Introducción al manejo de Excepciones en Java

¿Te pasó alguna vez que uno de tus programas "se rompa"?

Seguramente sí. Aunque sea odioso, te habrá pasado que, corriéndolo desde Eclipse, tu programa emitió —en vez de lo esperado y justo antes de abortar— un texto en rojo y algunas palabras medio raras. Te puede haber pasado al tipear un valor no numérico mientras se esperaba un valor numérico, por ejemplo. O tratando de acceder al método de un objeto no instanciado (donde la variable en vez del objeto esperado tenía un null).

Sin embargo, si mirás ese mensaje de error con ojos de programador vas a encontrar, en esas pocas líneas, información realmente valiosa.

*Las **Excepciones** (ése es el nombre técnico para estos mensajes) proveen un mecanismo de tratamiento ordenado y uniforme para cualquier situación anómala que se produzca en tiempo de ejecución de un programa y que impida el flujo natural de ejecución de sus instrucciones.*

Las Excepciones son aliadas importantes del programador, pues ayudan no sólo a que el código sea más claro, sino que gracias a la forma en que funcionan nos permiten escribir programas más robustos, algo que no es tan fácil de conseguir con la metodología tradicional o imperativa. *El conocimiento de cómo funcionan, cuáles son sus características y su posterior buen uso son parte importante de lo que un programador debe conocer.*

Esto no es más que una introducción al tema, y veremos en profundidad esto mismo más adelante. Pero es bueno que conozcamos ahora algunos casos de tratamiento de Excepciones para mejorar la calidad de nuestros programas.

A partir de esta guía vas a aprender a:

1. *Evitar errores de ejecución en la carga de datos por teclado cuando el dato es numérico.*
2. *Generar y capturar Excepciones en la creación de objetos cuando el constructor reciba argumentos inválidos o no esperados.*

Antes de empezar

Básicamente, las excepciones...

- … son *eventos* que pueden ocurrir durante la ejecución de un programa.
- … causan, al producirse, una alteración en el circuito normal de ejecución del código, ya que introducen un *salto* desde donde se produce el error hacia el *punto de tratamiento más cercano*, dentro del mismo bloque de código ó en cualquier punto *hacia afuera* (hacia el método *main()*, la capa más externa y base en la *call stack*, o pila de llamadas).
- … son *mensajes*, ya que avisan al proceso invocante que algo ha pasado, y que ese algo no fue común. A diferencia de cualquier otra metodología que usemos para advertir al programa de la existencia de un error, son las únicas que no pueden ser ignoradas. Podemos ignorar un *false* devuelto por un método, o un código de error del tipo que fuere, pero no podemos ignorar algo que si no es tratado hará que el programa deje de funcionar.
- ... son *objetos* que contienen la información necesaria para saber qué pasó y en dónde, y todo el camino desde allí hasta el lugar de tratamiento del error (o el método main()) desandando la pila de llamadas, mostrando una "*foto*" de la memoria y de los pasos que dio el programa en pos de determinar y, si es posible, resolver el problema.
- … al ser instancias de clases que pertenecen a un árbol jerárquico, permiten que se pueda determinar su grado de importancia, la familia a la que pertenecen y el orden de tratamiento que hará falta para procesarlas en forma adecuada.

Algunos ejemplos de errores y situaciones excepcionales son:

- Dividir un entero por 0.
- Ingresar por teclado un valor incompatible con la variable receptora (por ejemplo, querer cargar un texto en una variable numérica).
- Intentar el acceso a un atributo de una variable de tipo objeto cuyo valor es *null*.
- Intentar el acceso a una posición de un arraylist con un índice que sale del rango aceptable para el tamaño de la estructura.

Ahora sí, comencemos a ver los casos de Excepciones que vamos a manejar en esta materia.

Caso 1: Tratamiento de Excepciones en la carga de datos

La intención de este caso es evitar que el programa aborte por la carga errónea de datos desde el teclado. Este caso es uno de esos *inevitables*, ya que no hay forma (simple) de que desde el código podamos controlar que el usuario de nuestro programa tipee un dato del tipo que esperamos. **Siempre** los puntos del programa que tienen *contacto con el mundo exterior* son factor de riesgo.

Existen dos partes bien diferenciadas respecto al manejo de excepciones: la generación de la excepción y su posterior tratamiento.

La generación de una excepción puede ser implícita o explícita. Una excepción es implícita cuando se genera automáticamente por un fallo natural en la ejecución del programa (con lo cual será de alguna de las clases de excepción ya contempladas por el lenguaje).

Este es nuestro caso. Veamos la siguiente instrucción:

```
int numero = Integer.parseInt(scanner.nextLine());
```

Esta línea intenta la carga desde teclado de un número entero: primero lee una línea desde el teclado con *scanner.nextLine()*. Eso genera un String que para ser usado como número necesita su conversión a entero, y ahí entra en juego el método *Integer.parseInt()*. Pero si el valor ingresado no puede convertirse en un número entero dará algo como lo que sigue:

```
Exception in thread "main" java.lang.NumberFormatException: For input
string: "esto que tipee no es un numero"
      at java.lang.NumberFormatException.forInputString(Unknown Source)
      at java.lang.Integer.parseInt(Unknown Source)
      at java.lang.Integer.parseInt(Unknown Source)
      at caso_a.TestExcepcionEnNros.main(TestExcepcionEnNros.java:11)
```

Afortunadamente las excepciones pueden ser capturadas y de ser posible tratadas. Esto se hace utilizando un bloque ***try-catch*** ó, en su versión completa, ***try-catch-finally***.

Trabajemos ahora sobre un ejemplo concreto: si *rodeamos* la línea de código anterior con un bloque ***try-catch-finally*** que sepa tratar la excepción que fue *lanzada* cuando se intentó transformar a entero un valor no numérico (NumberFormatException) podremos atrapar el error si éste se produce y controlar qué hacer en ese caso: así evitamos que el programa aborte:

```
int numero = -1;
try {
    System.out.println("El numero vale " + numero);
    System.out.print("ingresá un número: ");
    numero = Integer.parseInt(scanner.nextLine());
} catch (NumberFormatException nfe) {
    System.out.println("Hubo un error en la carga del numero: "
                       + nfe.getMessage());
} finally {
    System.out.println("El numero ahora vale " + numero);
}
```

Si traducimos esta porción de código Java a seudocódigo podríamos obtener algo así:

```
Creo la variable numero y la inicializo en -1
Intento
    asignar a numero lo que cargo por teclado
Atrapo, si se produce una ExcepcionDeFormateoDeNumero
    muestro "Hubo un error..." y (mensaje de la excepción)
Finalmente
    muestro el valor actual de numero, se haya modificado o no
```

Desglosándolo, el primer bloque incluye todo lo que ***intentamos*** hacer (las instrucciones o algoritmo que realiza lo que necesitamos). SI algo falla dentro de ese bloque, el programa saltará automáticamente al ***atrapo***, donde deben figurar una o más excepciones, y el tratamiento de éstas. Luego (y opcionalmente pues el ***finally*** no es obligatorio) lo que sigue a lo que intentamos hacer, haya existido una falla o no. Esta última parte puede parecer superflua (después de todo, si no estuviese podría hacer lo mismo en las líneas posteriores al ***catch***) pero explicita qué sucederá sea cual fuere lo que haya pasado, tanto si hubo un error como si no lo hubo.

Al ejecutar el código anterior, en caso de hacerlo correctamente obtendremos:

```
El numero vale -1
ingresá un número: 5
El numero ahora vale 5
```

Pero si en vez de tipear el numero tipeamos otra cosa, el resultado será…

```
El numero vale -1
ingresá un número: otra cosa
Hubo un error en la carga del numero: For input string: "otra cosa"
El numero ahora vale -1
```

A diferencia de no usar el ***try-catch***, ahora el programa trató el error y no abortó. No obstante, no obtuvimos cambios significativos de funcionalidad más allá de evitar que el programa "se rompa". Pero **sí hay algo importante**: la variable número **mantuvo su valor**

original. Esto quiere decir que cuando se produce una excepción, **la instrucción que la causa *no termina de ejecutarse***.

Veamos cómo hacer un método que sepa cargar un valor numérico sin romperse al cargar un valor no numérico y que nos permita reintentar la carga hasta obtener un valor válido:

```
import java.util.Scanner;

public class TestExcepcionEnNros {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int numero = pedirNumeroEntero(scanner);
        System.out.println("El numero cargado es " + numero);
    }

    private static int pedirNumeroEntero(Scanner scanner) {
        int valorNumerico = 0;
        boolean hayError;
        do {
            try {
                System.out.print("Ingresa un numero entero: ");
                valorNumerico = Integer.parseInt(scanner.nextLine());
                hayError = false;
            } catch (NumberFormatException nfe) {
                System.out.println("Hubo un error en la carga del numero.");
                hayError = true;
            }
        } while (hayError);
        return valorNumerico;
    }

}
```

El programa anterior muestra tanto el desarrollo del método *pedirNumeroEntero()* como su invocación desde el método *main()*. A diferencia del caso anterior este método no tiene un ***finally*** pues es innecesario: no hay nada que podamos poner en él. Al usarla, pedirá el ingreso de un número hasta que no haya un error de carga (es decir, hasta que se cargue un valor realmente numérico):

```
Ingresa un numero entero: h
Hubo un error en la carga del numero.
Ingresa un numero entero: i
Hubo un error en la carga del numero.
Ingresa un numero entero: j
Hubo un error en la carga del numero.
Ingresa un numero entero: 4
El numero cargado es 4
```

Así aseguramos que el programa procese correctamente la carga de un valor entero, evitando que el programa aborte cuando el valor no cumpla con los requisitos deseados.

Caso 2: Validar la creación de Objetos con Excepciones

El otro caso que vamos a tratar ahora es el lanzamiento de excepciones como parte de la validación en la creación de un objeto.

Supongamos que queremos crear una instancia de la clase ***Persona*** que debe tener cargados, indefectiblemente, su nombre, su año de nacimiento y si ha fallecido, su año de fallecimiento. Y, por supuesto, es un requisito de la clase que los valores numéricos estén cargados correctamente, dentro de rangos razonables. Veamos una clase ***Persona*** bien simple, escrita tal como lo venimos haciendo hasta ahora:

```
public class Persona {
    private String nombreCompleto;
    private int anioNacimiento;
    private int anioFallecimiento;

    /**
     * @param nombreCompleto
     *            Nombre completo de la persona.
     * @param anioNacimiento
     *            Anio de nacimiento de la persona. Debe ser menor o
     *            igual al anio actual.
     * @param anioFallecimiento
     *            Anio de fallecimiento de la persona. Debe ser mayor
     *            o igual al anio de nacimiento y menor o igual
     *            al anio actual.
     */
    public Persona(String nombreCompleto, int anioNacimiento,
                   int anioFallecimiento) {
        this.nombreCompleto = nombreCompleto;
        this.anioNacimiento = anioNacimiento;
        this.anioFallecimiento = anioFallecimiento;
    }
    /**
     * @return el nombre completo de la persona.
     */
    public String getNombreCompleto() {
        return nombreCompleto;
    }
    /**
     * @return el anio de nacimiento de la persona.
     */
    public int getAnioNacimiento() {
        return anioNacimiento;
    }
    /**
     * @return el anio de fallecimiento de la persona.
     */
```

```
    public int getAnioFallecimiento() {
        return anioFallecimiento;
    }
    @Override
    public String toString() {
        return "Persona [nombreCompleto=" + nombreCompleto
              + ", anioNacimiento=" + anioNacimiento
              + ", anioFallecimiento=" + anioFallecimiento + "]";
    }

}
```

El problema que tenemos con esta clase es que las validaciones de sus datos no son más que una *declaración de intenciones*, ya que no hay ninguna validación real que se esté haciendo.

Pues bien, entonces, nos proponemos validar en el constructor lo que haya que validar. Pero, hagamos lo que hagamos, si los datos que nos pasan desde el exterior son inválidos o simplemente incongruentes (por ejemplo, la fecha de fallecimiento es anterior a la de nacimiento, o alguna de las fechas está definida en el futuro), el objeto será igualmente creado, y no hay nada que podamos hacer al respecto. Podemos mostrar un cartel que avise del error, pero ya sabemos que eso no obliga a nada a quien creó la instancia de Persona.

¿Qué podemos hacer, entonces?

La solución está en el lanzamiento de una excepción.

Así como hay excepciones que se lanzan automáticamente, nosotros también podemos crear y lanzar excepciones por cuenta propia.

Observá estos nuevos constructores para la clase ***Persona***:

```
/**
* @param nombreCompleto
* @param anioNacimiento
*/
public Persona(String nombreCompleto, int anioNacimiento) {
    this(nombreCompleto, anioNacimiento, -9999);
}
/**
* @param nombreCompleto
* @param anioNacimiento
* @param anioFallecimiento
*/
public Persona(String nombreCompleto, int anioNacimiento,
                int anioFallecimiento) {
    // validamos todo previo a la asignación
    // se recibió un nombre completo para ponerle?
    if (nombreCompleto == null || nombreCompleto.isEmpty()) {
        throw new IllegalArgumentException(
            "El nombre no puede estar vacio ni ser null");
    }
```

```
    // ahora validamos que los anios sean validos y coherentes.
    int anioActual = Calendar.getInstance().get(Calendar.YEAR);
    if (anioNacimiento > anioActual) {
        throw new IllegalArgumentException(
            "El anio de nacimiento no puede estar en el futuro");
    } else if (anioFallecimiento != -9999) { // -9999 indica que vive
        if (anioFallecimiento > anioActual) {
            throw new IllegalArgumentException(
                "El anio de fallecimiento no puede estar " +
                "en el futuro");
        } else if (anioFallecimiento < anioNacimiento) {
            throw new IllegalArgumentException(
                "No puede fallecer antes de nacer.");
        }
    }
    this.nombreCompleto = nombreCompleto;
    this.anioNacimiento = anioNacimiento;
    this.anioFallecimiento = anioFallecimiento;
}
```

Esto ya está mejor que lo anterior, pues primero se hacen todas las validaciones, y en caso de encontrar algún valor inválido, se crea y lanza una excepción (notá que es de un nuevo tipo, distinta a las que vimos hasta ahora) que es complementada con un mensaje personalizado que indica lo más claramente posible cuál es el problema.

No obstante, el código del segundo constructor ha quedado *sucio*, lleno de validaciones que parecen estar de más en un constructor (pues validar no parece ser una responsabilidad del constructor), y si bien no necesitás cargar -9999 como año de fallecimiento, cualquiera podría ingresar ese dato (sí, es ilógico, pero puede pasar) y eso estaría mal.

¿Cómo podemos hacer para limpiar el constructor y evitar estos problemas?

La solución está en usar *setters*.

La siguiente nueva versión de la clase ***Persona*** valida y carga cada dato donde corresponde: en el *setter* perteneciente a cada atributo. Así, cada constructor invoca a los setters reenviándole a cada uno el valor que ha recibido como parámetro:

```
import java.util.Calendar;

public class Persona {
    private static final int VALOR_AUN_VIVE = -9999;
    private String nombreCompleto;
    private int anioNacimiento;
    private int anioFallecimiento;
    private int anioActual;

    /**
     * Constructor para personas que aún viven.
     *
     * @param nombreCompleto
```

```
     * @param anioNacimiento
     */
    public Persona(String nombreCompleto, int anioNacimiento) {
        this.nombreCompleto, anioNacimiento, VALOR_AUN_VIVE);
    }

    /**
     * Constructor para personas que ya han fallecido.
     *
     * @param nombreCompleto
     * @param anioNacimiento
     * @param anioFallecimiento
     */
    public Persona(String nombreCompleto, int anioNacimiento,
                   int anioFallecimiento) {
        setAnioActual();
        this.setNombreCompleto(nombreCompleto);
        this.setAnioNacimiento(anioNacimiento);
        this.setAnioFallecimiento(anioFallecimiento);
    }

    /**
     * Guarda el anio actual para hacer las validaciones correspondientes a
     * las fechas de nacimiento y fallecimiento.
     */
    private void setAnioActual() {
        anioActual = Calendar.getInstance().get(Calendar.YEAR);
    }

    /**
     * @param nombreCompleto
     *            El nombre completo a asignar.
     */
    private void setNombreCompleto(String nombreCompleto) {
        // se recibió un nombre completo para ponerle?
        if (nombreCompleto == null || nombreCompleto.isEmpty()) {
            throw new IllegalArgumentException(
                "El nombre no puede estar vacio ni ser null");
        }
        this.nombreCompleto = nombreCompleto;
    }

    /**
     * @param anioNacimiento
     *            El anio de nacimiento a asignar. No puede ser mayor al anio
     *            anio actual.
     */
    private void setAnioNacimiento(int anioNacimiento) {
        if (anioNacimiento > anioActual) {
            throw new IllegalArgumentException(
```

```
                "El anio de nacimiento no puede estar en el futuro");
        }
        this.anioNacimiento = anioNacimiento;
    }

    /**
     * @param anioFallecimiento
     *            El anio de fallecimiento a asignar. No puede ser menor al
     *            anio de nacimiento ni mayor al anio actual.
     */
    public void setAnioFallecimiento(int anioFallecimiento) {
        if (anioFallecimiento != VALOR_AUN_VIVE) {
            if (anioFallecimiento > anioActual) {
                throw new IllegalArgumentException("El anio de " +
                        "fallecimiento no puede estar en el futuro");
            } else if (anioFallecimiento < anioNacimiento) {
                throw new IllegalArgumentException("No puede " +
                        "fallecer antes de nacer.");
            }
        }
        this.anioFallecimiento = anioFallecimiento;
    }

    /**
     * @return El nombre completo de la persona.
     */
    public String getNombreCompleto() {
        return nombreCompleto;
    }

    /**
     * @return El anio de nacimiento de la persona
     */
    public int getAnioNacimiento() {
        return anioNacimiento;
    }

    /**
     * @return El anio de fallecimiento de la persona.
     * Si vale -9999 significa que la persona aun vive.
     */
    public int getAnioFallecimiento() {
        return anioFallecimiento;
    }

    @Override
    public String toString() {
        return "Persona [nombreCompleto=" + nombreCompleto +
                ", anioNacimiento=" + anioNacimiento +
                ", anioFallecimiento=" + anioFallecimiento + "]";
```

```
    }

}
```

Así, cada método hace lo que corresponde pero, ¿cómo funciona?

Cuando una excepción se lanza, sea esta generada por el mismo programa o por nosotros, interrumpe la ejecución del método actual y empieza a desandar la pila de llamadas (*callStack*) camino al método *main()*, a la espera de que alguien la atrape. Si nadie la ha atrapado antes de que se acabe la pila de llamadas, el programa abortará como siempre que se produce una excepción. Te invito a que pruebes esta clase desde un main como el que sigue. Creá un lote de prueba de *Caja Blanca* y así probar cambiando los datos en la creación con todas las posibilidades que se te ocurran, observando qué pasa en cada caso:

```
public static void main(String[] args) {
    Persona unaPersona = null;
    try {
        unaPersona = new Persona("Juan", 1984, 2018);
    } catch (RuntimeException re) {
        System.out.println(re.getMessage());
    } finally {
        System.out.println("Los datos de la persona son " + unaPersona);
    }
}
```

Para cerrar: *las Excepciones son una excelente herramienta dentro de las disponibles en los lenguajes de programación modernos. Como todas las demás herramientas de un lenguaje, deben ser usadas con responsabilidad y mesura, ya que no son la solución infalible a todos los problemas y están al mismo nivel de cualquier otra herramienta del lenguaje. Un buen entendimiento y uso de las excepciones, teniéndolas en cuenta en el diseño de nuestros programas, harán que nuestro código sea más robusto, confiable y adaptable a nuevas circunstancias.*