

UD 4

GESTIÓN DE ERRORES

Pedro J. Ponce de León

Versión 20151005



Gestión Errores

Objetivos

- **Saber utilizar las sentencias de control de excepciones para observar, indicar y manejar excepciones, respectivamente.**
- **Comprender las ventajas del manejo de errores mediante excepciones frente a la gestión de errores tradicional de la programación imperativa.**
- **Comprender la jerarquía de excepciones estándar.**
- **Ser capaz de crear excepciones personalizadas**
- **Ser capaz de procesar las excepciones no atrapadas y las inesperadas.**
- **Comprender la diferencia entre excepciones verificadas y no verificadas**

Indice

- Motivación
- Excepciones
 - Concepto
 - Lanzamiento y captura
 - Especificación de excepciones
- Excepciones estándar en Java
- Excepciones de usuario
- Particularidades de las excepciones
- Regeneración de excepciones

Gestión de Errores

Motivación

- Gestión de errores 'tradicional' (o al estilo de C)

```
int main (void)
{
    int res;
    if (puedo_fallar () == -1)
    {
        cout << "¡Algo falló!" << endl;
        return 1;
    }
    else cout << "Todo va bien..." << endl;
    if (dividir (10, 0, res) == -1)
    {
        cout << "¡División por cero!" << endl;
        return 2;
    }
    else cout << "Resultado: " << res << endl;
    return 0;
}
```

Flujo normal
Flujo de error

Gestión de Errores

Motivación

- Nos obliga a definir un esquema de programación similar a :
 - Llevar a cabo tarea 1
 - *Si se produce error*
 - *Llevar a cabo procesamiento de errores*
 - Llevar a cabo tarea 2
 - *Si se produce error*
 - *Llevar a cabo procesamiento de errores*
- A esto se le llama código ***espaguetti***
- Problemas de esta estrategia
 - Entremezcla la lógica del programa con la del tratamiento de errores (disminuye legibilidad)
 - El código cliente (llamador) no está obligado a tratar el error
 - Los 'códigos de error' no son consistentes.

Gestión de Errores

Motivación

- ¿Qué podemos hacer en lugar de crear código *spaguetti*?
 - Abortar el programa
 - ¿Y si el programa es crítico?
 - Usar indicadores de error globales
 - El código cliente (llamador) no está obligado a consultar dichos indicadores.
 - USAR EXCEPCIONES
 - La idea básica es
 - Cuando no disponemos de información suficiente para resolver el problema en el contexto actual (método o ámbito), lo indicamos a nuestro llamador mediante una excepción.

Gestión de Errores

Excepciones: Concepto

- Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
- Son objetos (instancias de clases) que contienen información sobre el error.
- Las excepciones se tratan mediante sentencias de control del flujo de error que separan el código para manejar errores del resto mediante :
 - **throw, try, catch, finally**
- Por defecto, una excepción no se puede ignorar: hará que el programa aborte.
 - Una excepción pasará sucesivamente de un método a su llamador hasta encontrar un bloque de código que la trate.

Gestión de Errores

Excepciones: **Comportamiento**

- Las excepciones son **lanzadas** (`throw`) por un método cuando éste detecta una condición excepcional o de error.
- Esto interrumpe el control normal de flujo y provoca la finalización prematura de la ejecución del método y su retorno al llamador.

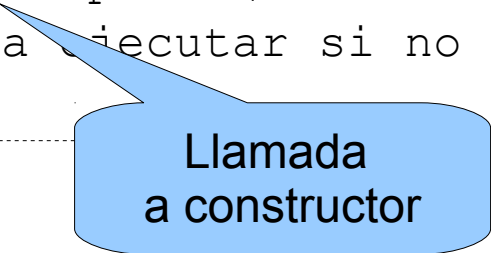
Gestión de Errores

Excepciones: **Lanzamiento**

- La cláusula **throw** lanza una excepción
 - Una excepción es un objeto.
 - La clase **Exception** (Java) representa una excepción general.

Lanzamiento

```
int LlamameConCuidado(int x) {  
    if (condicion_de_error(x) == true)  
        throw new Exception("valor "+x+" erroneo");  
    //... código a ejecutar si no hay error ...  
}
```



Llamada
a constructor

Gestión de Errores

Excepciones: **Comportamiento**

- Las excepciones pueden ser **capturadas** (`try/catch`), normalmente por el código cliente (llamador).
- Si el llamador no captura la excepción, su ejecución también terminará y la excepción 'saldrá' de nuevo hacia un ámbito más externo y así sucesivamente hasta encontrar un lugar donde es capturada.
- Una excepción no capturada provocará que el programa aborte.

```
try {  
    f(...); // f puede lanzar Excepcion  
} catch (Excepcion ex) {  
    // tratamiento del error  
}
```

Gestión de Errores

Excepciones: **Captura**

- La instrucción **catch** es como una llamada a función: recibe un argumento.

Captura

```
public static void main() {  
    try {  
        LlamameConCuidado(-0);  
    } catch (Exception ex) {  
        System.err.println("No tuviste cuidado: "+ ex);  
        ex.printStackTrace();  
    }  
}
```

Gestión de Errores

Excepciones: Java

```
void Func() {  
  
    if(detecto_error1) throw new Tipo1();  
    ...  
    if(detecto_error2) throw new Tipo2();  
    ...  
}
```



constructor

```
try  
{  
    // Codigo de ejecución normal  
    Func(); // puede lanzar excepciones  
}  
catch (Tipo1 ex)  
{  
    // Gestión de excep tipo 1  
}  
catch (Tipo2 ex)  
{  
    // Gestión de excep tipo 2  
}  
finally {  
    // se ejecuta siempre  
}  
//continuación del código
```

Gestión de Errores

Excepciones: C++

```
void Func() {  
  
    if (detecto_error1) throw Tipo1();  
    ...  
    if (detecto_error2) throw Tipo2();  
    ...  
}
```



constructor

```
try  
{  
    // Código de ejecución normal  
    Func(); // puede lanzar excepciones  
    ...  
}  
catch (Tipo1 &ex)  
{  
    // Gestión de excep tipo 1  
}  
catch (Tipo2 &ex)  
{  
    // Gestión de excep tipo 2  
}  
catch (...)  
{  
    /* Gestión de cualquier excepción no  
       capturada mediante los catch  
       anteriores*/  
}  
//Continuación del código
```

Gestión de Errores

Excepciones: **Sintaxis**

- En C++ y en Java:
 - El bloque **try** contiene el código que forma parte del funcionamiento normal del programa.
 - El bloque **catch** contiene el código que gestiona los diversos errores que se puedan producir.
- Sólo en JAVA:
 - Se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema. El bloque finally se puede ejecutar:
 - (1) tras finalizar el bloque try o
 - (2) después de las cláusulas catch.

Gestión de Errores

Excepciones: **Funcionamiento**

- **Funcionamiento:**
 1. Ejecutar instrucciones **try**
 - Si hay error, interrumpir el bloque **try** e ir a bloque **catch** correspondiente
 2. Continuar la ejecución después de los bloques **catch**
- La excepción es capturada por el bloque **catch** cuyo argumento coincida con el tipo de objeto lanzado por la sentencia **throw**. La búsqueda de coincidencia se realiza sucesivamente sobre los bloques **catch** en el orden en que aparecen en el código hasta que aparece la primera concordancia.
- En caso de no existir un manejador adecuado a una excepción determinada, se desencadena un protocolo que, por defecto, produce sin más la finalización del programa. (En Java la excepción es capturada y mostrada por la máquina virtual).

Gestión de Errores

Excepciones: **especificación de excepción**

- En Java y C++ un método debe/puede indicar, en su declaración, qué excepciones puede lanzar (directa o indirectamente) mediante la **especificación de excepción**.
- Este especificador se utiliza en forma de sufijo en la declaración de la función y tiene la siguiente **sintaxis** en Java:

throws (<lista-de-tipos>)

- <lista-de-tipos> indica qué excepciones puede lanzar el método.

```
public int f() throws E1, E2 {...}
```

f () puede lanzar excepciones de tipo E1 o E2.

Gestión de Errores

Excepciones: **Uso correcto**

```
void f() {  
    try { g(); } catch(Exception ex) {  
        System.err.println(ex.queHaPasado());  
    }  
    // sigue...  
}
```

f() captura el tipo de excepciones que puede lanzar h()

```
void g() {  
    h();  
    // sigue...  
}
```

g() no la captura...

```
void h() {  
    if (algo_fallar)  
        throw new Exception("¡Mecachis!");  
    // sigue...  
}
```

h() puede lanzar una excepción...

Uso correcto de las excepciones

- Un método que lanza excepciones se limita a señalar que se ha producido algún tipo de error, pero, por regla general, no debe tratarlo. Delegará dicho tratamiento en quienes invoquen a dicho método.

Gestión de Errores

Excepciones: **Uso incorrecto (1)**

- No debes usar excepciones para situaciones no excepcionales, como como una alternativa a bucles for, bloques do-while o simples 'if'.

```
// Ejemplo de abuso de excepciones
// para detectar el fin de un bucle
try {
    int[] a = new int[5];
    int i=0;
    while (true)
        a[i++] = x;
} catch (ArrayIndexOutOfBoundsException ex) {
    ...
}
```

Gestión de Errores

Excepciones: **Uso incorrecto (2)**

- No debes capturar una excepción en el mismo método donde la lanzas:

```
// Ejemplo de 'autocaptura' de excepciones
try {
    if (error)
        throw new Exception();
    f();
} catch (Exception ex) {
    System.err.println("¡hay problemas!");
}
```

¡Mejor usa if-else!

```
if (error)
    System.err.println("¡hay problemas!");
else
    f();
```

Gestión de Errores

Excepciones estándares en Java

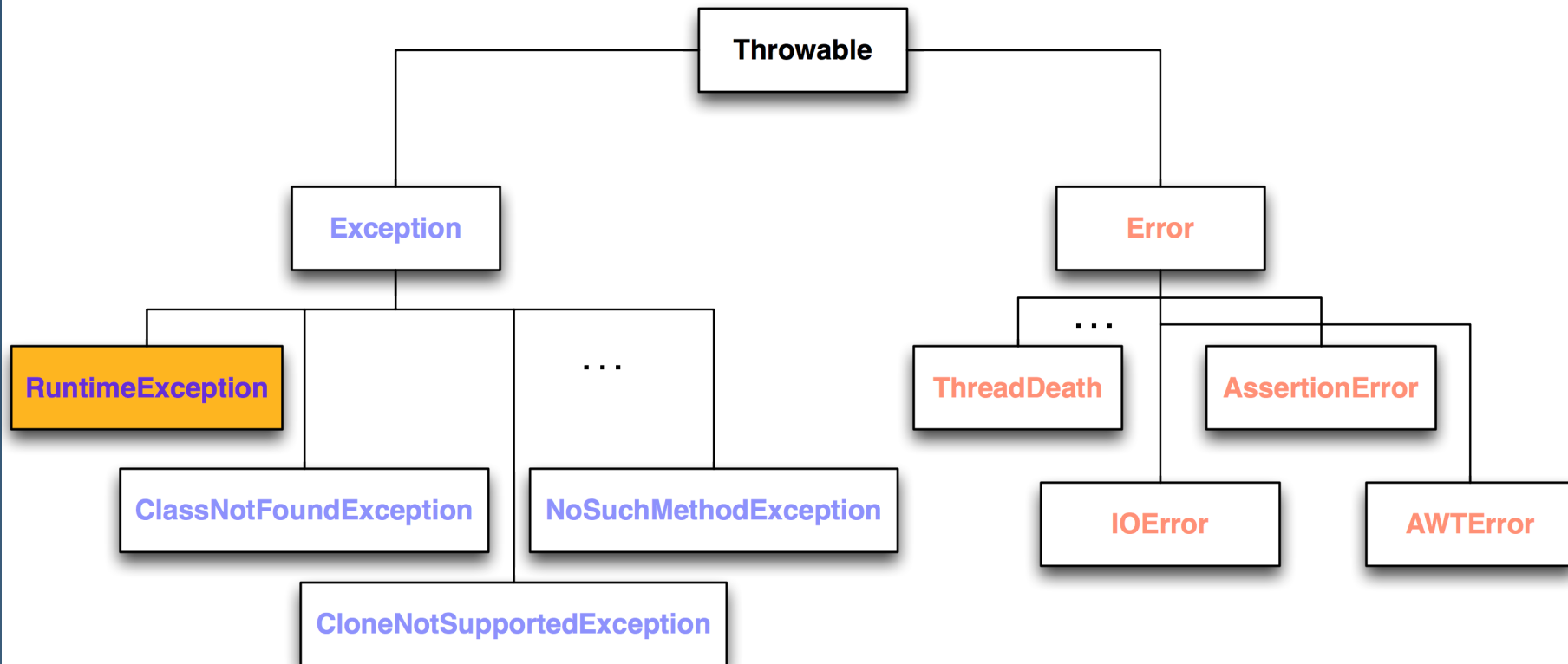
- Todas las excepciones lanzadas por componentes de la API de Java son tipos derivados de la clase **Throwable** (definida en `java.lang`), que tiene, entre otros, los siguientes métodos:

```
class Throwable {  
    ...  
    public Throwable();  
    public Throwable(String message);  
    ...  
    public String getMessage();  
    public void printStackTrace();  
    public String toString(); // nombre clase + message  
}
```

- Estos métodos están disponibles cuando creamos excepciones de usuario (más adelante).

Gestión de Errores

Excepciones estándar en Java



Excepciones verificadas (checked exceptions):

Si un método lanza alguna de estas excepciones directa o indirectamente, excepto las de tipo *RuntimeException*, Java obliga a que lo declare en su *especificación de excepciones*, en tiempo de compilación.

Gestión de Errores

Excepciones estándar en Java

Exception:

Indican errores de ejecución del API de Java o nuestros. Son excepciones verificadas (excepto las de tipo `RuntimeException`).

Error:

Indican errores de compilación o del sistema. Reservado para la JVM. Normalmente no se capturan.

RuntimeException:

Suelen indicar errores de programación, no del usuario, por lo que tampoco se suelen capturar.

Todas las excepciones estándar disponen de al menos dos constructores: Uno por defecto y otro que acepta un `String` (que describe lo ocurrido).

Podemos generar (throw) cualquier tipo de excepción estándar, pero no se suelen generar las de tipo `Error` o `RuntimeException`.

Gestión de Errores

Excepciones estándar en Java

Lista de excepciones más comunes

java.lang

- ClassCastException (RT)
- ArithmeticException (RT)
- ArrayIndexOutOfBoundsException (RT)
- ClassCastException (RT)
- ClassNotFoundException
- IllegalArgumentException (RT)
- NullPointerException (RT)
- NumberFormatException (RT)

java.io :

- EOFException
- FileNotFoundException

(RT: RuntimeException)

Gestión de Errores

Excepciones verificadas

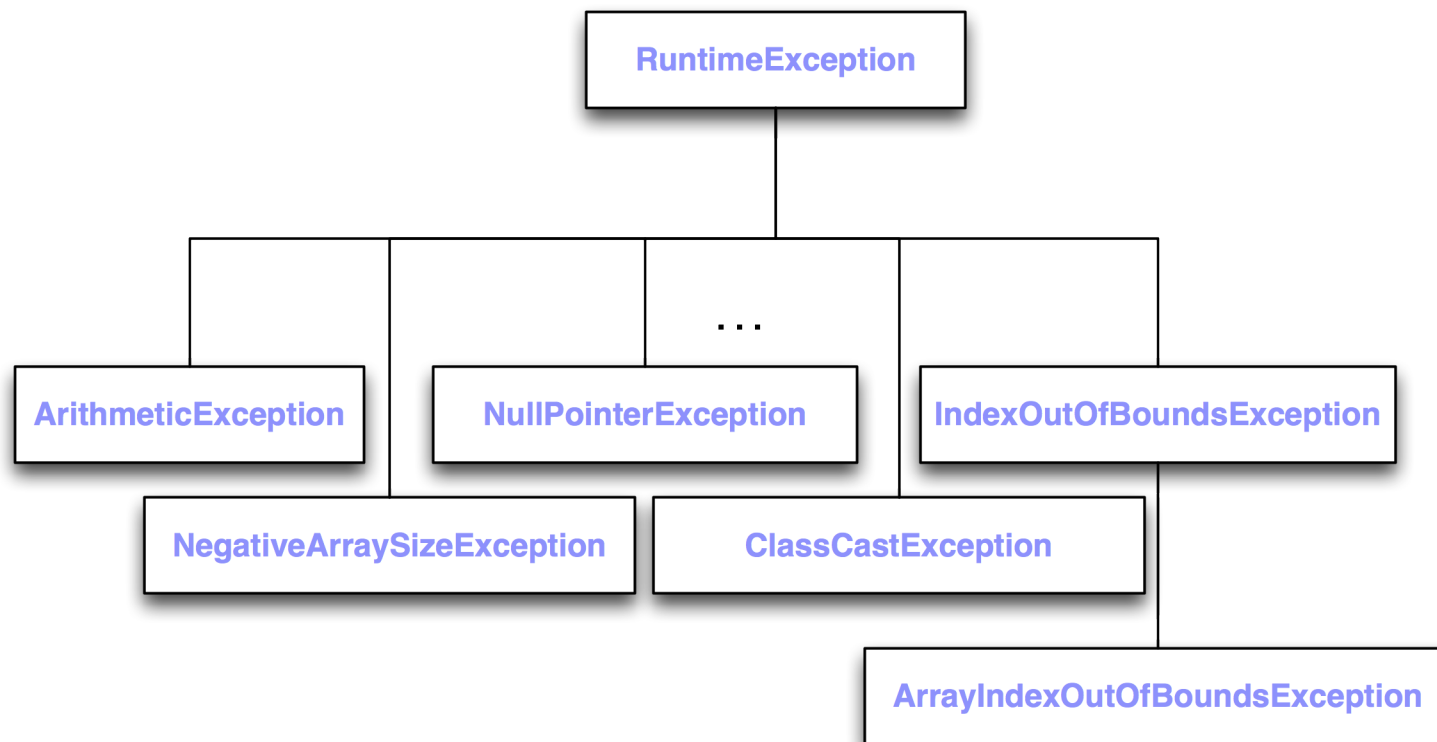
Ejemplo: *FileNotFoundException*

```
void f(String fichero) throws FileNotFoundException {  
    File f = new File(fichero);  
    if (!f.exists()) {  
        throw new FileNotFoundException(fichero);  
    }  
    // usamos f aquí ...  
}
```

```
void f(String fichero) {  
    File f = new File(fichero);  
    try {  
        Scanner sc = new Scanner(f); // puede lanzar FileNotFoundException  
        // usamos sc si puede abrir f  
    } catch (FileNotFoundException ex) {  
        System.err.println("Fichero no encontrado");  
        ex.printStackTrace();  
    }  
}
```


Gestión de Errores

RuntimeException: **Excepciones no verificadas**



Excepciones no verificadas (unchecked exceptions):

Son todas aquellas que derivan (`extends`) de RuntimeException.

Si un método puede lanzarlas, no es obligatorio incluirlas en su especificación de excepciones.

Gestión de Errores

RuntimeException: **Excepciones no verificadas**

Ejemplo: *NumberFormatException*

```
try {  
    Scanner sc = new Scanner(System.in);  
    String num = sc.next();  
  
    float f = Float.valueOf(num);  
} catch (NumberFormatException ex) {  
    System.err.println("Se esperaba un número real");  
    ex.printStackTrace();  
}
```

Gestión de Errores

RuntimeException: **Excepciones no verificadas**

Ejemplo: *ArrayIndexOutOfBoundsException*

```
void f(int[] a, int i) {  
    System.out.println(a[i]+10);  
}
```

```
void f(int[] a, int i) throws ArrayIndexOutOfBoundsException {  
    System.out.println(a[i]+10);  
}
```

```
void f(int[] a, int i) {  
    try {  
        System.out.println(a[i]+10);  
    } catch (ArrayIndexOutOfBoundsException ex)  
        System.err.println("Indice fuera de rango");  
        ex.printStackTrace();  
}
```

Gestión de Errores

RuntimeException: **Excepciones no verificadas**

Ejemplo: *IllegalArgumentException*

```
void f(int x) // f() no acepta valores negativos
{
    if (x < 0)
        throw new IllegalArgumentException("¡negativos no!");
    // usar x aquí...
}
```

Gestión de Errores

RuntimeException: **especificación de excepción**

- La ausencia de especificador indica que la función podría lanzar cualquier excepción no verificada:

```
public int f() {  
    if (algo_falla)  
        throw new RuntimeException("Algo ha fallado");  
}
```

- O ninguna excepción

```
public int g() {  
    return 3+2;  
}
```

Gestión de Errores

Excepciones de usuario

- Es habitual tipificar errores de una aplicación creando clases de objetos que representan diferentes circunstancias de error.
- La ventaja de hacerlo así es que
 - Podemos incluir información extra al lanzar la excepción.
 - Podemos agrupar las excepciones en jerarquías de clases.
- Creamos excepciones de usuario tomando como base la clase `Exception`:

```
class miExcepcion extends Exception {  
    private int x;  
  
    public miExcepcion(int a, String m) {  
        super(msg); x=a; }  
    public String queHaPasado() { return getMessage(); }  
    public int getElCulpable() { return x; }  
}
```

Gestión de Errores. Particularidades

Orden de captura de excepciones

- El orden de colocación de los bloques catch es importante.

```
void f() { if (error) throw new miExcepcion("ERROR 1"); }
void g() { if (error) throw new RuntimeException("ERROR 2"); }
void h() { if (error) throw new Exception("ERROR 3"); }
void probando() {
    try {
        f();
        g();
        h();
    }
    catch (miExcepcion e1) { ... tratar error 1 ... }
    catch (RuntimeException e2) { ... tratar error 2 ... }
    catch (Exception e3) { ... tratar error 3 ... }
}
```

Si colocamos el 'catch (Exception e3)' el primero, todos los errores serían tratados como errores 3. El compilador de Java detecta esta situación incorrecta.

Gestión de Errores. Particularidades

Anidamiento de bloques **try**

- Los bloques try/catch se pueden anidar

```
try {  
    if (error1) throw new miExcepcion("ERROR 1");  
    try {  
        if (error2) throw new RuntimeException("ERROR 2");  
        try {  
            if (error3) throw new Exception("ERROR 3");  
        }  
        catch (Exception e3) { ... tratar error 3 ... }  
    }  
    catch (RuntimeException e2) { ... tratar error 2 ... }  
}  
catch (miExcepcion e1) { ... tratar error 1 ... }
```


Gestión de errores. Particularidades

Excepciones en constructores

- Si se produce una excepción en un constructor (y no se captura dentro del mismo) el objeto no llega a ser construido.
- Normalmente esto sucede cuando los argumentos del constructor no permiten crear un objeto válido.

Opciones:

- Lanzar una excepción de usuario
- Lanzar `IllegalArgumentException`

```
class Construcccion {  
    public Construcccion(int x) throws IllegalArgumentException {  
        if (x<0)  
            throw IllegalArgumentException("No admito negativas.");  
    }  
  
    public static void main(String args[]) {  
        try {  
            Construcccion c = new Construcccion(-3);  
        } catch (Exception ex) { ... 'c' no ha sido construido ... }  
    }  
}
```

Gestión de Errores

Regeneración de una excepción

- A veces es necesario tratar parte del problema que generó una excepción a un determinado nivel y regenerarla para terminar de tratarlo a un nivel superior.
- Estrategias
 1. Regenerar la misma excepción
 2. Generar una excepción distinta

Gestión de Errores

Regeneración de una excepción

- Estrategia 1: regenerar la misma excepción

```
public class Rethrowing {  
    public static void f() throws Exception {  
        throw new Exception("thrown from f()");  
    }  
    public static void g() throws Exception {  
        try {  
            f();  
        } catch (Exception e) {  
            // tratar parte del problema aquí...  
            throw e; // regenerar la excepción  
        }  
    }  
  
    public static void main(String[] args) {  
        try {  
            g();  
        } catch (Exception e) {  
            // terminar de tratar el problema generado en f()  
        }  
    }  
}
```

Gestión de Errores

Regeneración de una excepción

- Estrategia 2: generar una excepción distinta

```
class OneException extends Exception {}
class TwoException extends Exception {}

public class RethrowNew {
    public static void f() throws OneException {
        throw new OneException();
    }
    public static void g() throws TwoException {
        try {
            f();
        } catch(OneException e) {
            // tratar parte del problema aquí...
            throw new TwoException(); // generar nueva excepción
        }
    }

    public static void main(String[] args) {
        try {
            g();
        } catch(TwoException e) {
            // terminar de tratar el problema generado en f()
        }
    }
}
```

Gestión de Errores

Excepciones: Resumen

■ **Ventajas:**

- Separar el manejo de errores del código normal
- Agrupar los tipos de errores y la diferenciación entre ellos
- Obliga al código cliente a tratar (o ignorar) expresamente las condiciones de error.

■ **Inconvenientes**

- Sobrecarga del sistema para gestionar los flujos de control de excepciones.

Gestión de errores

A tener en cuenta:

- Si se produce una excepción
 - ¿Se limpiará todo apropiadamente?
 - ¿Debemos tratarla en el lugar donde se produce?
 - ¿Mejor pasarla al siguiente nivel (al llamador)?
 - ¿Tratarla parcialmente y relanzarla (la misma u otra excepción diferente) al siguiente nivel?
 -
- Lo habitual es delegar en el llamador el tratamiento de la excepción:
 - “No captures una excepción si no sabes qué hacer con ella”

Gestión de Errores

Ejemplo

- EJERCICIO

- **Definid una excepción de usuario llamada**

- ExcepcionDividirPorCero que sea lanzada por el siguiente método al intentar dividir por cero:

```
class Division {  
    static float div(float x, float y)  
    { return x/y; }  
}
```

- Escribe un programa que invoque a div() y trate correctamente la excepción.

Gestión de Errores

Ejemplo

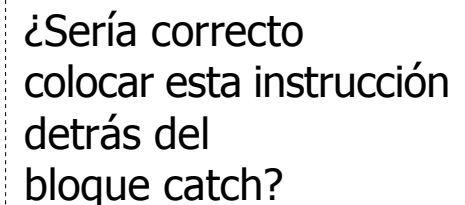
- SOLUCIÓN:

```
class ExcepcionDividirPorCero extends Exception {  
    public ExcepcionDividirPorCero(String msg)  
    { super(msg); }  
}
```


Gestión de Errores

Ejemplo

```
class Divisora {  
  
    public static void main(String args[])  
    {  
        double dividendo, divisor, resultado;  
        dividendo = 4.0;  
        divisor = 0.0;  
        try {  
            resultado = Division.div(dividendo, divisor);  
            System.out.println(dividendo+"/"+divisor+"="+resultado);  
        }  
        catch (ExcepcionDividirPorCero exce)  
        {  
            System.err.println(exce.getMessage());  
            exce.printStackTrace();  
        }  
    }  
}
```



¿Sería correcto
colocar esta instrucción
detrás del
bloque catch?

Gestión de Errores

Excepciones: Ejercicio propuesto

■ **Ejercicio**

- Define una clase `PilaEnteros` que gestione una pila de enteros. La pila se crea con una capacidad máxima. Queremos que la pila controle su desbordamiento al intentar apilar más elementos de los que admite, mediante una excepción de usuario `ExcepcionDesbordamiento`. Define la clase `PilaEnteros` y sus métodos `Pila()` y `apilar()`.
- Implementa un programa principal de prueba que intente apilar más elementos de los permitidos y capture la excepción en cuanto se produzca (dejando de intentar apilar el resto de elementos).

Gestión de errores

Bibliografía

- Bruce Eckel. ***Piensa en Java 4ª edición***
 - Cap. 12
- Bruce Eckel. ***Thinking in Java, 3rd. edition***
 - Cap. 9
- C. Cachero et al. ***Introducción a la programación orientada a objetos***
 - Cap. 5 (ejemplos en C++)