	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Dra. Rocío Alejandra Aldeco Pérez

Asignatura: Programación orientada a objetos -1323

Grupo: 6

No de Práctica(s): 10 - 11

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de Equipo de cómputo empleado:

No. de Lista o Brigada:

Semestre: 2023-1

Fecha de entrega: 16 de noviembre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 10-11.

Excepciones y errores / Manejo de archivos.

OBJETIVOS

- Identificar bloques de código propensos a generar errores y aplicar técnicas adecuadas para el manejo de situaciones excepcionales en tiempo de ejecución.
- Implementar el intercambio de datos (lectura y escritura) entre fuentes externas (archivos y/o entrada y salida estándar) y un programa (en un lenguaje orientado a objetos).

ACTIVIDADES

- Capturar excepciones de un bloque de código seleccionado.
- Capturar errores de un bloque de código seleccionado.
- Crear archivos de texto plano.
- Leer archivos de texto plano.
- Escribir en archivos de texto plano.

INTRODUCCIÓN

La ley de Pareto, también conocida como la regla 80/20, establece que el 80% de las consecuencias proviene del 20% de las causas.

Una máxima en el desarrollo de software dicta que el 80% del esfuerzo (en tiempo y recursos) produce el 20% del código. Así mismo, en términos de calidad, el 80 % de las **fallas** de una aplicación son producidas por el 20% del código. Por lo tanto, detectar y manejar errores es la parte más importante de una aplicación robusta.

Una aplicación puede tener diversos tipos de **errores**, los cuales se pueden clasificar en tres grandes grupos:

- **Errores sintácticos:** Son todos aquellos errores que se generan por **infringir las normas de escritura de un lenguaje**: coma, punto y coma, dos puntos, palabras reservadas mal escritas, etc. Normalmente son detectados por el compilador o por el intérprete (según el lenguaje de programación utilizado) al procesar el código fuente.

- **Errores semánticos (o lógicos):** Son errores más sutiles, se producen cuando la sintaxis del código es correcta, pero la **semántica o significado** no es el que se pretendía. Los compiladores e intérpretes solo se ocupan de la estructura del código que se escribe y no de su significado, por lo tanto, un error semántico puede hacer que el programa termine de forma anormal, con o sin un mensaje de error.

  No todos los errores semánticos se manifiestan de una forma obvia. Un programa puede continuar en ejecución después de haberse producido errores semánticos, pero su estado interno puede ser distinto del esperado. Quizás las variables contengan los datos incorrectos, o bien es posible que el programa siga un camino distinto del pretendido.

Eventualmente, la consecuencia será un resultado incorrecto. Estos errores se denominan lógicos, ya que, aunque el programa no se bloquea, la lógica que representan contiene un error.

- **Errores de ejecución:** Son errores que se presentan cuando la aplicación se está ejecutando. Su origen puede ser diverso, se pueden producir por un **uso incorrecto del programa** por parte del usuario (si ingresa una cadena cuando se espera un número), o se pueden presentar debido a **errores de programación** (acceder a localidades no reservadas o hacer divisiones entre cero), o debido a algún **recurso externo** al programa (al acceder a un archivo o al conectarse a un servidor o cuando se acaba el espacio en la pila (stack) de la memoria).

Un **error en tiempo de ejecución** provoca que la aplicación termine abruptamente. Los lenguajes orientados a objetos proveen mecanismos para **manejar errores de ejecución**.

Los programas necesitan comunicarse con su entorno, tanto para obtener datos e información que deben procesar, como para devolver los resultados obtenidos. El manejo de archivos se realiza a través de **streams** o **flujos de datos** desde una fuente hacia un repositorio. La fuente inicia el flujo de datos, por lo tanto, se conoce como flujo de datos de entrada. El repositorio termina el flujo de datos, por lo tanto, se conoce como flujo de datos de salida. Es decir, tanto la fuente como el repositorio son nodos de flujos de datos.

INSTRUCCIONES

Realiza las siguientes actividades después de leer y revisar en clase la *Práctica de Estudio 10: Excepciones y errores* y *11: Manejo de archivos.*

1. Crea una clase llamada *Conversiones* que contenga los métodos ***int hexToDec (String)*** y ***String decToHex (int)*** que convertirán un número hexadecimal a decimal y un decimal a hexadecimal, respectivamente.
2. Deberás crear las siguientes excepciones para indicar que se ha recibido un dato con el formato equivocado:
   a) Para el método ***hexToDec***:
   - ***InvalidHexException***: cuando se recibe un número que no tiene formato hexadecimal
   - ***NegativeValueEnteredException***: cuando se recibe un número negativo

   b) Para el método ***decToHex***:
   - Revisa las excepciones de la clase ***Scanner*** y úsalas para indicar si no se está leyendo un número decimal.
   - ***NegativeValueEnteredException***: cuando se recibe un número negativo
3. Deberás leer los números de un archivo e imprimir los resultados en Alphagrader.
4. Deberás elegir correctamente los métodos a utilizar en cada caso para que los resultados sean los correctos.
5. Recuerda agregar las excepciones derivadas del uso de archivos.
6. Cuando estés seguro de que tu programa es correcto, súbelo a Alphagrader.
7. Explica el proceso que seguiste para manejar excepciones y archivos. Finalmente concluye.

DESARROLLO

Descripción de actividades realizadas

Para trabajar en esta práctica se creó el proyecto *Practica10-11_EmilianoMendoza*. Dentro del proyecto se creó el paquete *mx.unam.fi.poo.excepciones*, que es donde se crearan las clases de excepciones y la clase *Conversiones*.

1. Analizando las excepciones de los métodos:

Para el método ***hexToDec*** se tienen las posibles excepciones:

- ***InvalidHexException***
- ***NegativeValueEnteredException***

Para el método ***decToHex*** se tienen las siguientes excepciones:

- ***NegativeValueEnteredException***
- Excepciones de la clase ***Scanner***

Sin embargo, dado el parámetro *dec* en la declaración del método, se pretende que el método reciba un valor de tipo *int* al invocar a este método. Pasarle cualquier otro tipo de dato como parámetro a este método generaría un error de sintaxis. Por lo tanto, se consideró "filtrar" otros tipos de datos (que no sean números) dentro del método ***hexToDec***, que recibirá un *String* (más ideal para hacer comparaciones de caracteres). Por lo tanto:
Para el método ***hexToDec*** se tienen las posibles excepciones:

- ***InvalidHexException***
- ***NegativeValueEnteredException***
- ***NotANumberException***
- ***NotADecimalNumberException***

Para el método ***decToHex*** se tiene únicamente la siguiente excepción:

- ***NegativeValueEnteredException***

Por lo tanto, se marcarán las excepciones ***InvalidHexException***, ***NegativeValueEnteredException***, ***NotANumberException*** y ***NotADecimalNumberException*** para el método ***hexToDec***, y la excepción ***NegativeValueEnteredException*** para el método ***decToHex***.

2. Creación de las excepciones:

InvalidHexException

El mensaje de esta excepción es: *EL FORMATO NO CORRESPONDE A UN NUMERO HEXADECIMAL*

```
public class InvalidHexException extends Exception {
    public InvalidHexException() {
        super("EL FORMATO NO CORRESPONDE A UN NUMERO HEXADECIMAL");
    }
}
```

NegativeValueEnteredException

El mensaje de esta excepción es: *LOS NUMEROS NEGATIVOS NO SON ACEPTADOS*

```
public class NegativeValueEnteredException extends Exception {
    public NegativeValueEnteredException() {
        super("LOS NUMEROS NEGATIVOS NO SON ACEPTADOS");
    }
}
```

Derivado de la excepción ***InputMismatchException***, se necesita verificar que tipo de error se produce por el dato de entrada. Las posibles excepciones serán marcadas con las clases ***NotANumberException*** y ***NotADecimalNumber***.

notANumberException

El mensaje de esta excepción es: *EL FORMATO NO CORRESPONDE A UN NUMERO*

```
public class NotANumberException extends Exception {

    public NotANumberException() {
        super("EL FORMATO NO CORRESPONDE A UN NÚMERO");
    }
}
```

notADecimalNumberException

El mensaje de esta excepción es: *EL FORMATO NO CORRESPONDE A UN NUMERO DECIMAL*

```
public class NotADecimalNumberException extends Exception {
    public NotADecimalNumberException() {
        super("EL FORMATO NO CORRESPONDE A UN NÚMERO DECIMAL");
    }
}
```

3. Implementación de los métodos de conversiones:

hexToDec

La primera validación que se hace dentro del método es que la cadena recibida no tenga ningún carácter numérico. Para la validación se utiliza una expresión regular. De no tener ningún valor numérico en la cadena recibida se lanza la excepción ***NotANumberExcepcion***.

Suponiendo que un número hexadecimal negativo esté representado de la forma 0X-123ABC o de la forma -0X123ABC, se utilizan otras expresiones regulares para encontrar cadenas con estos patrones. Si la cadena pasada a este método coincide con alguno de estos patrones para números negativos, se arroja la excepción ***NegativeValueEnteredException***.

Luego, se trata de verificar que la cadena recibida sea un intento de convertir un número hexadecimal (que empiece con el prefijo 0X o 0x). Se utiliza otra expresión regular para hacer la validación. Si no empieza con el prefijo 0X o 0x, y dado que sí tiene (pero no únicamente) caracteres

numéricos, se arroja la excepción ***NotADecimalNumberException***.

Si no se presenta el caso de número negativo, se valida que la cadena recibida tenga el formato de número hexadecimal con otra expresión regular. Cualquier cadena que no cumpla con el patrón (debe iniciar con el prefijo 0x o 0X, luego uno o varios caracteres entre A y F o dígitos del 0 al 9), arrojará la excepción ***InvalidHexException***

Si las validaciones se concretan, se convierte la subcadena de *hexString* después del prefijo 0X a un número decimal entero y se retorna este valor.

```
public static int hexToDec(String hexString) throws NegativeValueEnteredException,
InvalidHexException {
        if (Pattern.matches("^0[xX]-[A-F\\d]+$", hexString) || Pattern.matches("^-0[xX][A-
F\\d]+$", hexString))
            throw new NegativeValueEnteredException();

        if (!Pattern.matches("^0[Xx][A-F\\d]+$", hexString))
            throw new InvalidHexException();

        hexString = hexString.substring(2);
        int dec = Integer.parseInt(hexString, 16);
        return dec;
    }
```

decToHex

Para validar que el número recibido sea positivo, se lanza la excepción ***NegativeValueEnteredException*** si el parámetro *dec* es menor a 0. Si se concreta la validación, se convierte el número decimal a hexadecimal y se retorna como un *String*. Es importante recordar que se está asumiendo que datos que no sean de tipo *int* se filtrarán desde el método *hexToDec*.

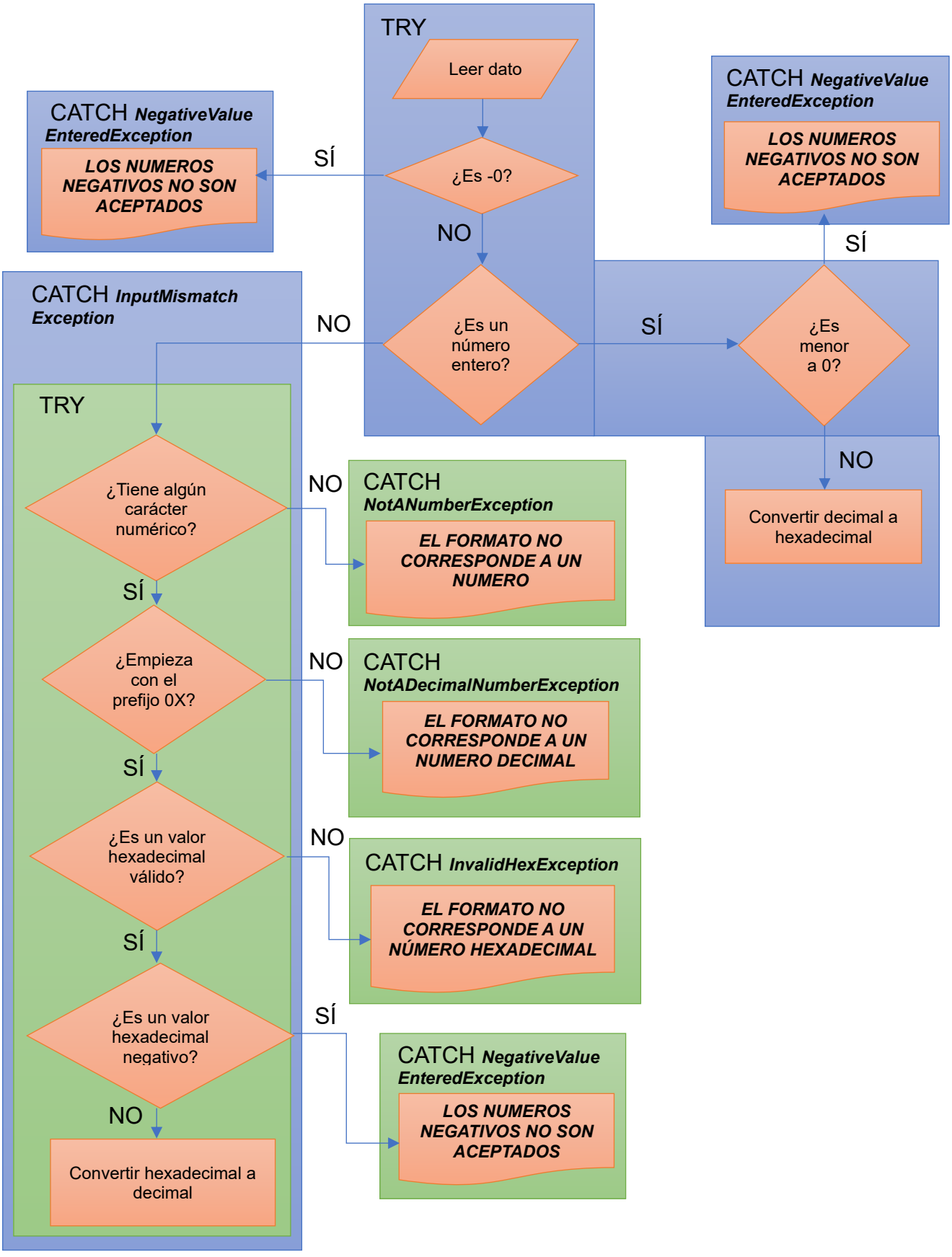
```
public static String decToHex(int dec) throws NegativeValueEnteredException {
        if (dec < 0)
            throw new NegativeValueEnteredException();
        String hexString = Integer.toHexString(dec);
        return "0X" + hexString.toUpperCase();
    }
```

4. Implementación del método *main* para realizar conversiones de datos desde la entrada estándar*:*

La clase *Scanner* arroja la excepción ***InputMismatchException*** con el método *nextInt*. Es decir, si se pretende leer un tipo de dato entero desde el teclado, recibir otro tipo de dato (por ejemplo, un carácter, una cadena, un numero con punto flotante, etc.) arrojará la excepción ***InputMismatchException***. Esta excepción servirá para asegurar que el dato en la entrada del programa sea entero.

Nota: Adicionalmente, para validar el caso en el que el número ingresado sea -0, se debe verificar de manera independiente y previo a la validación numérica para números enteros (si no se hace antes, *Java* toma por iguales los números 0 y -0). Por lo tanto, se agrega la condición por si la entrada del programa es -0. En este caso, se debe arrojar la excepción ***NegativeValueEnteredException*** (y es conveniente marcarla en la declaración del método *main*).

El flujo del programa se puede representar con el siguiente diagrama:


TRY
Leer dato
¿Es -0?
SÍ
CATCH NegativeValue EnteredException
LOS NUMEROS NEGATIVOS NO SON ACEPTADOS
NO
¿Es un número entero?
NO
CATCH InputMismatch Exception
SÍ
¿Es menor a 0?
SÍ
CATCH NegativeValue EnteredException
LOS NUMEROS NEGATIVOS NO SON ACEPTADOS
NO
Convertir decimal a hexadecimal
TRY
¿Tiene algún carácter numérico?
NO
CATCH NotANumberException
EL FORMATO NO CORRESPONDE A UN NUMERO
SÍ
¿Empieza con el prefijo 0X?
NO
CATCH NotADecimalNumberException
EL FORMATO NO CORRESPONDE A UN NUMERO DECIMAL
SÍ
¿Es un valor hexadecimal válido?
NO
CATCH InvalidHexException
EL FORMATO NO CORRESPONDE A UN NÚMERO HEXADECIMAL
SÍ
¿Es un valor hexadecimal negativo?
SÍ
CATCH NegativeValue EnteredException
LOS NUMEROS NEGATIVOS NO SON ACEPTADOS
NO
Convertir hexadecimal a decimal

5. Implementación del método *main* para leer los números de un archivo*:*

Dado que la entrada del programa es el nombre del archivo de texto del cual se tomarán los datos, primero se lee de la entrada estándar una cadena que guardará el nombre del archivo para lectura.

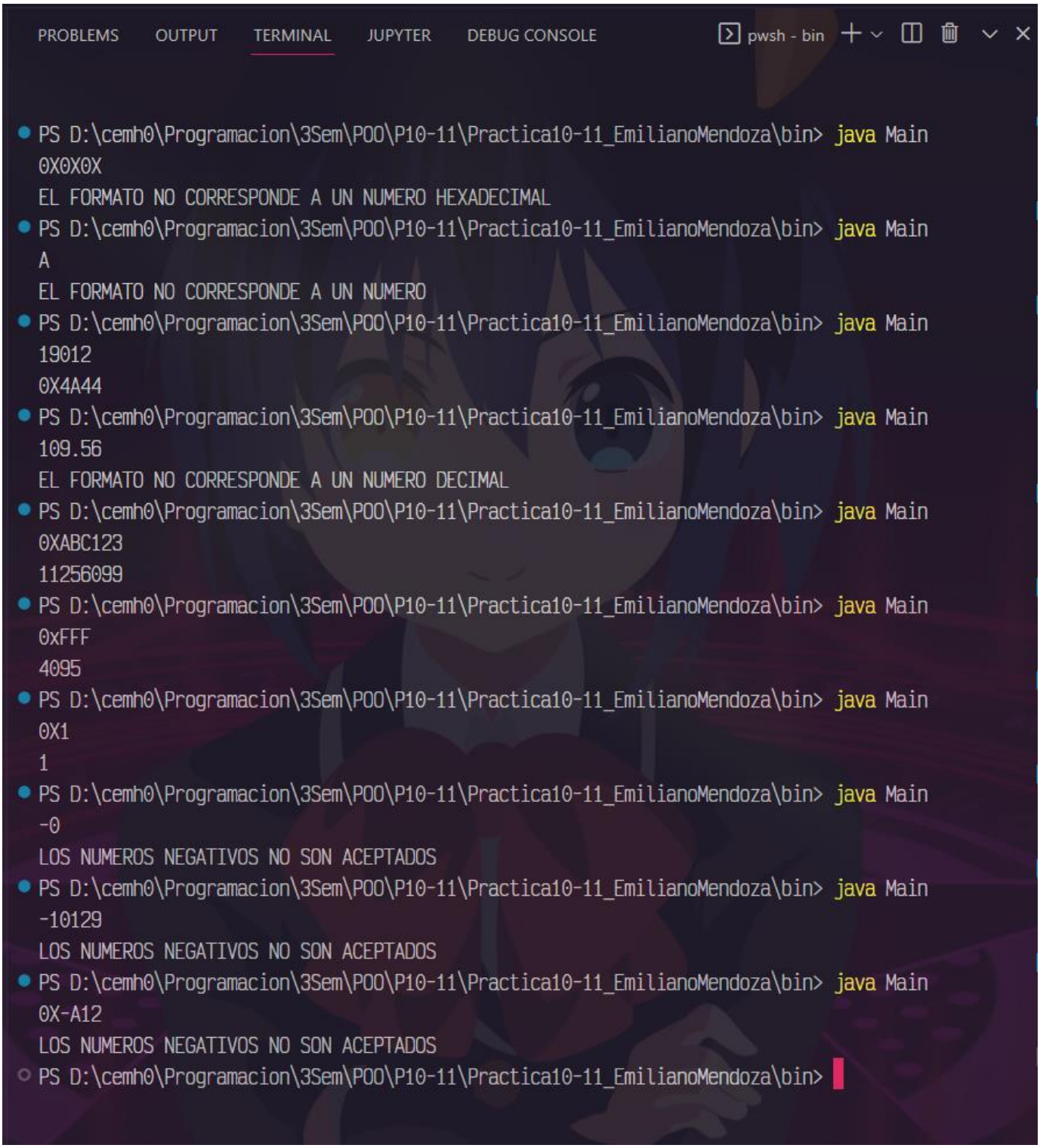
Como paso siguiente, se implementa una estructura *try-catch* para manejar excepciones derivadas con el uso de archivos

En el *try*, se crea un objeto de la clase *FileReader* para poder obtener los caracteres desde la fuente (archivo de texto cuyo nombre se leyó como entrada del programa). Además, se creó un buffer con la clase *BufferedReader* para utilizar el método *readLine* y así leer los caracteres con mayor eficiencia. Mientras el archivo que se está leyendo tenga líneas de texto, se guardan en una cola (esto para capturar todos los datos y poder cerrar el flujo antes de hacer las operaciones).

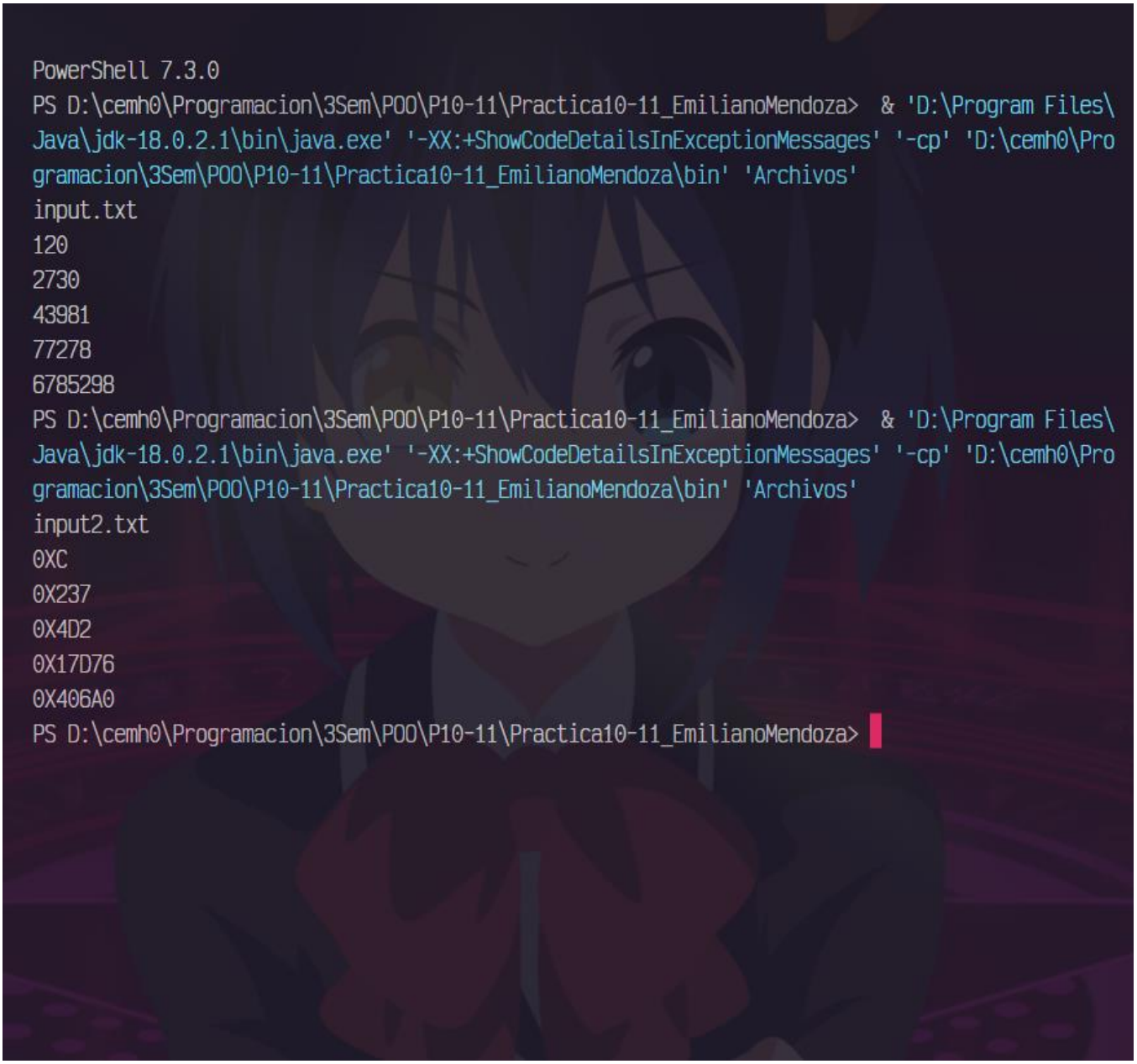
Si se realizó correctamente la lectura de los datos del archivo, se construye otra estructura *try-catch* para manejar las excepciones derivadas del uso de los métodos de conversiones.

6. Por último, se crearon los archivos de texto con las entradas para el programa.

7. Ejecución de método *main* para manejo de excepciones:



8. Ejecución de método *main* para manejo de archivos:



CONCLUSIONES

En esta práctica se implementó el manejo de excepciones para aplicar técnicas adecuadas para responder a estos errores. Para ello, fue importante identificar que bloques de código son propensos a generar errores e identificar los errores que generan. De esta manera se crearon excepciones propias, derivadas de diferentes situaciones y se marcaron dichas excepciones en los métodos propensos a generar estas excepciones. Al usar excepciones marcadas controlamos el uso de estos métodos, siendo necesario el uso de una estructura *try-catch*.

Por otra parte, se implementó el manejo de archivos para esta práctica, leyendo archivos de texto. Es importante tratar los bloques de código relacionados al manejo de archivos con estructuras *try-catch* y manejar los posibles errores que surjan de esto.

Con todo lo anterior, mencionado, considero que se cumplieron los objetivos establecidos para esta práctica.

REFERENCIAS

- *Barnes David, Kölling Michael*
  Programación Orientada a Objetos con Java.
  Tercera Edición.
  Madrid
  Pearson Educación, 2007

- *Deitel Paul, Deitel Harvey*
  Cómo programar en Java.
  Séptima Edición.
  México
  Pearson Educación, 2008

- *Martín, Antonio*
  Programador Certificado Java 2.
  Segunda Edición.
  México
  Alfaomega Grupo Editor, 2008

- *Dean John, Dean Raymond*
  Introducción a la programación con Java.
  Primera Edición.
  México
  Mc Graw Hill, 2009

- *Sierra Katy, Bates Bert*
  SCJP Sun Certified Programmer for Java 6 Study Guide.
  Mc Graw Hill

Yo, Carlos Emiliano Mendoza Hernández, hago mención que esta práctica fue de mi autoría.