Área de Ingeniería Telemática
Departamento de Ingeniería de Sistemas y Automática
Escuela Superior de Ingenieros
Universidad de Sevilla

EXCEPCIONES

1. OBJETIVO

El objetivo de esta práctica es mostrar la gestión de condiciones excepcionales en los lenguajes de programación orientados a objeto mediante el uso de excepciones, y de forma específica el lenguaje de programación Java. Para ello se muestra qué es una excepción, cómo se captura y cómo se lanza desde un programa. Esta práctica presenta cómo crear excepciones propias de la aplicación, y se muestra la forma de tratarlas en el depurador `jdb`.

2. DESCRIPCIÓN

Las excepciones son situaciones anormales que suceden cuando se provoca un fallo durante la ejecución de un programa. En los lenguajes de programación que no tienen gestión de excepciones, los errores se gestionan normalmente utilizando códigos de error. La gestión de excepciones de Java es una solución más directa para la gestión de errores en tiempo de ejecución y con un enfoque orientado a objetos.

Por ejemplo, se producen excepciones en un programa cuando se hace una división de un entero por cero, o cuando se intenta acceder al elemento de la posición 4 de un vector cuando el vector sólo tiene 2 elementos.

Las excepciones en Java son *objetos* que describen una condición excepcional, es decir, un error que se ha producido en un fragmento de código. Cuando surge una condición excepcional, se crea un objeto que representa la excepción y se *lanza* la excepción avisando de que se ha producido una situación especial.

El método que recibe la excepción puede elegir *capturarla* y gestionarla él mismo o no. En algún punto del programa, la excepción es capturada y procesada. Las excepciones pueden ser generadas por el **intérprete** de Java o pueden ser generadas **por el propio código**.

Excepciones no capturadas

Antes de aprender cómo se gestionan las excepciones en los programas en Java, es útil ver qué ocurre cuando no se gestionan. Los errores de ejecución pueden presentarse en cualquier lenguaje de programación. Empecemos por un primer ejemplo en el lenguaje de programación C. Descargue el código de la plataforma virtual para realizar la práctica 4 y descomprímalo.
Se proporciona el siguiente código que intencionadamente incluye un error de división por cero.

```c
#include <stdio.h>

int main() {
     int d = 0;
     int a = 0;

      a = 42 /d;
      printf("Este codigo no se ejecuta nunca\n");

      return 0;
}
```

Practica4Ejercicio00.c

Utilice la herramienta `make` para compilar y ejecutar el código de la siguiente forma y observe el resultado:

`make -f makeP4 compila_c00`

`make -f makeP4 ejecuta_c00`

Este mismo ejemplo se puede realizar sobre el lenguaje de programación Java.

```java
package fp2.poo.practica4;

public class Practica4Ejercicio01 {
    public static void main( String args[] ) {
        int d = 0;
        int a = 42 /d;
        System.out.println( "Este codigo nunca se ejecuta" );
    }
}
```

Practica4Ejercicio01.java

Utilice la herramienta `make` para compilar los ejemplos que se van a ver a continuación, de la siguiente forma:

`make -f makeP4 compilaJava`

y ejecute el código de este primer ejemplo en Java de la siguiente forma:

`make -f makeP4 ejecuta01`

Nota: a partir de ahora cada ejecución de los ejemplos se realizará mediante **`make -f makeP4 ejecutaXX`** con **`XX`** que indica el número del ejercicio que se está probando.

Cuando el intérprete de Java detecta el intento de dividir por cero, construye un nuevo objeto del tipo `Exception` y *lanza* la excepción. Esto hace que se detenga la ejecución del programa, ya que una vez que se ha lanzado una excepción, tiene que ser *capturada* por el gestor de excepciones y tratada inmediatamente. En este ejemplo, no hemos proporcionado un *gestor de excepciones propio,* por lo que la excepción será capturada por el gestor proporcionado por el

intérprete de Java. Cualquier excepción que no es capturada por el programa será tratada por el gestor por defecto, quien muestra un mensaje describiendo la excepción, imprime el trazado de la pila del lugar donde se produjo la excepción y termina el programa.

Esta es la salida generada cuando este ejemplo se ejecuta en el intérprete de Java.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at fp2.poo.practica4.Practica4Ejercicio01.main(Practica4Ejercicio01.java:22)
```

Observe que en el trazado de la pila se incluye:

- el nombre de la clase, **`fp2.poo.practica4.Practica4Ejercicio01`**;
- el nombre del método, **`main()`**;
- el nombre del archivo **`Practica4Ejercicio01.java`**,
- y el número de la línea, **`22.`**

Observe también que el tipo de la excepción que se ha lanzado es una Exception llamada **`AritmeticException`**, que describe más específicamente el tipo de error que se ha producido. Java proporciona distintos tipos de excepciones que coinciden con las clases de errores que se pueden producir durante la ejecución.

La traza de la pila siempre muestra la secuencia de llamadas a métodos en el momento del error. Esta es otra versión del programa anterior que introduce el mismo error pero en un método distinto del **`main().`**

```
package fp2.poo.practica4;

public class Practica4Ejercicio02 {

    static void metodo() {
        int d = 0;
        int a = 10/d;

        System.out.println("Este codigo nunca se ejecuta");
    }

    public static void main ( String args[] ) {
        Practica4Ejercicio02.metodo();

        System.out.println( "Este codigo nunca se ejecuta" );
    }
}
```

Practica4Ejercicio02.java

Ejecute el programa y observe la traza de la pila que en este caso incluye el método estático. La pila de llamadas es bastante útil para depurar ya que muestra los pasos que conducen al error.

Dado el siguiente ejemplo

```
package fp2.poo.practica4;

public class Practica4Ejercicio03 {
    public static void main(String[] args) {
        String str = null;

        str.length();
        System.out.println("Este codigo nunca se ejecuta");
    }
}
```

Practica4Ejercicio03.java

Ejecute el programa, e indique qué excepción se produce.

Captura de excepciones

Aunque el gestor de excepciones proporcionado por el intérprete de Java es útil para depurar, normalmente se deseará gestionar las excepciones que se produzcan en los programas, obteniendo así dos ventajas, por un lado, permite determinar el error y, por otro, evita que el programa termine automáticamente. La mayoría de los usuarios estarían cuando menos confusos, si el programa terminara su ejecución e imprimiese una traza de la pila cuando se produjese un error. Afortunadamente, es bastante fácil evitar esto.

Para protegerse de los errores de ejecución y poder gestionarlos, es necesario escribir el código que se quiere controlar dentro de un bloque **`try`**. Inmediatamente después del bloque **`try`** es necesario incluir la cláusula **`catch`** que especifica el tipo de excepción que se desea capturar.

Esta es la forma general de un bloque de gestión de excepciones:

```
try {

  //bloque de código

} catch (TipoExcepción1 exOb) {

  //gestor de excepciones para TipoExcepción1

} catch (TipoExcepción2 exOb) {

  //gestor de excepciones para TipoExcepción2

} finally {

   //bloque de código que se ejecutará antes de que termine
el bloque try

}
```

Tipos de excepciones

Todos los tipos de excepción son subclases de la clase **`Throwable`** (lanzable). Esta clase está en la parte superior de la jerarquía de clases de excepción. Inmediatamente debajo de **`Throwable`** hay dos subclases que dividen las excepciones en dos ramas distintas.

1. Una rama está encabezada por la clase **`Exception`**. Esta clase se utiliza para condiciones excepcionales que los programas de usuario deberían capturar. Esta es la clase de partida de las subclases que utilizaremos para crear nuestros propios tipos de excepción. Una subclase importante de **`Exception`** es **`RuntimeException`** (excepción durante la ejecución). Las excepciones de este tipo son definidas automáticamente por los programas e incluyen cosas del estilo de la división por cero o un índice inválido de una matriz.

2. La otra rama del árbol es la clase **`Error`**, que define las excepciones que no se suelen capturar en los programas en condiciones normales. Un ejemplo de este tipo de errores puede ser el desbordamiento de una pila. Este capítulo no trata las excepciones de tipo **`Error`**, porque se crean habitualmente como respuesta a fallos catastróficos que no suelen ser gestionados por los programas.

El siguiente programa incluye un bloque **`try`** y una cláusula **`catch`** que procesa la excepción **`AritmeticException`** generada por un error de división por cero.

```
package fp2.poo.practica4;

import java.lang.ArithmeticException;

public class Practica4Ejercicio04 {
    public static void main (String args[]){
        int d, a;

        try {
            d = 0;
            a = 42/d;
            System.out.println("Este codigo nunca se ejecuta");
        } catch ( ArithmeticException e ){
            System.out.println( "División por cero." + e );
        }
        System.out.println( "Después de catch." );
    }
}
```

Practica4Ejercicio04.java

Ejecute el programa y observe la salida.

Observe que la llamada a **`println()`** dentro del bloque **`try`** no se ejecuta. Una vez que se ha lanzado la excepción, el control del programa se transfiere desde el bloque **`try`** al bloque **`catch`**. Una vez se ha ejecutado la sentencia **`catch`**, el control del programa continúa en la siguiente línea del programa que hay después del bloque **`try`** / **`catch`**.

Las sentencias **try** y **catch** forman una unidad. El ámbito de la cláusula **catch** está restringido a aquellas sentencias especificadas en la sentencia **try** que la precede. Una sentencia **catch** no puede capturar una excepción lanzada por otras sentencia **try**, excepto en el caso de sentencias **try** anidadas. Las sentencias que están protegidas por **try** tienen que estar entre llaves, es decir, deben estar dentro de un bloque.

Una sentencia **catch** bien diseñada debería resolver la condición de excepción y continuar como si el error nunca se hubiese producido. Por ejemplo, en el siguiente programa cada iteración del bucle **for** obtiene dos enteros de forma aleatoria; divide uno entre otro y el resultado se utiliza para dividir el valor 12.345. El resultado final se guarda en **a**. Si cualquiera de las operaciones de división provoca un error de división por cero, este error será capturado y se asignará a **a** el valor cero, continuando la ejecución del programa.

```
package fp2.poo.practica4;

import java.util.Random;
import java.lang.ArithmeticException;

public class Practica4Ejercicio05 {
    public static void main (String args[]){
        int  a =0, b =0, c =0;
        Random r = new Random();

        for ( int i = 0; i < 2000; i++ ){
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345/(b/c);
            } catch ( ArithmeticException e ){
                System.out.println("División por cero."  + e);
                a = 0; //asigna cero y continua
            }
            System.out.println( "a: " + a );
        }
    }
}
```

Practica4Ejercicio05.java

Ejecute el programa para ver el resultado. Nótese que si el valor de c es mayor que el de b el resultado de la división entera será cero, y en este caso se lanzará una excepción del tipo `ArithmeticException`.

Descripción de una excepción

La clase **`Throwable`** sobrescribe el método **`toString()`** definido por la clase **`Object`**, devolviendo una cadena que contiene la descripción de una excepción. Se puede mostrar esta descripción simplemente pasando la excepción como argumento de una sentencia **`println()`**. Por ejemplo, el bloque **`catch`** del ejemplo anterior se puede escribir de la siguiente forma:

```
} catch (ArithmeticException e){
      System.out.println("Excepcion: " + e);
      a = 0; //asigna cero y continua
}
```

Realice la modificación propuesta al ejemplo y pruébela.

Al realizar esta modificación, cada error de división por cero presentará el siguiente mensaje:

`Excepción: java.lang.ArithmeticException: / by zero`

Aunque en este contexto esto no tiene un gran valor, la capacidad de mostrar la descripción de una excepción puede ser muy valiosa en otras circunstancias, especialmente cuando estamos experimentando con excepciones o cuando estamos depurando.

Manejando varios tipos de excepciones

En algunos casos, la misma secuencia de código puede activar más de un tipo de excepción. Para gestionar este tipo de situaciones, se pueden especificar dos o más cláusulas **catch**, cada una para capturar un tipo distinto de excepción. Cuando se lanza una excepción, se inspeccionan por orden las sentencias **catch** y se ejecuta la primera que coincide con el tipo de excepción que se ha producido. Después de ejecutar la sentencia **catch**, no se ejecuta ninguna de las restantes, continuando la ejecución al final del bloque **try/catch**. El siguiente programa gestiona dos tipos de excepción diferente (división por cero - `ArithmeticException` – e índice fuera de límites - `ArrayIndexOutOfBoundsException` -):

```
package fp2.poo.practica4;

import java.util.Random;

public class Practica4Ejercicio06 {
    public static void main (String args[]){
        try {
            int  a = 0;

            a = args.length;
            System.out.println( "a: " + a );
            int b = 0;
            b = 42 / a;
            int c[] = {1};
            c[42] = 99;
        } catch(ArithmeticException e){
            System.out.println( "División por cero. " + e );
        } catch( ArrayIndexOutOfBoundsException e ){
            System.out.println( "Indice fuera de limites. " + e );
        }
       System.out.println( "Después del bloque try/catch. " );
    }
}
```

Practica4Ejercicio06.java

Este programa provoca una división por cero si se ejecuta sin parámetros en la línea de órdenes, ya que **a** es igual a cero. No se producirá este error si se pasa un argumento en la línea de órdenes, ya que **a** tendría un valor mayor que cero. Sin embargo, se producirá una excepción `ArrayIndexOutOfBoundsException` , ya que la matriz de enteros **c** tiene longitud 1 y el programa intenta asignar un valor a **c[42]**.

Pruebe el ejemplo de la siguiente forma y observe la salida generada.

make -f makeP4 ejecuta06a
make -f makeP4 ejecuta06b

Cuando se utilizan múltiples sentencias **`catch`** es importante recordar que las subclases de excepción deben estar antes que la superclase. Esto es así porque la sentencia **`catch`** que utiliza una superclase capturará las excepciones de sus subclases y, por tanto, éstas no se ejecutarán si están después de la superclase. Además, en Java se producirá un error de código no alcanzable.

```
package fp2.poo.practica4;

/**
 *  Este programa contiene un error
 *  Una subclase debe estar antes que su superclase
 *  en una serie de sentencias catch. Si no es así,
 *  existirá código que no es alcanzable y se producirá
 *  un error en tiempo de compilación.
 */
public class Practica4Ejercicio07 {
    public static void main (String args[]){
        try {
            int  a = 0;
            int  b = 42/a;
            System.out.println("Este codigo nunca se ejecuta");
        } catch ( Exception e ){
            System.out.println("Sentencia catch para cualquier excepcion " + e);
        } catch ( ArithmeticException e ){ //ERROR
            /*
             * Esta sentencia catch nunca se ejecutará ya que la
             * ArithmeticException es una subclase de Exception
             */
            System.out.println( "Esto nunca se alcanza " );
        }
    }
}
```

Practica4Ejercicio07.java

Si intenta compilar este programa se producirá un mensaje de error indicando que la segunda sentencia **`catch`** no es alcanzable. Como **`ArithmeticException`** es una subclase de **`Exception`**, la primera sentencia **`catch`** gestionará todos los errores de **`Exception,`** incluidos los de **`ArithmeticException.`** Esto significa que la segunda sentencia **`catch`** nunca se ejecutará.

Compile el ejemplo de la siguiente forma:

make -f makeP4 compila07

Para eliminar este problema, es necesario invertir el orden de las sentencias **`catch`**.
Arregle el ejemplo de la forma indicada para que no presente el problema mencionado.

Sentencias try anidadas

Las sentencias **`try`** pueden estar anidadas, es decir, una sentencia **`try`** puede estar dentro del bloque de otra sentencia **`try`**.

Si en un bloque try se produce una excepción, se busca entre las sentencias catch correspondientes a este try, aquella que trate la excepción que se ha producido. En caso de no encontrarla, se busca en el bloque try inmediatamente superior. Esto continúa hasta que se encuentra la sentencia **`catch`** adecuada o se agotan las sentencias **`try`** anidadas. Si no se encuentra la sentencia **`catch`** buscada, el intérprete de Java gestionará esta excepción.

Este es un ejemplo que utiliza sentencias **`try`** anidadas.

```
package fp2.poo.practica4;

class Practica4Ejercicio08 {
    public static void main (String args[]) {
        try {
            int  a = args.length;
            /*
             * Si no hay ningún argumento en la línea de comandos
             * la siguiente sentencia genera un excepción. Div por 0
             */
            int b = 42/a;
            System.out.println( "a = " + a );

            try {
                /*
                 * bloque try anidado.
                 *
                 * Si se utiliza un argumento en la línea de comandos,
                 * la siguiente sentencia genera una excepc. Div. 0
                 */
                if (a == 1)
                    a = a / ( a - a );
                /*
                 * Si se le pasan dos argumento en Linea de Comandos
                 */
                if ( a == 2 ) {
                    int c[] = { 1 };
                    c[42] = 99; //genera excep. fuera de limites
                }
            } catch ( ArrayIndexOutOfBoundsException e ) {
                System.out.println("Índice fuera de limites. " + e);
            }
        } catch ( ArithmeticException e ){
            System.out.println( "División por cero: " + e );
        }
    }
}
```

Practica4Ejercicio08.java

Ejecute el programa anterior sin ningún argumento (**`ejecuta08a`**), con uno (**`ejecuta08b`**) y con dos argumentos (**`ejecuta08c`**) en línea de comandos:

```
make -f makeP4  ejecuta08a
make -f makeP4  ejecuta08b
make -f makeP4  ejecuta08c
```

Como se puede ver, este programa anida un bloque **try** dentro de otro. El programa funciona de la siguiente manera:

1. Cuando se ejecuta el programa sin argumentos en la línea de órdenes, se genera una excepción de división por cero en el bloque **try** externo.

2. La ejecución del programa con un argumento en la línea de órdenes genera una excepción de división por cero en el bloque **try** interno. Como el bloque interno no captura esta excepción, se pasa al bloque externo, donde es gestionada.

3. Si se ejecuta el programa con dos argumentos, se produce una excepción en el bloque **try** interno al sobrepasar los límites del tamaño de la matriz.

Nota: Las sentencias **try** anidadas pueden aparecer de formas menos obvias que la anterior cuando están involucradas llamadas a métodos. Por ejemplo, dentro de un bloque **try** puede haber una llamada a un método y dentro de ese método puede haber otra sentencia **try**. En este caso, la sentencia **try** que se encuentra dentro del método está anidada con el bloque **try** exterior que llama al método.

Lanzamiento de una excepción

Hasta ahora, sólo se han mencionado las excepciones generadas por el intérprete de Java. Sin embargo, utilizando la sentencia **throw** es posible hacer que el programa lance una excepción de manera explícita. La forma general de la sentencia `throw` es la siguiente:

throw *Instanciathrowable;*

Aquí, *Instanciathrowable* tiene que ser un objeto de tipo **Throwable** o una subclase de Throwable. Los tipos simples, como int o char, y las clases que no son **Throwable,** como String y Object no se pueden utilizar como excepciones. Hay dos formas de obtener un objeto **Throwable**:

1. Utilizando el parámetro de la cláusula **catch**,
2. Creando un objeto del tipo **Throwable** con el operador **new**.

El flujo de la ejecución se detiene inmediatamente después de la sentencia **throw** y cualquier sentencia posterior no se ejecuta. Se inspecciona el bloque **try** más cercano que la engloba para ver si tiene una sentencia **catch** cuyo tipo coincida con el de la excepción. Si la encuentra, el control se transfiere a esa sentencia. Si no, se inspecciona el siguiente bloque **try** que la engloba y así sucesivamente. Si no encuentra ninguna sentencia **catch** cuyo tipo coincida con el de la excepción, entonces el gestor de excepciones por defecto detiene el programa e imprime el trazado de la pila.

A continuación vemos un ejemplo que crea y lanza una excepción. El gestor que captura la excepción la vuelve a lanzar al gestor más externo:

```
ackage fp2.poo.practica4;

class Practica4Ejercicio09 {

    /*
     * Las sentencias try pueden estar implícitamente anidadas
     * a través de llamadas a métodos
     */
    static void metodo (){
        try {
            throw new NullPointerException( "demo" );
        } catch ( NullPointerException e ) {
            System.out.println( "Captura dentro de demo" + e );
            /*
             * Vuelve a lanzar la excepción capturada
             */
            throw e;
        }
    }

    public static void main (String args[]){
        try {
            metodo();
        } catch ( NullPointerException e ){
            System.out.println("Nueva captura: " + e);
        }
    }
}
```

Practica4Ejercicio09.java

Este programa tiene dos oportunidades de tratar el mismo error. Primero, **`main()`** establece un contexto de excepción y después llama a **`metodo ()`**. El método **`metodo ()`** establece después otro contexto de gestión de excepciones y lanza inmediatamente una nueva instancia de **`NullPointerException`**, que es capturada en la sentencia **`catch`** de la siguiente línea. Entonces se vuelve a lanzar la excepción.

El programa también muestra cómo crear un objeto de las excepciones estándares de Java. Preste especial atención a la línea siguiente:

```
throw new NullPointerException("demo");
```

Aquí, el operador **new** se utiliza para construir una instancia de **`NullPointerException`**. Todas las excepciones que están en el núcleo de Java tienen dos constructores:

1. Uno sin parámetros y
2. Otro que tiene un parámetro de tipo cadena.

Cuando se utiliza la segunda forma, el argumento especifica una cadena que describe la excepción. Esta cadena se imprime cuando se utiliza este objeto como argumento de **`print()`** o **`println()`**.

Nota: También se puede obtener con una llamada al método **`getMessage()`**, que está definido en **`Throwable`**.

Ejecute el programa para probar el ejemplo y observe el resultado.

Declaración de excepciones en métodos

Si un método es capaz de provocar una excepción que no maneja él mismo, debería especificar este comportamiento para que los métodos que lo llaman puedan protegerse frente a esta excepción.

Esto se puede hacer incluyendo una cláusula **`throws`** en la declaración del método.

Una cláusula **`throws`** lista los tipos de excepciones que un método puede lanzar. Todas las excepciones que un método puede lanzar deben estar declaradas en una cláusula **`throws`** (salvo algunas propias de Java), si no están declaradas se producirá un error de compilación.

Esta es la forma general de la declaración de un método que incluye una cláusula **`throws`**.

```
tipo nombre_del_método(lista_de_parámetros) throws lista_de_excepciones
{
  //cuerpo del método
}
```

Aquí, ***lista_de_excepciones*** es la lista de excepciones, separadas por comas, que un método puede lanzar.

Este es un ejemplo de un programa incorrecto que intenta lanzar una excepción sin tener código para capturarla. Como el programa no utiliza una cláusula **`throws`** para declarar esta circunstancia, el programa no compilará.

```
package fp2.poo.practica4;


class Practica4Ejercicio10 {
    static void metodoDemo (){
        System.out.println( "Dentro de metodoDemo " );
        throw new IllegalAccessException( "metodoDemo " );
    }

    public static void main ( String args[] ){
        metodoDemo ();
    }
}
```

Practica4Ejercicio10.java

Pruebe a compilar el ejemplo de la siguiente forma:

`make -f makeP4 compila10`

Para hacer que se compile este ejemplo hay que realizar dos cambios. En primer lugar, es necesario declarar que `metodoDemo()` lanza una excepción **`IllegalAccessException`**. En segundo lugar, el método **`main()`** tiene que definir una sentencia **`try/catch`** que capture esta excepción.

```
package fp2.poo.practica4;

import java.lang.IllegalAccessException;

class Practica4Ejercicio11 {
    static void metodoDemo () throws IllegalAccessException{
        System.out.println("Dentro de metodoDemo");
            throw new IllegalAccessException("metodoDemo");
    }

    public static void main (String args[]){
        try{
            Practica4Ejercicio11.metodoDemo();
        }catch(IllegalAccessException e){
            System.out.println("Captura "+e);
        }
    }
}
```

Practica4Ejercicio11.java

Ejecute el ejemplo de la siguiente forma:

make -f makeP4 ejecuta11

Bloque de finalización

Cuando se lanzan excepciones, la ejecución de un método sigue un trayecto no lineal bastante brusco que altera el flujo normal del método. Dependiendo de cómo esté codificado el método, incluso es posible que una excepción provoque que el método termine de forma prematura. Esto puede ser un problema en algunos métodos. Por ejemplo, si un método abre un archivo cuando comienza y lo cierra cuando finaliza, entonces no sería deseable que el mecanismo de gestión de la excepción omita la ejecución del código que cierra el archivo. La palabra clave **`finally`** está diseñada para resolver este problema.

`finally` crea un bloque de código que se ejecutará después del bloque **`try/catch`** y antes del siguiente bloque **`try/catch`**. El bloque **`finally`** se ejecutará tanto si se lanza una excepción como si no. Si se lanza una excepción, el bloque **`finally`** se ejecutará incluso aunque no haya ninguna sentencia **`catch`** que capture dicha excepción. Siempre que un método vaya a devolver el control al método llamante desde un bloque **`try/catch`**, mediante una excepción no capturada, o una sentencia **`return`** explícita, se ejecuta la cláusula **`finally`** justo antes del final del método. Esto puede resultar útil para cerrar descriptores de archivo y liberar cualquier otro recurso que se hubiese asignado al comienzo de un método con la intención de liberarlo antes de terminar. La cláusula **`finally`** es opcional. Sin embargo, cada sentencia **`try`** necesita, al menos, una cláusula **`catch`** o **`finally`**.

Este es un ejemplo que muestra tres métodos que terminan de varias maneras, todas ejecutando sus cláusulas **`finally.`**

```
package fp2.poo.practica4;

import java.lang.IllegalAccessException;

public class Practica4Ejercicio12 {

    /*
     * lanza una excepción fuera del método
     */
    static void metodoA () {
        try {
            System.out.println("Dentro de metodoA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("Sentencia finally metodoA");
        }
    }

    /*
     * ejecuta la sentencia return dentro del bloque try
     */
    static void metodoB () {
        try {
            System.out.println("Dentro de metodoB");
            return;
        } finally {
            System.out.println("Sentencia finally metodoB");
        }
    }

    /*
     * ejecuta un bloque try normalmente
     */
    static void metodoC () {
        try {
            System.out.println("Dentro de metodoC");
        } finally {
            System.out.println("Sentencia finally metodoC");
        }
    }

    public static void main (String args[]){
        try {
            metodoA ();
        } catch ( Exception e ) {
            System.out.println( "Excepción Capturada " + e );
        }
        metodoB ();
        metodoC ();
    }
}
```

Practica4Ejercicio12.java

En este ejemplo, en el método **`main`**, se invoca al **`metodoA()`**, al **`metodoB()`** y al **`metodoC()`**. El **`metodoA()`** sale prematuramente de la sentencia **`try`** lanzando una excepción del tipo **`RuntimeException`**, y se ejecuta el bloque **`finally`**. En programa principal se captura la excepción lanzada por el **`metodoA()`** y continúa su ejecución. El **`metodoB()`** sale de la sentencia **`try`** mediante una sentencia **`return`** y se ejecuta de todas formas la cláusula **`finally`** antes de que **`metodoB()`** devuelva el control. En **`metodoC()`**, la

sentencia **`try`** se ejecuta normalmente sin que se produzcan errores, y a pesar de ello, también ejecuta el bloque **`finally`**.

Pruebe el ejemplo y observe el resultado, con la siguiente orden:

```
make -f makeP4 ejecuta12
```

Nota: Si un bloque **`finally`** está asociado a una sentencia **`try`**, el bloque **`finally`** se ejecutará cuando finalice la sentencia **`try`**.

Excepciones del núcleo de Java

Java define algunas clases de excepciones en su paquete estándar **`java.lang`**. Algunas de estas clases aparecían en los ejemplos anteriores. Muchas de estas excepciones son subclases del tipo **`RuntimeException`**. Las excepciones derivadas de **`RuntimeException`** están disponibles automáticamente en todos los programas Java, ya que el paquete **`java.lang`** se importa implícitamente. Además, no es necesario incluir estas excepciones en la lista **`throws`** de ningún método.

En términos de Java, estas excepciones son conocidas con el nombre de *excepciones no comprobadas*, ya que el compilador no comprueba si un método trata o lanza estas excepciones.

Existen otras excepciones definidas en **`java.lang`** que deben ser incluidas en la lista de excepciones que lanza un método si realmente ese método genera una de estas excepciones y no la trata. Éstas excepciones son conocidas como *excepciones comprobadas*.

Java define otros tipos de excepciones en otros paquetes de su biblioteca de clases.

Excepciones propias

Aunque las excepciones del núcleo de Java tratan la mayoría de los errores comunes, es conveniente poder crear tipos propios de excepción que permitan tratar situaciones específicas en las aplicaciones. Para hacer esto, es necesario definir una subclase de **`Exception`**, que es una subclase de **`Throwable`**. Realmente, las excepciones no necesitan implementar nada, simplemente su existencia en el sistema permite utilizarlas como excepciones.

La clase **`Exception`** no define ningún método, simplemente hereda los métodos definidos por **`Throwable`**. Por eso, todas las excepciones, incluidas las creadas por nosotros mismos, tienen los métodos definidos por **`Throwable`**.

El siguiente ejemplo declara una nueva subclase de **`Exception`** y utiliza esta subclase para indicar una condición de error en un método. Esta clase sobrescribe el método **`toString()`** (de la clase **`Object`**), permitiendo que la descripción de la excepción se imprima utilizando **`println()`**.

```
package fp2.poo.practica4;

import java.lang.Exception;

/**
 * Descripcion: Implementacion de una Excepcion.
 *
 * version 1.0 Mayo 2011
 * Fundamentos de Programacion II
 */
public class Practica4Ejercicio13Exception extends Exception{

    private int detalle;

    public Practica4Ejercicio13Exception(int a){
       detalle = a;
    }

    public String toString(){
        return "Practica4Ejercicio13Exception[" + detalle + "]";
    }
}
```

Practica4Ejercicio13Exception.java

```
package fp2.poo.practica4;

import java.lang.Exception;
import fp2.poo.practica4.Practica4Ejercicio13Exception;

public class Practica4Ejercicio13Main {

    static void calcula (int a) throws Practica4Ejercicio13Exception{
        System.out.println("Ejecuta calcula (" + a + ")" );
        if ( a > 10 )
            throw new Practica4Ejercicio13Exception(a);
        System.out.println("Finalizacion normal");
    }

    public static void main (String args[]){
        try {
            calcula (1);
            calcula (20);
        } catch ( Practica4Ejercicio13Exception e ){
            System.out.println( "Excepción Capturada " + e );
        }
    }
}
```

Practica4Ejercicio13Main.java

Este ejemplo define una subclase de **`Exception`** llamada **`Practica4Ejercicio13Exception`**. Esta subclase es bastante simple, ya que sólo tiene un constructor y un método **toString()**que visualiza el valor de la excepción.

La clase **`Practica4Ejercicio13Main`**,define un método llamado **`calcula()`** que lanza un objeto de **`Practica4Ejercicio13Exception`**.

La excepción es lanzada cuando el parámetro entero de **`calcula()`** es mayor que 10. El método **main()** establece un gestor de excepciones para **`Practica4Ejercicio13Exception.`** Se llama a **`calcula()`** con un valor menor que 10 y con uno mayor que 10 para mostrar los dos caminos que sigue el código.

```
make -f makeP4 ejecuta13
```

Uso de excepciones

La gestión de excepciones proporciona un mecanismo poderoso para controlar programas complejos que tienen muchas características dinámicas durante la ejecución.

Es importante pensar que **try**, **throw** y **catch** son maneras limpias de manejar errores y problemas inesperados en la lógica de un programa. Si es como la mayoría de los programadores, entonces estará acostumbrado a devolver un código de error cuando se produce un fallo en un método. Cuando programe en Java debería terminar con este hábito. Cuando un método puede fallar tiene que lanzar una excepción. Esta es la manera más limpia de gestionar los fallos.

Por último, no debería considerar que las sentencias de gestión de excepciones de Java sean un mecanismo general para realizar ramificaciones, ya que si hace esto, sólo conseguirá enmarañar su código y complicar su mantenimiento.

Uso de excepciones con el jdb

Recuerde que las órdenes **`catch`** e **`ignore`** se pueden utilizar en el jdb. La orden `catch` permite indicar al depurador que este debe detener la ejecución del programa en el caso de que se lance la excepción que le indiquemos. La sintaxis de esta orden es `catch excepcion` (incluyendo el paquete al que pertenece la excepción `excepcion`). De forma opuesta, la orden ignore nos permite anular el efecto de una orden `catch` previa.

Ejecute la siguiente sentencia

```
jdb -classpath ./bin -sourcepath ./src fp2.poo.practica4.Practica4Ejercicio13Main
```

Capture la excepción que se ha implementado (**`fp2.poo.practica4.Practica4Ejercicio13Exception`**) con el comando **`catch`** dentro de jdb, de la siguiente forma:

```
catch fp2.poo.practica4.Practica4Ejercicio13Exception
```

Y ejecute con **`run`**.

Ejercicio : Modificar la clase **`Calculadora`** de la práctica 1 para que se limite el rango de los operandos de 0 a 100, en los métodos *sumar*, *restar*, *multiplicar* y *dividir* dos números. Para ello se crea la clase **`ValoresFueraDeRangoExcepcion`**. En el caso de que los números sobre los que se quiera operar estén fuera de este rango, se debe lanzar esta excepción. Crear un programa principal que utilice la calculadora para hacer varios cálculos y capture la excepción **`ValoresFueraDeRangoExcepcion`** y muestre un mensaje por pantalla. También se debe capturar la excepción de división por 0 en el caso de la división.