Practica 8 Excepciones

> **Objetivo** Conocer e interpretar los distintos modelos de excepciones que implementan los lenguajes de programación

Ejercicio 1

¿Explique claramente a qué se denomina excepción?

Una excepción es una situación inesperada o inusual que surge durante la ejecución de un programa y no puede ser manejada en el contexto actual.

Ejercicio 2

¿Qué debería proveer un lenguaje para el manejo de las excepciones?

Un lenguaje debería proveer un mecanismo para el manejo de excepciones. Debe permitir que los programadores puedan lanzar excepciones explícitamente en su código cuando detecten una situación excepcional o un error. También debe permitir la captura de excepciones para manejarlas de forma adecuada. Un lenguaje debería permitir la definición de diferentes tipos de excepciones ayudando a clasificar las excepciones en categorías más específicas.

¿Todos los lenguajes lo proveen?

No todos los lenguajes de programación proporcionan un mecanismo completo para el manejo de excepciones. Algunos lenguajes más antiguos o de bajo nivel pueden tener un enfoque más limitado o no proporcionar un manejo de excepciones integrado. Sin embargo, la mayoría de los lenguajes modernos y populares, como Java, C++, Python y C#, sí brindan soporte completo para el manejo de excepciones. Estos lenguajes tienen palabras clave y sintaxis específicas para lanzar, capturar y manejar excepciones de manera efectiva.

Ejercicio 3

¿Qué ocurre cuando un lenguaje no provee manejo de excepciones?

Cuando un lenguaje de programación no proporciona un mecanismo integrado para el manejo de excepciones, puede ser más difícil manejar situaciones excepcionales o errores de manera adecuada. Sin un sistema de manejo de excepciones, los errores pueden propagarse y afectar el flujo normal del programa o, en el peor de los casos, provocar fallas inesperadas.

Si un lenguaje no ofrece soporte nativo para el manejo de excepciones, aún es posible simular un mecanismo de manejo de excepciones utilizando técnicas alternativas. Una forma común de hacerlo es mediante el uso de estructuras de control condicionales (por ejemplo, if-else) y la utilización de valores de retorno especiales para indicar errores.

¿Se podría simular? Explique cómo lo haría

Se podría simular de la siguiente manera:

- 1 Identificar situaciones excepcionales
- 2 Devolver valores de error (ya que no hay excepciones)
- 3 Validar valores de retorno (después de llamados a funciones o métodos)
- 4 Propagación manual de errores
- 5 Usar bloques try-catch simulados con if-else.

Ejercicio 4

Cuando se termina de manejar la excepción, la acción que se toma luego es importante. Indique

¿Qué modelos diferentes existen en este aspecto?

En cuanto a los modelos de manejo de excepciones, existen principalmente dos enfoques:

- **Reasunción**: después de manejar una excepción, la ejecución continúa desde el punto donde se lanzó la excepción, como si no hubiera ocurrido ningún error. El código dentro del bloque de manejo de excepciones se ejecuta y luego se reanuda el flujo normal del programa.
- **Terminación**: después de manejar una excepción, la ejecución no continúa desde el punto donde se lanzó la excepción. En cambio, se produce una terminación abrupta del flujo normal del programa y se salta a un punto de retorno predefinido o se realiza una acción específica definida por el manejador de la excepción.

Dé ejemplos de lenguajes que utilizan cada uno de los modelos presentados anteriormente. Por cada uno responda respecto de la forma en que trabaja las excepciones.

Ejemplos de lenguajes que utilizan estos modelos de manejo de excepciones:

- `1)` PL/1 utiliza el criterio de reasunción. Las excepciones se llaman **conditions**. Los manejadores se declaran con la sentencia ON: `ON CONDITION(Nombre-excepción) Manejador` . El manejador puede ser una instrucción o un bloque. ADA utiliza el criterio de terminación. Las excepciones se definen en la zona de definición de las variables y tienen el mismo alcance.
- `2)` En PL/1 las excepciones se lanzan explícitamente con la palabra clave `Signal condition (Nombre-excepción)` . En ADA se pueden lanzar excepciones explícitamente con la palabra clave `raise` .
- `3)` PL/1 tiene una serie de excepciones ya predefinidas con su manejador asociado. Son las **Built-in exceptions**. Los manejadores se ligan **dinámicamente** con las excepciones. Una excepción siempre estará ligada con el último manejador definido. El alcance de un manejador termina cuando finaliza la ejecución de la unidad donde fue declarado. En ADA los manejadores pueden encontrarse en el final de cuatro diferentes unidades del programa: Bloque, Procedimiento, Paquete o Tarea. Al producirse una excepción, se termina la unidad y si tiene un manejador en ese ámbito se ejecuta, sino se propaga dinámicamente (se lleva a otro ámbito).

¿Cuál de esos modelos es más inseguro y por qué?

El modelo de terminación es considerado más inseguro que el modelo de reasunción. Esto se debe a que en el modelo de terminación, después de manejar una excepción, se produce una terminación abrupta del flujo normal del programa, lo que puede dejar recursos sin liberar o dejar el programa en un estado inconsistente. Además, el control del flujo de ejecución puede ser más difícil de seguir y razonar en comparación con el modelo de finalización normal, donde la ejecución continúa desde el punto donde se lanzó la excepción. Por lo tanto, se considera que el modelo de finalización normal proporciona una mayor seguridad y control en el manejo de excepciones.

Ejercicio 5

La propagación de los errores, cuando no se encuentra ningún manejador asociado, no se implementa igual en todos los lenguajes. Realice la comparación entre el modelo de Java, Python y PL/1, respecto a este tema. Defina la forma en que se implementa en un lenguaje conocido por Ud.

La forma en que se implementa la propagación de errores cuando no se encuentra ningún manejador asociado difiere entre Java, Python y PL/1:

- **Java**: En Java, si no se encuentra ningún manejador asociado para una excepción, se produce una terminación abrupta del programa y se muestra un mensaje de error en la consola. El flujo de ejecución se detiene y se muestra un stack trace que indica el tipo de excepción y la secuencia de llamadas que condujeron al error.
- **Python**: En Python, si no se encuentra ningún manejador asociado para una excepción, el programa también se termina abruptamente y se muestra un traceback en la consola. Similar a Java, se indica el tipo de excepción y la secuencia de llamadas que llevaron al error.
- **PL/1**: En PL/1, cuando no se encuentra un manejador asociado para una excepción, el programa se detiene y se produce una terminación abrupta, generando un dump de diagnóstico que muestra información sobre el estado del programa y el punto en el que se produjo la excepción. Este dump de diagnóstico es utilizado para el análisis y depuración del programa.

En resumen, tanto Java como Python presentan una terminación abrupta del programa cuando no se encuentra ningún manejador para una excepción. Muestran un mensaje de error y un traceback que proporciona información sobre el error. Por otro lado, en PL/1, se produce una terminación abrupta del programa y se genera un dump de diagnóstico para facilitar el análisis y la depuración.

Ejercicio 6

Sea el siguiente programa escrito en Pascal

```
...
Procedure Manejador;
  Begin ... end;
Procedure P(X:Proc);
  begin
  ....
  if Error then X;
  ....
end;

Procedure A;
begin
  ....
  P(Manejador);
  ....
end;
....
```

¿Qué modelo de manejo de excepciones está simulando?

Está manejando el modelo de reasunción.

¿Qué necesitaría el programa para que encuadre con los lenguajes que no utilizan este modelo? Justifique la respuesta.

Debería tener algún tipo de sentencia determinante en el manejador, cosa que cuando se ejecute se acabe el problema.

Ejercicio 7

Sea el siguiente programa escrito en Pascal

```
Program Principal;
var x:int; b1,b2:boolean;

Procedure P (b1:boolean);
  var x:int;
  Procedure Manejador1
  begin
    x:=x + 1;
  end;
begin
  x:=1;
  if b1=true then Manejador1;
  x:=x+4;
end;

Procedure Manejador2;
begin
  x:=x * 100;
end;
```

```
Begin
  x:=4;
  b2:=true;
  b1:=false;
  if b1=false then Manejador2;
  P(b);
  write (x);
End
```

a) Implemente este ejercicio en PL/1 utilizando manejo de excepciones

```
Prog Principal
DCL x DEC FIXED (3,2) INIT (0);
// como se instancian los boolean en pl1?
    PROC P
        DCL x DEC FIXED (3,2) INIT (0);
        ON CONDITION Manejador1 begin
            x = x + 1;
        end;
        x = 1;
        if(b1 = true) then SIGNAL CONDITION Manejador1
        x = x + 4
        ON CONDITION Manejador2 begin
                x = x * 100;
        end;
    end;
Begin
    x = 4;
    b2 = true;
    b1 = false;
    // el manejador 2 deberia estar acá porque sino se rompe todo
    ON CONDITION Manejador2 begin
    if(b1 = false) then SIGNAL CONDITION Manejador2;
    P;
    PUT SKIP LIST(x);
end;
```

b) ¿Podría implementarlo en JAVA utilizando manejo de excepciones? En caso afirmativo, realícelo.

Teóricamente no, porque esto es con el modelo de reasunción, TENGO QUE CONSULTAR.

Ejercicio 8.

Sean los siguientes, procedimientos de un programa escrito en JAVA

```
public static void main (String[] argos){
  Double array_doubles[]= new double[500];
  for (int i=0; i<500; i++){
    array_doubles[i]=7*i;
  }
  for (int i=0 ; i<600 ; i=i+25){
    try{
      system.out.println(“El elemento en “+ i + ” es “ + acceso_por_indice (array_doubles,i));
    }
    catch(ArrayIndexOutOfBoundsException e){
      system.out.println(e.tostring());
    }
    catch(Exception a){
      system.out.println(a.tostring());
    }
    finally{
      system.out.println(“sentencia finally”);
    }
  }
}
Public static double acceso_por_indice (double [] v, int indice) throws Exception; ArrayIndexOutOf
  if ((indice>=0) && (indice<v.length)){
    Return v[indice];
  }
  else{
    if (indice<0){
      // caso excepcional
      Throw new ArrayIndexOutOfBoundsException(“ el índice” + indice + “ es un número negativo”);
      }
    else{
      // caso excepcional
      Throw new Exception(“ el indice” + indice + “ no es una posición válida”);
    }
  }
}
```

a) Analizar el ejemplo y decir qué manejadores ejecuta y en qué valores quedan las variables. JUSTIFIQUE LA RESPUESTA.

Imprime:

- try: “El elemento 0 es 0”, finally: “sentencia finally”
- try: “El elemento 25 es 175”, finally: “sentencia finally”
- try: “El elemento 50 es 350”, finally: “sentencia finally”
- try: “El elemento 75 es 525”, finally: “sentencia finally”
- …

- catch: “El indice 500 no es una posición válida”, finally: “sentencia finally”
- catch: “El indice 525 no es una posición válida”, finally: “sentencia finally”
- catch: “El indice 550 no es una posición válida”, finally: “sentencia finally”
- catch: “El indice 575 no es una posición válida”, finally: “sentencia finally”

b) La excepción se propaga o se maneja en el mismo método? ¿Qué instrucción se agrega para poder propagarla y que lleve información?.

Se propaga del método `acceso_por_indice` al main. Se agrega la instrucción `Throw new Exception`, que lo recibe el main con el `catch(ArrayIndexOutOfBoundsException e)`.

c) como modificaría el método “acceso_por_indice” para que maneje él mismo la excepción

Debería haber un try-catch dentro del método.

Ejercicio 9

Indique diferencias y similitudes entre Phyton y Java con respecto al manejo de excepciones.

Tanto Python como Java tienen mecanismos de manejo de excepciones para controlar y responder a situaciones de error en tiempo de ejecución. A continuación, se presentan algunas diferencias y similitudes entre Python y Java en cuanto al manejo de excepciones:

Diferencias

- **Sintaxis** Python utiliza bloques `try-except` para capturar y manejar excepciones, mientras que Java utiliza bloques `try-catch-finally`. La sintaxis en Python es más concisa y utiliza menos palabras clave que en Java.
- **Tipos de excepciones** En Java, las excepciones se dividen en dos categorías principales: excepciones verificadas (checked exceptions) y excepciones no verificadas (unchecked exceptions). Las excepciones verificadas deben ser declaradas en la firma del método o manejadas explícitamente, mientras que las excepciones no verificadas no requieren acciones específicas de manejo. En Python, no hay una distinción estricta entre excepciones verificadas y no verificadas, todas las excepciones pueden ser manejadas sin requerir una declaración explícita.
- **Manejo de múltiples excepciones** En Python, se puede manejar más de una excepción en un solo bloque `except` utilizando una tupla de tipos de excepción. En Java, se requiere un bloque `catch` separado para cada tipo de excepción que se desea manejar.
- **Excepciones personalizadas** En Java, es común definir y lanzar excepciones personalizadas creando nuevas clases que hereden de `Exception` o sus subclases. En Python, también se pueden crear excepciones personalizadas mediante la creación de clases que hereden de `Exception` o sus subclases.

Similitudes

- **Jerarquía de excepciones** Tanto Python como Java tienen una jerarquía de clases de excepción, donde las excepciones más específicas se derivan de clases base más generales. Esto permite capturar y manejar excepciones de manera más granular según sea necesario.
- **Uso de bloques de manejo de excepciones** Tanto en Python como en Java, se utilizan bloques de manejo de excepciones para rodear el código propenso a generar excepciones. Los bloques `try-except` en Python y `try-catch` en Java permiten capturar excepciones y proporcionar un código de manejo para responder adecuadamente.
- **Uso de bloques `finally`** Tanto en Python como en Java, se puede utilizar un bloque `finally` opcionalmente después de los bloques `try-except` o `try-catch`. El bloque `finally` se ejecuta siempre, independientemente de si se lanza o maneja una excepción, lo que permite realizar acciones de limpieza o liberación de recursos necesarios.

En general, tanto Python como Java ofrecen características robustas para el manejo de excepciones, aunque difieren en algunos aspectos de sintaxis y enfoque.

Ejercicio 10

¿Qué modelo de excepciones implementa Ruby?.

Ruby implementa un modelo de excepciones similar al modelo de reasunción. A diferencia del modelo de terminación que se encuentra en lenguajes como Java, Ruby utiliza una combinación de manejo de excepciones y continuación para controlar el flujo de ejecución cuando se produce una excepción.

¿Qué instrucciones específicas provee el lenguaje para manejo de excepciones y cómo se comportan cada una de ellas?

Ruby proporciona las siguientes instrucciones específicas para el manejo de excepciones:

1. `begin rescue end` Este bloque se utiliza para envolver el código que puede generar una excepción. El bloque `rescue` captura una excepción específica o una lista de excepciones y proporciona un código de manejo para responder a la excepción. Pueden haber varios bloques `rescue` para manejar diferentes tipos de excepciones. El bloque `rescue` se ejecuta cuando se produce una excepción y coincide con el tipo especificado.

   ```
   begin
       # Código que puede lanzar una excepción
   rescue TipoDeExcepcion
       # Código de manejo para el tipo de excepción especificado
   end
   ```

2. `raise` Esta instrucción se utiliza para lanzar manualmente una excepción en cualquier parte del código. Puede especificar el tipo de excepción o simplemente usar `raise` para relanzar la

excepción actualmente activa.

```
raise TipoDeExcepcion, "Mensaje de error opcional"
```

3. `ensure` El bloque `ensure` se utiliza para especificar un código que siempre se ejecuta, ya sea que ocurra una excepción o no. Es útil para realizar acciones de limpieza o liberación de recursos, independientemente de si se produjo una excepción o no.

```
begin
    # Código que puede lanzar una excepción
rescue TipoDeExcepcion
    # Código de manejo para el tipo de excepción especificado
ensure
    # Código que se ejecuta siempre, independientemente de las excepciones
end
```

Además de estas instrucciones, Ruby también proporciona una clase base llamada `Exception` de la cual se derivan todas las excepciones en Ruby. Esto permite la creación de excepciones personalizadas mediante la definición de nuevas clases que hereden de `Exception` o sus subclases.

En resumen, Ruby implementa un modelo de excepciones basado en el manejo y la continuación, proporcionando bloques `begin` - `rescue` - `end` para capturar y manejar excepciones, `raise` para lanzar manualmente excepciones y `ensure` para ejecutar código de limpieza.

Ejercicio 11

Indique el mecanismo de excepciones de javascript.

El mecanismo de excepciones en JavaScript se basa en el uso de bloques `try-catch-finally` para capturar y manejar las excepciones. Aquí tienes una descripción de cada parte del mecanismo:

- 1. `try` : El bloque `try` se utiliza para envolver el código que puede generar una excepción. Dentro del bloque `try` , se coloca el código que se desea ejecutar y que puede potencialmente lanzar una excepción.

```
try {
  // Código que puede lanzar una excepción
}
```

- 2. `catch` : El bloque `catch` se utiliza para capturar una excepción específica y proporcionar un código de manejo para responder a la excepción capturada. Dentro del bloque `catch` , se especifica el tipo de excepción que se desea capturar y se asigna a una variable que se puede utilizar para obtener información adicional sobre la excepción.

```
try {
    // Código que puede lanzar una excepción
} catch (excepcion) {
    // Código de manejo para la excepción capturada
}
```

- 3. `finally` : El bloque `finally` es opcional y se utiliza para especificar un código que siempre se ejecutará, independientemente de si se produjo una excepción o no. El bloque `finally` se ejecuta incluso si se captura una excepción en el bloque `catch` o si se utiliza la instrucción `return` para salir del bloque `try` .

```
try {
  // Código que puede lanzar una excepción
} catch (excepcion) {
  // Código de manejo para la excepción capturada
} finally {
  // Código que se ejecuta siempre, independientemente de las excepciones
}
```

- 4. `throw` : La instrucción `throw` se utiliza para lanzar manualmente una excepción en cualquier parte del código. Puedes lanzar una instancia de la clase `Error` o cualquier otra clase derivada de `Error` para representar una excepción específica.

```
throw new Error("Mensaje de error");
```

Cuando se produce una excepción dentro del bloque `try` , el flujo de ejecución salta al bloque `catch` correspondiente que maneja la excepción. Si no se encuentra un bloque `catch` que pueda manejar la excepción lanzada, la excepción se propaga hacia arriba en la pila de llamadas hasta que se encuentre un bloque `catch` adecuado o hasta el punto de entrada del programa.

El bloque `finally` se ejecuta siempre, ya sea que se haya producido una excepción o no, y es útil para realizar acciones de limpieza o liberación de recursos.

Es importante tener en cuenta que JavaScript también proporciona varios tipos predefinidos de excepciones, como `Error` , `SyntaxError` , `TypeError` , entre otros, que se pueden lanzar y capturar para manejar situaciones de error específicas.

Ejercicio 12

Sea el siguiente programa escrito en PYTHON: Indique el camino de ejecución

```
#!/usr/bin/env python
#calc.py

def dividir():
  x = a / b
  print (("Resultado"), (x))

while True:
  try:
    a = int(input("Ingresa el primer numero: \n"))
    b = int(input("Ingresa el segundo numero: \n"))
    dividir()
    break
  except ZeroDivisionError:
    print ("No se permite dividir por cero")
  finally:
    print ("Vuelve a probar"
```

a) Describa qué caminos ejecuta para diferentes valores de ingreso

- Valores: `10` y `2`. Se ejecuta normal y se imprime: “Resultado 5” y “vuelve a probar”.
- Valores: `10` y `0`. Se ejecuta hasta la excepción y se imprime: “No se permite dividir por cero” y “vuelve a probar”.

Y creo que no hay otro caro porque el más importante es el de la división por 0, lo demás todo bien :p

b) Agregar el uso de una excepción anónima

Para agregar el uso de una excepción anónima (también conocida como excepción genérica) en el código se puede utilizar la clase **`Exception`** sin especificar ningún tipo específico de excepción. Esto capturará cualquier excepción que se produzca en el bloque **`try`** y ejecutará el código de manejo correspondiente.

```python
#!/usr/bin/env python

def dividir():
    x = a / b
    print("Resultado:", x)

while True:
    try:
        a = int(input("Ingresa el primer numero: \n"))
        b = int(input("Ingresa el segundo numero: \n"))
        dividir()
        break
    except ZeroDivisionError:
        print("No se permite dividir por cero")
    except Exception:
        print("Se produjo una excepción")
    finally:
        print("Vuelve a probar")
```

En este código, se agregó un nuevo bloque **`except Exception`** después del bloque **`except ZeroDivisionError`** . Esto capturará cualquier excepción que no sea una **`ZeroDivisionError`** específica y ejecutará el código de manejo correspondiente, que en este caso simplemente imprime "Se produjo una excepción".

Ejercicio 13

Sea el siguiente código escrito en JAVA

```
public class ExcepcionUno extends Exception {

  public ExcepcionUno(){
    super(); // constructor por defecto de Exception
  }

  public ExcepcionUno( String cadena ){
    super( cadena ); // constructor param. de Exception
  }
}

public class ExcepcionDos extends Exception {
  public ExcepcionDos(){
    super(); // constructor por defecto de Exception
  }
  public ExcepcionDos( String cadena ){
    super( cadena ); // constructor param. de Exception
  }
}

public class ExcepcionTres extends Exception {
  public ExcepcionTres(){
    super(); // constructor por defecto de Exception
  }
  public ExcepcionTres( String cadena ){
    super( cadena ); // constructor param. de Exception
  }
}

public class Lanzadora {
  public void lanzaSiNegativo(int param) throws ExcepcionUno {
    if (param < 0)
      throw new ExcepcionUno("Numero negativo");
  }
  public void lanzaSimayor100(int param) throws ExcepcionDos {
    if (param >100 and param<125)
      throw new ExcepcionDos("Numero mayor100");
  }
  public void lanzaSimayor125(int param) throws ExcepcionTres {
    if (param >= 125)
      throw new ExcepcionTres("Numero mayor125");
  }
}

import java.io.FileInputStream;
import java.io.IOException;

public class Excepciones {
  public static void main(String[] args) {
    // Para leer un fichero
    Lanzadora lanza = new Lanzadora();
    FileInputStream entrada = null;
    int leo;
    try {
      entrada = new FileInputStream("fich.txt");
      while ((leo = entrada.read()) != -1){
```

```
          if (leo < 0)
            lanza.lanzaSiNegativo(leo);
          else if (leo > 100)
            lanza.lanzaSimayor100(leo);
        }
        entrada.close();
        System.out.println("Todo fue bien");
      }
      catch (ExcepcionUno e) { // Personalizada
        System.out.println("Excepcion: " + e.getMessage());
      }
      catch (ExcepcionDos e) { // Personalizada
        System.out.println("Excepcion: " + e.getMessage());
      }
      catch (IOException e) { // Estándar
        System.out.println("Excepcion: " + e.getMessage());
      }
      finally {
        if (entrada != null)
          try {
            entrada.close(); // Siempre queda cerrado
          }
          catch (Exception e) {
            System.out.println("Excepcion: " + e.getMessage());
          }
        System.out.println("Fichero cerrado.");
      }
    }
  }
```

a) Indique cómo se ejecuta el código. Debe quedar en claro los caminos posibles de ejecución, cuales son los manejadores que se ejecutan y cómo se buscan los mismos y si en algún caso se produce algún error

Se definen tres clases de excepciones personalizadas: `ExcepcionUno`, `ExcepcionDos` y `ExcepcionTres`. Cada una de ellas extiende la clase base `Exception` y tiene constructores para recibir una cadena de mensaje de error.

Se define la clase `Lanzadora`, que contiene tres métodos: `lanzaSiNegativo`, `lanzaSimayor100` y `lanzaSimayor125`. Cada método lanza una excepción personalizada según una condición específica.

En la clase `Excepciones`, se crea una instancia de `Lanzadora` y se declara un objeto `FileInputStream` llamado `entrada` para leer un archivo.

Dentro del bloque `try`, se utiliza un bucle `while` para leer el contenido del archivo y se aplican algunas condiciones. Si se cumple alguna de las condiciones, se llama al método correspondiente de la instancia de `Lanzadora` para lanzar la excepción correspondiente.

Si ocurre alguna excepción, se captura en los bloques `catch` correspondientes según el tipo de excepción. Primero se capturan las excepciones personalizadas (`ExcepcionUno` y `ExcepcionDos`) y luego se captura una excepción estándar (`IOException`) para manejar errores de E/S.

Dentro del bloque `finally`, se asegura de que el archivo se cierre correctamente utilizando el método `close()` del objeto `entrada`, incluso si se produjo una excepción o no. Si se produce una excepción al intentar cerrar el archivo, se captura y se muestra un mensaje de error.

Finalmente, se imprime "Fichero cerrado" para indicar que el proceso ha finalizado.

Los caminos posibles de ejecución y los manejadores que se ejecutan son los siguientes:

- Si no se produce ninguna excepción en el bloque `try`, se ejecutará el código dentro del bloque `finally` para cerrar el archivo y se imprimirá "Todo fue bien".
- Si se produce una excepción `ExcepcionUno`, se capturará en el primer bloque `catch`, se mostrará el mensaje de la excepción y se ejecutará el código dentro del bloque `finally`.
- Si se produce una excepción `ExcepcionDos`, se capturará en el segundo bloque `catch`, se mostrará el mensaje de la excepción y se ejecutará el código dentro del bloque `finally`.
- Si se produce una excepción `IOException`, se capturará en el tercer bloque `catch`, se mostrará el mensaje de la excepción y se ejecutará el código dentro del bloque `finally`.
- Si se produce una excepción al intentar cerrar el archivo en el bloque `finally`, se capturará y se mostrará el mensaje de error correspondiente.

En caso de que ocurra algún error, se mostrará el mensaje de error correspondiente en el bloque `catch` o en el bloque `finally` si se produce una excepción al intentar cerrar el archivo.

Es importante destacar que el código proporcionado no maneja excepciones específicas para el caso de una condición en la que `leo` sea mayor a 125, lo cual podría generar un comportamiento inesperado.

Ejercicio 14

Dado el siguiente código en Java. Indique todos los posibles caminos de resolución, de acuerdo a los números que vaya leyendo del archivo.

```
class ExcepcionE1 extends Exception {

  publicExcepcionE1(){
    super(); // constructor por defecto de Exception
  }

  publicExcepcionE1( String cadena ){
    super( cadena ); // constructor param. de Exception
  }
}

class ExcepcionE2 extends Exception {

  PublicExcepcionE2(){
    super(); // constructor por defecto de Exception
  }

  PublicExcepcionE2( String cadena ){
    super( cadena ); // constructor param. de Exception
  }
}

// Esta clase lanzará la excepción
public class Evaluacion {
  void Evalua( int edad ) throws ExcepcionE1, ExcepcionE2 {
    if ( edad < 18 )
      throw new ExcepcionE1( "Es una persona menor de edad" );
    else if ( edad> 70 )
      throw new ExcepcionE2( "Es persona mayor de edad" );
  }

  void Segmenta( int edad ) throws ExcepcionE1, ExcepcionE2 {
    if ( edad < 35 )
      throw new ExcepcionE1( "Es una persona joven" );
  }
}

class AnalisisEdadPoblacion{
  public static void main( String[] args ){
  // Para leer un fichero
  Evaluacion Invoca = new Evaluacion();
  FileInputSt ream entrada = null;
  int leo;
  try{
    entrada = new FileInputStream( "fich.txt" );
    while ( ( leo = entrada.read() ) != -1 ) {
      try {
        if (leo<0) {
          throw new ExcepcionE1( "Edad inválida" );
        }
        else{
          if (leo>120){
            throw new ExcepcionE1( "Edad inválida" );
          }
        }
        invoca.evalua (leo);
```

```
          invoca.segmenta( leo );
          System.out.println( " ( Es persona adulta, Todo fue bien" );
        }
        catch ( ExcepcionE2 e ){
          System.out.println( "Excepcion: " + e.getMessage() );
        }
        catch ( ExcepcionE1 e ){
          System.out.println( "Excepcion: " + e.getMessage() );
        }
      }
    }
    catch (FileNotFoundException e1) {
      System.out.println("No se encontró el archivo");
    }
    catch (IOException e) {
      System.out.println("Problema para leer los datos");
    }
    finally {
      if (entrada != null)
        try {
          entrada.close();
        } catch (Exception e) {
          System.out.println("Excepcion: " + e.getMessage());
        }
      System.out.println("Fichero cerrado.");
      }
    }
}
```

El código presenta los siguientes caminos de ejecución:

- 1. Si se produce una excepción **`FileNotFoundException`** al intentar abrir el archivo, se capturará en el bloque **`catch`** correspondiente y se mostrará el mensaje "No se encontró el archivo".
- 2. Si se produce una excepción **`IOException`** al leer los datos del archivo, se capturará en el bloque **`catch`** correspondiente y se mostrará el mensaje "Problema para leer los datos".
- 3. Si se produce una excepción **`ExcepcionE1`** o **`ExcepcionE2`** durante la ejecución del bucle **`while`** , se capturará en los bloques **`catch`** correspondientes y se mostrará el mensaje de la excepción "Edad inválida".
- 4. Si no se produce ninguna excepción durante la ejecución del bucle **`while`** , se ejecutarán los métodos **`evalua()`** y **`segmenta()`** de la instancia **`Invoca`** de la clase **`Evaluacion`** .
  - Si se lanza una excepción **`ExcepcionE1`** en el método **`evalua()`** , se capturará en el bloque **`catch`** correspondiente y se mostrará el mensaje de la excepción "Es una persona menor de edad".
  - Si se lanza una excepción **`ExcepcionE2`** en el método **`evalua()`** , se capturará en el bloque **`catch`** correspondiente y se mostrará el mensaje de la excepción "Es persona mayor de edad".
  - Si se lanza una excepción **`ExcepcionE1`** en el método **`segmenta()`** , se capturará en el bloque **`catch`** correspondiente y se mostrará el mensaje de la excepción "Es una persona joven".

- 5. Después de ejecutar los métodos `evalua()` y `segmenta()`, se imprimirá el mensaje "Es persona adulta, Todo fue bien".

Finalmente, en el bloque `finally`, se cerrará el archivo y se imprimirá "Fichero cerrado". Si se produce una excepción al intentar cerrar el archivo, se capturará en el bloque `catch` correspondiente y se mostrará el mensaje de la excepción.

En resumen, los posibles caminos de resolución dependen de las excepciones que se lancen durante la ejecución del programa, como `FileNotFoundException`, `IOException`, `ExcepcionE1` y `ExcepcionE2`. Cada excepción se captura en un bloque `catch` específico y se muestra un mensaje correspondiente.