace que cuando se llame al método en el que yo le he puesto miexcepcion, haya que tratarla obligatoriamente.

Esto lo puedo hacer por ejemplo en una autentificación de usuario, que compruebe el usuario y si el usuario es inválido me lance una excepción.

Try {

Obj.metodo();

} Catch (miexcepcion e) {

-----

-----

}

MÁS SOBRE FORMATOS

JTextfield no nos deja manipular los formatos, por lo tanto cuando tenga un campo en el que sí quiera formato, tendré que usar otro componente distinto, el JFormatedTextField.

Es el componente caja de texto, con todas las características del JTextField, y que se diferencia de éste en que puede trabajar con formatos. Pero, los formatos, los controlan otros objetos distintos, esos objetos se los tengo que pasar al constructor de JFormatedTextField.

Si estamos hablando de formatos de datos numéricos, la clase es NumberFormat. Me sirve para crearme objetos que controlarán formatos numéricos, igual que SimpleDateFormat creaba objetos de fechas. Ésta clase no tiene constructor, se crea a partir de métodos de instancia:

Yyy= NumberFormat.getInstance();

“””””””””””””””””””””.getCurrencyInstance();

“””””””””””””””””””””.getPercentInstance();

“””””””””””””””””””””.getIntegerInstance();

getInstance: me devuelve un formato numérico con las características del locale. (comas para los decimales, puntos para los millares, etc)

getCurrencyInstance: Nos hace lo mismo pero para un formato monetario, cuando la magnitud a representar es dinero, la moneda del país; en nuestro caso, dos decimales y símbolo monetario del euro.

getPercentInstance: Formato porcentual: 3 enteros, dos decimales y el símbolo de porcentaje %.

getIntegerInstance: un valor entero sin decimales.

Decidiendo del valor numérico que quiera usar, me crearé uno u otro, y se lo pasaremos al JFormatedTextfield.

Aparte de los métodos de instancia, puedo modificar el NumberFormat, mediante otros métodos:

.setMaximumIntegerDigit: Cantidad máxima de dígitos en la parte entera

.setMinimunIntegerDigit: cantidad mínima de dígitos en la parte entera.

.setMaximumFractionDigit: Cantidad máxima de dígitos en la parte decimal

.setMinimumFractionDigit: Cantidad mínima de dígitos en la parte decimal.

Y por supuesto, para convertir una primitiva a cadena de caracteres con el formato que me interese, el método

.Format (double, int, long, etc) y me devuelve ese valor como un String.

Una vez creado el NumberFormat, se lo paso al JFormatedTextField (obj). Si el formateador acepta ese formato nuevo, se lo come, si no vuelve al formato interior. Para que el JFormatedTextField nos funcione, además aparte de meterle el NumberFormat hay que pasarle un valor inical, si no se lo pasamos el NumberFormat no funciona; mediante el método:

setValue(objeto que corresponda al formato, si el formato es entero, le pasaré un integer, si he puesto un formato con decimales le pasaré un double o float). Si le pasas una primitiva, el compilador no dará error pero el JFormatedTextFormat no funcionará. Se le pueden pasar objetos, con lo cual le puedo pasar un objeto SimpleDateFormat para que sólo acepte fechas.

Cuando quier trabajar con un formato que no se ajusta a ninguno de los establecidos (DNIs, valor numérico +letra, no es una letra…) podré usar el método maskFormatter

Importante sobre las máscaras: Cuando defino una máscara par aun valor, el usuario tiene que rellenar completamente la máscara, si no, no vale. En el ejemplo de los DNIs, habrá que poner un 0 por delante para que sean los 8 dígitos. Para instanciarlo, llamamos al constructor:

maskFormatter xxx=new MaskFormatter (“”); como argumento le podemos pasar algunos parámetros:

#-->Cada almohadilla, un dígito numérico.

U🡪 Cada U, un carácter letra en mayúscula.

L🡪 Cada L, un valor carácter en minúscula. (si el usuario mete mays, cambia automáticamente el caso igual en minúscula)

A🡪 O letra o número.

\*🡪 Cualquier cosa.

H🡪Para hexadecimal.

Para el NIF: ########-U

Aparte de que el constructor lanza una excepción activa, el objeto maskFormatted se le pasa al constructor…pero hay que activarla antes de poder usarlas. En el objeto MaskFormatted hay que llamar al método:

xxxsetAllowInvalid(false), y ponerlo a false, viene en true predefinido.

Cuando definimos una máscara, el usuario se encuentra el campo, y le aparece el guioncito. Yo puedo hacer que aparezca un carácter debajo para que le sirva de guía al usuario.

setPlaceHolderCharacter(‘\_’); Le sale en el siguiente carácter que tiene que meter, es como un cursor.

Si queremos resolver el hecho de que el usuario no me llene la máscara por completo, tiene un método para decirle que me la rellene:

setPlaceHolder (“”). Cuando tengo activado este método, y el usuario no rellena la máscara, completa la máscara con el string que le indico. Los pone por la derecha.

Para seleccionar el escuchador del buttonGroup no vale ActionListener ya que lo que quiero es que al seleccionar uno de los botones no me deje escribir en la otra caja, para eso usaré ItemListener que sí me permite hacer eso.

En el escuchador de los botones, preguntaremos por la propiedad name, ya que al internacionalizar es un parámetro que no cambia. Si preguntamos por text, que sí cambia, habría que cambiarlo.