	**Carátula para entrega de prácticas**
Facultad de Ingeniería	**Laboratorio de Docencia**

Laboratorio de Computación Salas A y B

Profesor: René Adrián Dávila Pérez

Asignatura: Programación Orientada a Objetos

Grupo: 1

Número de Práctica: 9-10

Integrante(s): 321573670

No. de Lista: N/A

Semestre: 2025-1

Fecha de Entrega: 25 / Octubre / 2024

Observaciones:

Calificación:

Índice

1. Introducción

- **Planteamiento del problema:**
  Los diagramas UML son parte escencial del desarrollo y planeación de cualquier programa pues nos permiten expresar distintas relaciones que existen en el mismo de una forma mucho más comprensible y manejable, el conocimiento y capacidad de plantear correctamente dichos diagramas es algo escencial para la correcta implementación del paradigma.

  De manera similar, el manejo eficiente y correcto de errores y excepciones en nuestro programa es de suma importancia cuando consideramos el desarrollo del mismo, es por ello que se vuelve parte fundamental conocer la manera de manejar dichas situaciones e implementar reacciones adecuadas del programa.

  Se busca que los programas descritos a continuación sirvan de apoyo en el proceso de adquisición de dichos conocmientos por parte del alumnado:

  - Un programa en el que se defina una excepción personalizada, con base en el bloque `try-catch` para capturar la división entre cero.
  - Un programa para el cual se defina una excepción personalizada, basándonos en la cláusula `throws`, para manejar la raíz cuadrada de un número negativo.
  - Un programa para el cual se escriba una excepción personalizada con la cual se indique un error si es que existen números repetidos en una lista de enteros previamente ingresada por el usuario.
  - Un programa con el cual se lea una cadena y se haga uso de una excepción personalizada para imprimir un error en caso de que no contenga vocales.

- **Motivación:**
  La correcta implementación de los programas propuestos brindará al alumnado un profundo conocimiento sobre la implementación del manejo de excepciones en Java y los posibles usos que se le puede dar a dicha utilidad. Por otra parte la realización de los diagramas UML permitirá el desarrollo de una visión a la crucialidad y utilidad de los mismos al momento de la organización y planeación de un programa.

- **Objetivos:**
  Se desea que el alumnado pueda desarrollar correctamente los problemas propuestos, que sea capaz de comprender las implementaciones y funcionalidad del manejo de exepciones y que conozca e implemente distintos tipos de diagramas UML.

2. Marco Teórico

2.1. Manejo de excepciones.

El desarrollo de cualquier programa puede volverse muy abstracto y su implementación puede resultar bastante compleja, esto hace que los programas sean propensos a que se presenten errores en la ejecución, es por ello qe la estructura y manejo de excepciones en Java se vuelve esencial para garantizar la estabilidad y confiabilidad de los programas, permitiendo capturar, identificar y gestionarlos de manera eficiente [1].

El manejo de excepciones en Java sigue un modelo estructurado que permite capturar y manejar errores mediante bloques `try-catch-finally`. La palabra clave `try` define un bloque en el que se puede lanzar una excepción, mientras que `catch` permite capturar una excepción específica y realizar el tratamiento adecuado. Finalmente, `finally` asegura la ejecución de código independientemente de si ocurre o no una excepción [2].

Además, Java permite la creación de excepciones personalizadas mediante la definición de nuevas clases que extiendan `Exception`. Esto facilita el manejo de errores específicos en aplicaciones complejas, garantizando un flujo de ejecución controlado y comprensible. Por ejemplo, las excepciones personalizadas son útiles para gestionar situaciones únicas como divisiones por cero, raíces cuadradas de números negativos o validaciones de listas de datos [3].

2.2. Diagramas UML y su Clasificación

La técnica de modelado UML es ampliamente utilizada para visualizar y planificar el diseño de software. Existen varios tipos de diagramas UML, divididos en dos categorías principales: diagramas de estructura y diagramas de comportamiento [4].

Diagramas de Estructura: Estos diagramas muestran la estructura estática del sistema.

- **Diagrama de Clases:** Representa las clases, atributos, métodos y relaciones entre ellas. Es útil para visualizar la estructura general de un sistema [5].
- **Diagrama de Objetos:** Muestra instancias específicas de las clases en un momento particular de la ejecución.
- **Diagrama de Componentes:** Representa la disposición de los componentes del sistema.

Diagramas de Comportamiento: Estos diagramas ilustran el comportamiento dinámico del sistema.

- **Diagrama de Secuencia:** Representa la interacción entre objetos en el tiempo.
- **Diagrama de Actividades:** Muestra el flujo de trabajo del sistema.
- **Diagrama de Estados:** Representa los diferentes estados de un objeto y sus transiciones [6].

La correcta implementación y uso de estos diagramas ayuda a estructurar y simplificar el diseño de sistemas complejos, fomentando una mejor comunicación entre los desarrolladores y facilitando la comprensión de programas con alta abstracción [7].

3. Desarrollo

- Ejercicio 0:

  Para este ejercicio se buscó implementar un método con el cual se realicé la división en el cual se maneje la excepción en la que se intenta calcular con un denominador 0.

  En primer lugar se definió una clase "DivisionPorCeroException" la cual extiende a la clase "Exception", dentro de dicha clase se definió su constructor el cual toma como parámetro un `String` y hace uso de `super()` para realizar la construcción.

  Posteriormente se definió la clase "OperacionMatematica", en este se definió `dividir()` con parámetros de tipo doble para el numerador y

denominador, en dicho método se compara el denominador con 0, en caso de cumplirse se lanza la excepción definida con el mensaje “No es posible dividir entre cero...”, si el denominador no es cero se retorna el cálculo correspondiente.

Finalmente se definió la clase principal en la cual se instanció la clase “OperacionMatematica”, posteriormente se implementó la funcionalidad de todo el programa dentro de un bloque `try-catch`, en caso de que la división sea entre cero se manda a imprimir el error generado con el mensaje del objeto de la excepción.

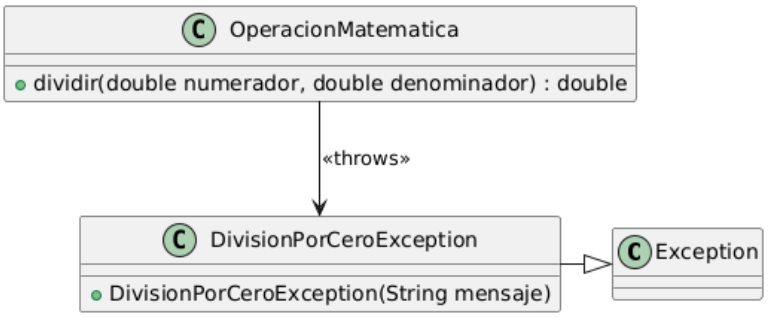


Figura 1: Diagrama UML de clases

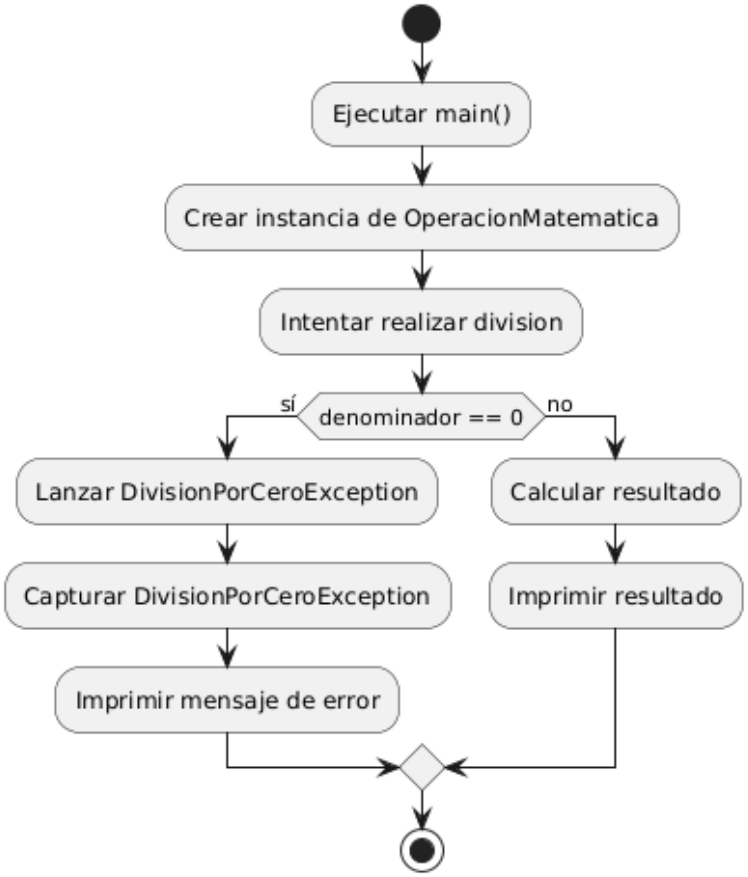


Figura 2: Digrama UML de actividades

- Ejercicio 1:

  En este ejercicio se buscaba la implementación de un programa para calcular la raíz de un número dado en la que se pudiese manejar la excepción en la que el número es negativo.

  Se comenzó por la implementación de una clase para la excepción, esta fue denominada "RaizNegatizaException" que extiende a la clase "Exception", dentro de la misma se define un constructor que hace uso de `super()`.

  Después se implementó la clase "OperacionMatematica2" en la cual se definió el método `raizCuadrada()` en la que se verifica que el número

sea mayor a 0 en cuyo caso se retorna la operación, en caso contrario se lanza la excepción definida.

Por último se definió la clase principal, de manera similar al ejercicio anterior se definió todo dentro de un bloque `try-catch` en el que se maneja la excepción.

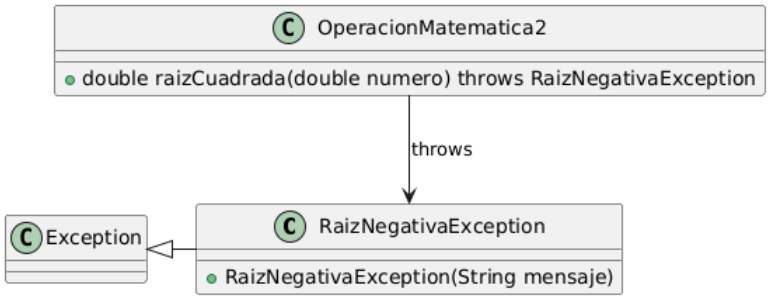


Figura 3: Diagama UML de clases

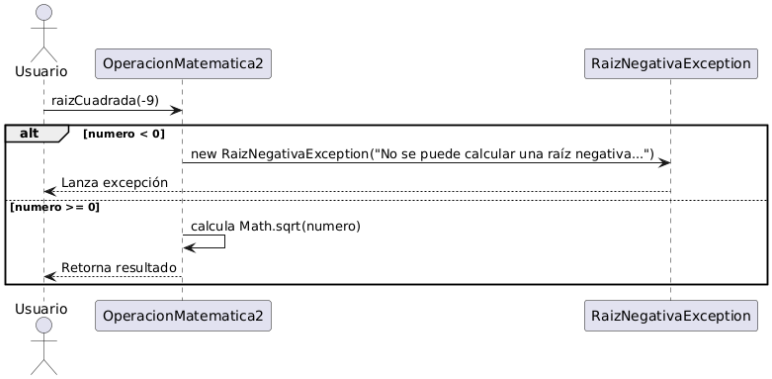


Figura 4: Diagrama UML de secuencia

- Ejercicio 2:

  La resolución de este ejercicio consistió en el desarrollo de un programa en el que se comparan los enteros de una lista, los cuales son ingresados por el usuario, en caso de la existencia de números repetidos se manejaría con una excepción.

  Primero se definió la excepción dentro de la clase "NumeroDuplicadoException" la cual extiende a la clase "Exception", para esta se definió su

contructor que a su vez hace uso del constructor de “Exception” por medio de `super()`.

Una vez definida la excepción se definió la funcionalidad para la lectura y almacenamiento de los enteros ingresados por el usuario, para esto se definió la clase “IngresaUsuario” con el método `leerNumerosUsuario()` dentro del cual se definió un iterador en el cual se ingresa cierta cantidad de enteros por parte del usuario, mismos que se agregar a una `ArrayList<>`, finalmente se retorna la lista.

Posteriormente se definió la clase “RevisionDuplicado” en la cual se definió el método ‘`checarDuplicado()`, en este método se definió la lógica para la revisión de los números duplicados haciendo uso de un ciclo `for` y un condicional dentro de este cuya funcionalidad es revisar si se repetian los valores y de ser así lanzar la excepción.

Por último se definió la clase principal, en esta se hizo uso del bloque `try-catch` para implementar la lógica de ejecución y el manejo de la excepción en caso de que se repitan valores.

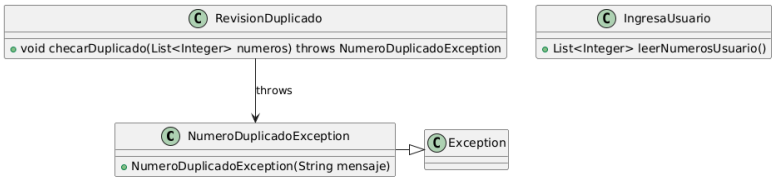


Figura 5: Diagrama UMl de clases

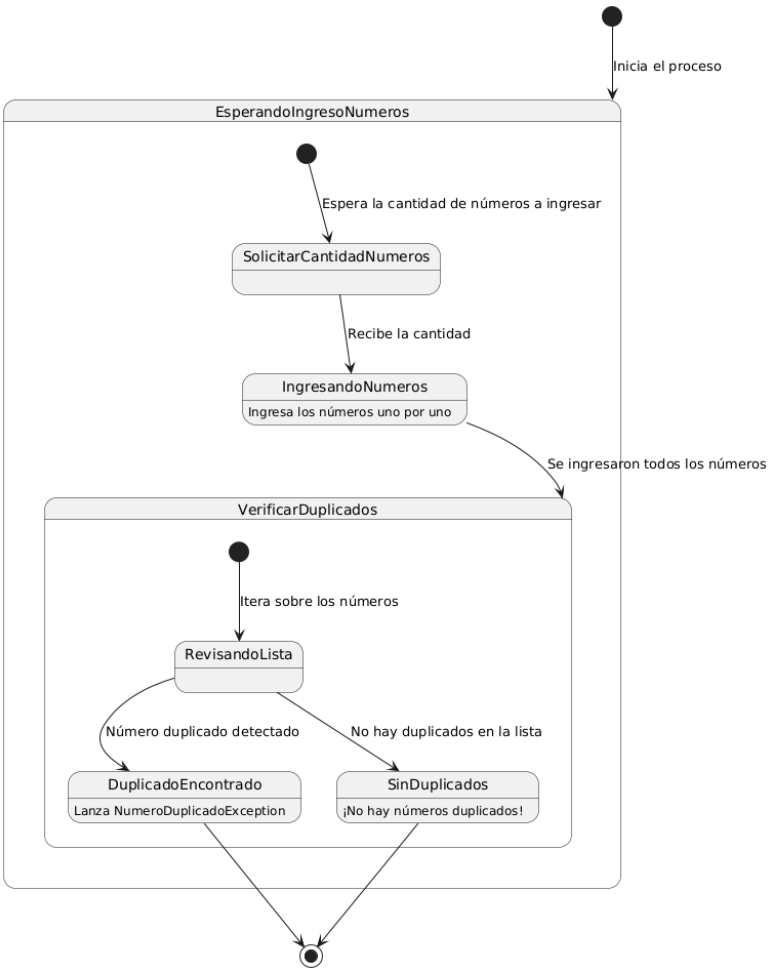


Figura 6: Diagrama tipo state machine

- Practica 9-10:

  Para el ejercicio final de la práctica se solicitó la implementación de un programa en el cual se pueda realizar la revisión de una cierta palabra para conocer si es que contiene vocales, en caso contrario se manejaría la excepción.

  Se comenzó por definir la excepción para la falta de vocales, esto se logró a través de la definición de la clase "ExcepcionNoVocales" que extiende a "Exception", en su constructor se hace uso de `super()` para la reutilización de código.

A continuación se implementó la lógica para la comparación en una clase "LecturaCadena". Para dicha clase se definieron tres atributos, un atributo privado para la palabra a analizar y dos atributos estáticos para un contador y un arreglo con las vocales. El constructor de la clase toma como parámetro un `String` que posteriormente define como la palabra que será analizada haciendo uso de su método `setPalabra()`. Finalmente se definió el método `revisarVocales()` con el cual se verifica la presencia de vocales en la palabra dada, en caso de que la palabra no cuente con ninguna vocal se lanza la excepción, esto se logro haciendo uso de un `for-each` para iterar a través de cada letra en la palabra, con dos condicionales anidados dentro.

Para finalizar se definió la clase principal en la cual se realiza la lectura de la cadena gracias a un objeto "Scanner", posteriormente se instancia la clase "LecturaCadena" con la palabra como parámetro del constructor, finalmente se realiza la revisión. Al estar todo definido dentro de un bloque `try-catch`, el manejo de la excepción significa que si la palabra no contiene vocales se imprimirá el mensaje de error adecuado.

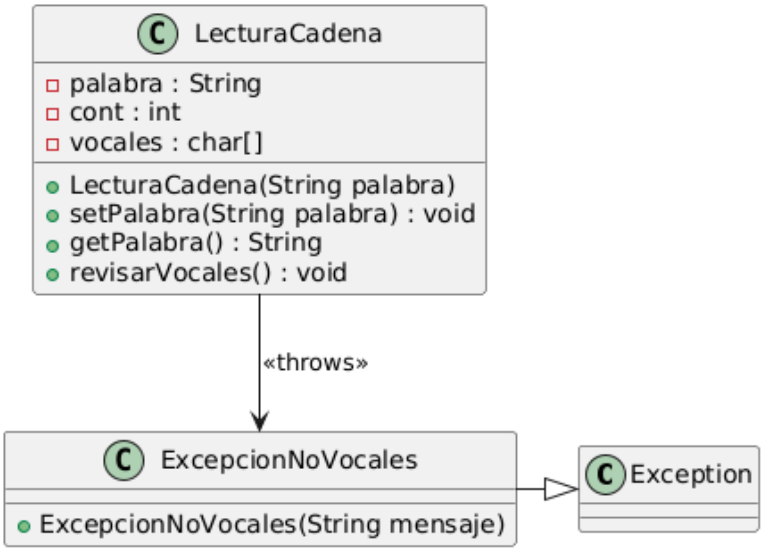


Figura 7: Diagrama UML para las clases del programa.

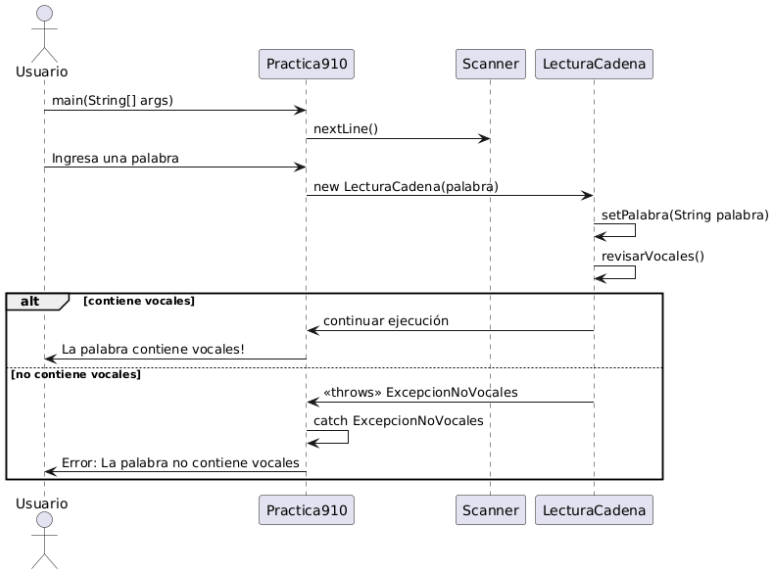


Figura 8: Diagrama UML para la secuencia del programa

4. Resultados

- Ejercicio 0:

  Código:

```java
class DivisionPorCeroException extends Exception{
    public DivisionPorCeroException(String mensaje){
        super(mensaje);
    }
}
```

Figura 9: Definición de la excepción para manejar la división entre cero.

```java
class OperacionMatematica{
    public double dividir(double numerador, double denominador) throws DivisionPorCeroException{
        if(denominador == 0){
            throw new DivisionPorCeroException("No es posible dividir entre cero...");
        }
        return numerador / denominador;
    }
}
```

Figura 10: Clase para la cual se define el método `dividir()` en el cual se calcula dicha operación y en caso de que el denominador sea 0 se lanza la excepción definida.

```java
public class Ejercicio0{
    Run main | Debug main
    public static void main(String[] args) {
        OperacionMatematica operacion = new OperacionMatematica();
        try {
            double resultado = operacion.dividir(10,0);
            System.out.println("Resultado: "+ resultado);
        } catch (DivisionPorCeroException e) {
            System.out.println("Error: "+ e.getMessage());
        }
    }
}
```

Figura 11: Clase principal con el bloque `try-catch` en el cual se manda a llamar el método para la división y se imprime el resultado, en caso de que el denominador sea 0 se imprime el error.

Ejecución:

```
Error: No es posible dividir entre cero...
```

Figura 12: Ejecución con un denominador 0.

- Ejercicio 1:

  Código:

```java
class RaizNegativaException extends Exception{
    public RaizNegativaException(String mensage){
        super(mensage);
    }
}
```

Figura 13: Clase en la que se declara la excepción para el cálculo de la raíz negativa.

```java
class OperacionMatematica2{
    public double raizCuadrada(double numero) throws RaizNegativaException{
        if(numero < 0){
            throw new RaizNegativaException("No se puede calcular una raíz negativa...");
        }
        return Math.sqrt(numero);
    }
}
```

Figura 14: Clase en la que se define el método para efectuar la operación de la raíz cuadrada, en caso de que el número a dividir sea menor a 0 se lanza la excepción definida.

```java
public class Ejercicio1{
    Run main | Debug main
    public static void main(String[] args) {
        OperacionMatematica2 operacion = new OperacionMatematica2();

        try {
            double resultado = operacion.raizCuadrada(-9);
            System.out.println("Resultado: "+resultado);
        } catch (RaizNegativaException e) {
            System.out.println("Error: "+e.getMessage());
        }
    }
}
```

Figura 15: Clase principal en la que se efectua la operación dentro de un bloque `try-catch`, en caso de ser negativa se imprime el error pertinente.

Ejecución:

```
Error: No se puede calcular una raíz negativa...
```

Figura 16: Ejecución con una raíz negativa.

- Ejercicio 2:

  Código:

```java
class NumeroDuplicadoException extends Exception{
    public NumeroDuplicadoException(String mensaje){
        super(mensaje);
    }
}
```

Figura 17: Clase para la definición de la excepción en caso de encontrar un número duplicado.

```java
class IngresaUsuario{
    public static List<Integer> leerNumerosUsuario() {
        List<Integer> numeros = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);

        System.out.println(x:"¿Cuántos números deseas ingresar?");
        int cont = scanner.nextInt();

        System.out.println(x:"Ingresa los enteros: ");
        for (int i = 0; i < cont; i++) {
            int num = scanner.nextInt();
            numeros.add(num);
        }
        scanner.close();
        return numeros;
    }
}
```

Figura 18: Clase en la que se define la lógica para la lectura de los enteros a comparar, estos son agregados a un `ArrayList<>` de enteros.

```java
class RevisionDuplicado{
    public static void checarDuplicado(List<Integer> numeros) throws NumeroDuplicadoException{
        Set<Integer> numerosUnicos = new HashSet<>();

        for (int num : numeros) {
            if (numerosUnicos.contains(num)){
                throw new NumeroDuplicadoException(mensaje:"Número duplicado encontrado.");
            }
            numerosUnicos.add(num);

        }
    }
}
```

Figura 19: Clase en la cual se implementa la revisión de los números, iterando a través de la lista, en caso de que se encuentre un número duplicado se lanza la excepción.

```java
public class Ejercicio2{
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        try {
            List<Integer> numeros = IngresaUsuario.leerNumerosUsuario();
            RevisionDuplicado.checarDuplicado(numeros);
            System.out.println(x:"¡No hay  números duplicados!");
        } catch (NumeroDuplicadoException e) {
            System.out.println("Error: "+ e.getMessage());
        }
    }
}
```

Figura 20: Clase principal en la que se define, dentro de un bloque `try-catch`, la funcionalidad para revisar la lista de números ingresador por el usuario y lanza el error en caso de la existencia de duplicados.

Ejecución:

```
¿Cuántos números deseas ingresar?
6
Ingresa los enteros:
3
4
5
6
7
8
¡No hay  números duplicados!
```

Figura 21: Ejecución en la que no se encuentran números duplicados.

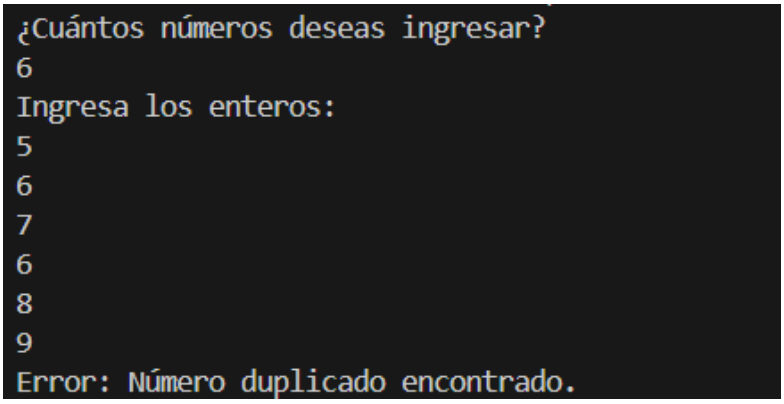


Figura 22: Ejecución en la que se encuentran números duplicados y se imprime el error en terminal.

- Practica 9-10:

  Código:

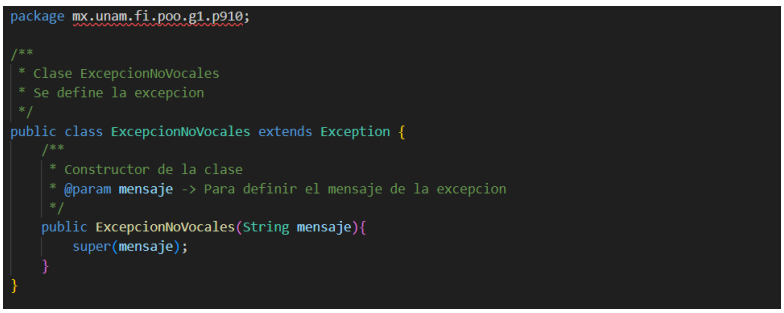
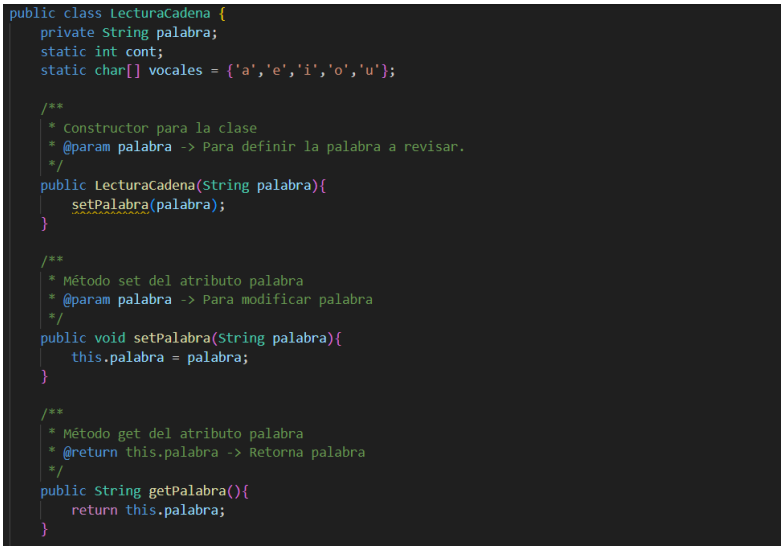


Figura 23: Clase en la que se define la excepción para el caso de no encontrar vocales en la cadena.



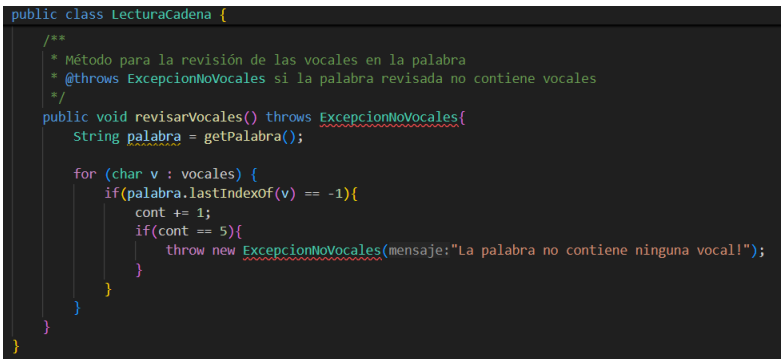
```java
public class LecturaCadena {
    private String palabra;
    static int cont;
    static char[] vocales = {'a','e','i','o','u'};

    /**
     * Constructor para la clase
     * @param palabra -> Para definir la palabra a revisar.
     */
    public LecturaCadena(String palabra){
        setPalabra(palabra);
    }

    /**
     * Método set del atributo palabra
     * @param palabra -> Para modificar palabra
     */
    public void setPalabra(String palabra){
        this.palabra = palabra;
    }

    /**
     * Método get del atributo palabra
     * @return this.palabra -> Retorna palabra
     */
    public String getPalabra(){
        return this.palabra;
    }
```

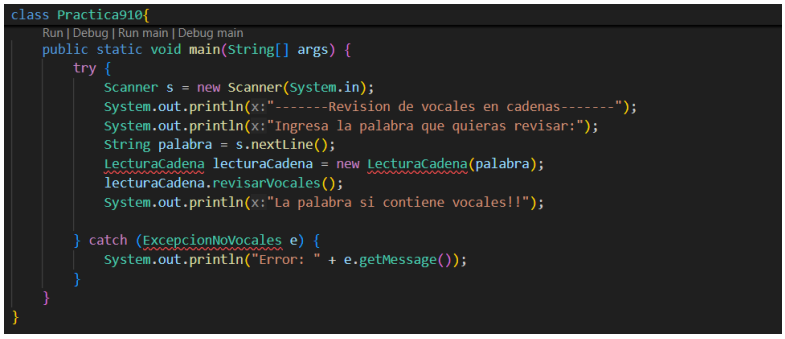
Figura 24: Clase con la cual se realiza la revisión, en esta parte del código se define el constructor de la clase, los atributos estáticos y privados y los respectivos métodos para acceder a dichos atributos.



```java
public class LecturaCadena {
    /**
     * Método para la revisión de las vocales en la palabra
     * @throws ExcepcionNoVocales si la palabra revisada no contiene vocales
     */
    public void revisarVocales() throws ExcepcionNoVocales{
        String palabra = getPalabra();

        for (char v : vocales) {
            if(palabra.lastIndexOf(v) == -1){
                cont += 1;
                if(cont == 5){
                    throw new ExcepcionNoVocales(mensaje:"La palabra no contiene ninguna vocal!");
                }
            }
        }
    }
}
```

Figura 25: Parte de la clase en la que se define el método para la revisión de las vocales presentes en la cadena, en caso de no encontrar ninguna se lanza la excepción.

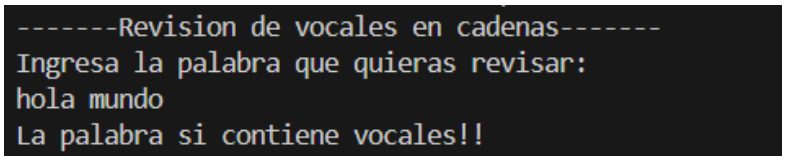


```java
class Practica910{
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        try {
            Scanner s = new Scanner(System.in);
            System.out.println(x:"-------Revision de vocales en cadenas-------");
            System.out.println(x:"Ingresa la palabra que quieras revisar:");
            String palabra = s.nextLine();
            LecturaCadena lecturaCadena = new LecturaCadena(palabra);
            lecturaCadena.revisarVocales();
            System.out.println(x:"La palabra si contiene vocales!!");

        } catch (ExcepcionNoVocales e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

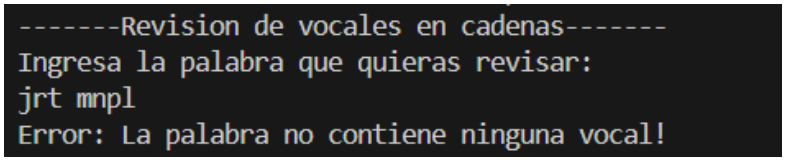
Figura 26: Clase principal en la que se define la funcionalidad para leer la cadena y posteriormente realizar la resvisión, en caso de que no contenga vocales se manda a imprimir el error con la excepción, todo esto dentro de un bloque `try-catch`.

Ejecución:



```
-------Revision de vocales en cadenas-------
Ingresa la palabra que quieras revisar:
hola mundo
La palabra si contiene vocales!!
```

Figura 27: Ejecución con una palabra con vocales.



```
-------Revision de vocales en cadenas-------
Ingresa la palabra que quieras revisar:
jrt mnpl
Error: La palabra no contiene ninguna vocal!
```

Figura 28: Ejecución en la que la palabra no tiene vocales.

5. Conclusiones

La correcta finalización de la práctica nos habla sobre el correcto desarrollo de todo lo esperado al inicio de la misma, los programas propuestos fueron solucionados de manera exitosa, implementando los conceptos que se esperaba implementar correctamente, además esto resultó en la cimentación

del conocimiento por parte del alumnado y de la ampliación de la visión en cuanto a la funcionalidad de los conceptos de manejo de errores y diagramas UML al momento de desarrollar un programa siguiendo el paradigma.

En general la práctica resultó fructífera y los objetivos fueron alcanzados de la forma en que se deseaba.

Referencias

[1] C. S. Horstmann, *Core Java Volume I–Fundamentals*. Prentice Hall, 2007.

[2] J. Bloch, *Effective Java*. Addison-Wesley, 2008.

[3] "Java Documentation: Exceptions." (2023), dirección: `https://docs.oracle.com/javase/tutorial/essential/exceptions/`.

[4] J. Rumbaugh, G. Booch e I. Jacobson, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[5] G. Booch, *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994.

[6] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[7] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.