

Excepciones



Excepciones

- Una **excepción** es un evento o problema que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de ejecución de instrucciones. Una **excepción** interrumpe el procesamiento normal porque no cuenta con la información necesaria para resolver el problema en el contexto en que sucedió. Todo lo que se puede hacer es abandonar dicho contexto y pasar el problema a un contexto de más alto nivel.
- **Java** usa excepciones para proveer de **manejo de errores** a sus programas. Ej.: acceso a posiciones inválidas de un arreglo, falta de memoria en el sistema, abrir un archivo inexistente en el *file system*, ejecutar una *query* sobre una tabla inexistente de una bd, hacer un *casting* a un tipo de dato inapropiado, etc.
- En Java cuando ocurre un error en un método, se llevan a cabo los siguientes pasos: 1) se crea un **objeto excepción** en la **heap** con el operador **new**, como cualquier otro objeto Java, 2) luego se **lanza la excepción**: se interrumpe la ejecución del método y el **objeto excepción** es expulsado del contexto actual. En este punto, comienza a funcionar el **mecanismo de manejo de errores**: buscar un lugar apropiado donde continuar la ejecución del programa; el lugar apropiado es el **manejador de excepciones**, cuya función es recuperar el problema.

Excepciones

En **Java** las excepciones se clasifican en:

- **Checked Exception o Verificables en Compilación:** representan condiciones excepcionales que las aplicaciones bien escritas podrían anticipar y recuperar; son errores que el compilador verifica que se contemplen y que pueden recuperarse. JAVA obliga a los métodos que disparan este tipo de excepciones a que capturen y manejen el error o que lo recuperen, en caso contrario que indiquen todas las excepciones *checked* que pueden producirse dentro de su alcance. Por ejemplo al intentar abrir un archivo en el *file system* podría dispararse una excepción, dado que el archivo puede no existir, en ese caso una solución posible es pedirle al usuario que ingrese un nuevo nombre o propagar la excepción; otro error posible es intentar ejecutar una sentencia sql errónea.

NO Verificables en Compilación

- **Runtime Exception:** son errores internos de la aplicación que en general la aplicación no puede anticipar ni recuperar. Estas excepciones en general son *bugs* del programa y se producen por errores de lógica o por el mal uso de la API JAVA. Por ejemplo las excepciones aritméticas (división por cero), excepciones por referencias nulas (acceso a un objeto mediante un puntero nulo), excepciones de indexación (acceso a un elemento de un arreglo con un índice muy chico ó demasiado grande) y error de *casting*. JAVA no obliga a que estas excepciones sean especificadas ni capturadas para su manejo. Conviene solucionar el error que produce el *bug*.
- **Error:** son errores externos a la aplicación, relacionadas al hardware, a la falta de memoria y que la aplicación no puede anticipar ni recuperar.

Ejemplo

```
import java.io.*;
public class InputFile {
    private FileReader in;
    throws FileNotFoundException

    public InputFile (String filename) {
        in=new FileReader(filename);
    }
    throws IOException

    public String getWord() {
        int c;
        StringBuffer buf=new StringBuffer();
        do {
            c=in.read();
            if (Character.isWhitespace((char)c))
                return buf.toString();
            else
                buf.append((char)c);
        } while (c!=-1)
        return buf.toString();
    }
}
```

Si compilamos la clase **InputFile**, el compilador dispara mensajes de error similares a estos:

InputFile.java: 11: Warning: Exception **java.io.FileNotFoundException** must be caught, or it must be declared in throws clause of this method.

in=new FileReader(filename);

InputFile.java: 19: Warning: Exception **java.io.IOException** must be caught, or it must be declared in throws clause of this method.

c=in.read();

El compilador detecta que tanto el constructor de la clase **InputFile** como el método *getWord()* no especifican ni capturan las excepciones que se generan dentro de su alcance, por lo tanto la compilación falla.

Ejemplo

<code>in=new FileReader(filename);</code>	<code>c=in.read();</code>
El nombre pasado como parámetro al constructor de la clase FileReader podría no existir en el <i>file system</i> , por tanto el constructor disparará la excepción: java.io.FileNotFoundException.	El método getWord() de la clase InputFile , lee del objeto FileReader creado en el constructor de la clase usando el método read() . Este método dispara la excepción: java.io.IOException si por algún motivo no se puede leer.

- Al disparar estas excepciones, el **constructor** y el método **read()** de la **clase FileReader** permiten que los métodos que los invocan capturen dicho error y lo recuperen de una manera apropiada.
- La versión original de la clase **InputFile** **ignora** que el **constructor** y método **read()** de la clase **FileReader** disparan excepciones. Sin embargo el compilador JAVA obliga a que toda excepción *checked* sea capturada o especificada. Por lo tanto la **clase InputFile no compila**.

En este punto tenemos dos opciones:

- Ajustar al **constructor** de la clase **InputFile** y al método **getWord()** para que capturen y recuperen el error, o
- Ignorar los errores y darle la oportunidad a los métodos que invoquen al **constructor** y al método **getWord()** de **InputFile** a que recuperen los errores (usando la cláusula *throws*).

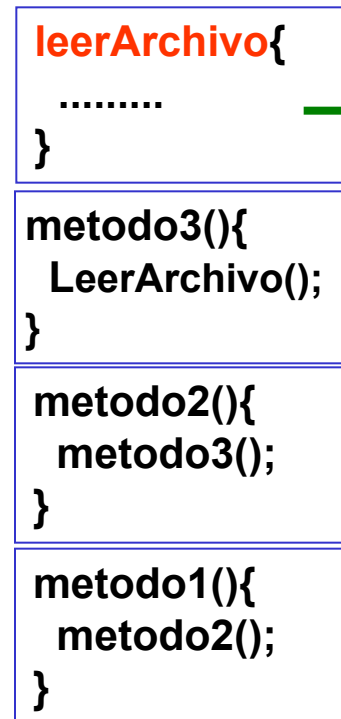
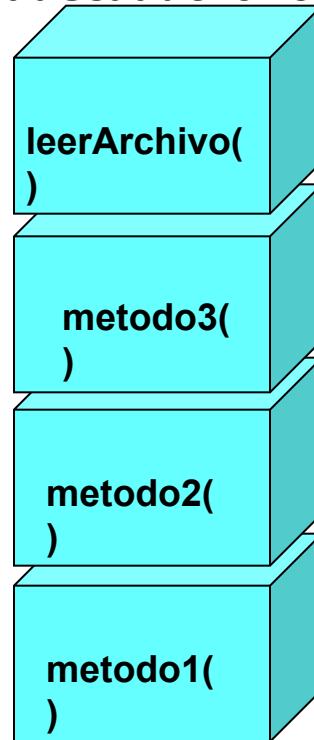
Búsqueda del Manejador de Excepciones

- Cuando un método dispara una excepción crea un objeto **Throwable** en la **heap** (la clase raíz de todas las excepciones), retorna dicho objeto y comienza a funcionar el **mecanismo de manejo de errores**. El sistema de ejecución de JAVA comienza a buscar en la **pila de ejecución de métodos invocados**, aquel que contenga un **manejador de excepciones adecuado para dicho error**.
- Un **manejador de excepción es adecuado** si el tipo de la excepción disparada coincide con la manejada.

Pila de ejecución

leerArchivo() dispara una excepción: crea un objeto Exception en la Heap; interrumpe su flujo normal de ejecución y el objeto excepción es lanzado del contexto actual y entregado a la VM.

El sistema de ejecución de Java comienza a buscar dónde continuar la ejecución: busca un **manejador de la excepción** apropiado en la pila de ejecución para recuperar el problema, comenzando por el método **leerArchivo()**. Podría retornar a un punto de la pila de ejecución bastante lejos del lugar dónde se produjo el error.



- Si el sistema de ejecución no encuentra un manejador apropiado en la pila de ejecución, la excepción será atendida por un manejador de *default* que finaliza la ejecución del programa.

Excepciones: Separar el Código

Usar excepciones permite SEPARAR el código regular del programa del código que maneja errores. JAVA permite escribir el flujo principal del código y tratar los casos excepcionales en otro lugar.

Cada cláusula **catch** es un manejador de excepciones, es similar a un método que tiene un único argumento de un tipo particular. Los identificadores **e1**, **e2**, **e3**, pueden usarse adentro del bloque de código del manejador, de la misma manera que los argumentos adentro del cuerpo de un método.

```
leerArchivo(){  
    try {  
        abrir archivo;  
        leer archivo;  
        cerrar archivo;  
    } catch (FileNotFoundException e1) {  
        hacerAlgo();    El archivo no se puede abrir  
    } catch (ReadFailedException e2) {  
        hacerAlgo();    Falla la lectura  
    } catch (FileCloseFailedException e3) {  
        hacerAlgo();    El archivo no se puede cerrar  
    }  
}
```

Flujo normal: permite concentrarse en el problema que se está resolviendo

Manejo de Excepciones: permite tratar los errores del código precedente

}

Excepciones: Propagar Errores

```
metodo1() {  
    try {  
        metodo2();  
    } catch (Exception e) {  
        procesarError();  
    }  
}
```

Flujo Normal

metodo1() es el único método interesado en recuperar el error que podría ocurrir en **leerArchivo()**

Manejo de Excepciones

```
metodo2() throws Exception {  
    metodo3();  
    //código  
    JAVA  
}
```

Se interrumpe la ejecución de **metodo2()** y se propaga el error ocurrido.

```
metodo3() throws Exception {  
    leerArchivo();  
    //código JAVA  
}
```

metodo2() y **metodo3()** propagan la excepción ocurrida en **leerArchivo()**. Ésta debe especificarse en la cláusula **throws** del método.

```
leerArchivo() throws Exception {  
    //código JAVA  
}
```

Dispara una excepción, se interrumpe el flujo normal de ejecución de **metodo3()** y se propaga el error.

```
leerArchivo() throws Exception {  
    Punto de creación del ERROR!!!  
    //código JAVA  
}
```

Se crea un objeto excepción con información sobre el error ocurrido, se interrumpe la ejecución de **leerArchivo()** y se lanza la excepción en busca de un manejador de la excepción

Jerarquía de Clases de Excepciones

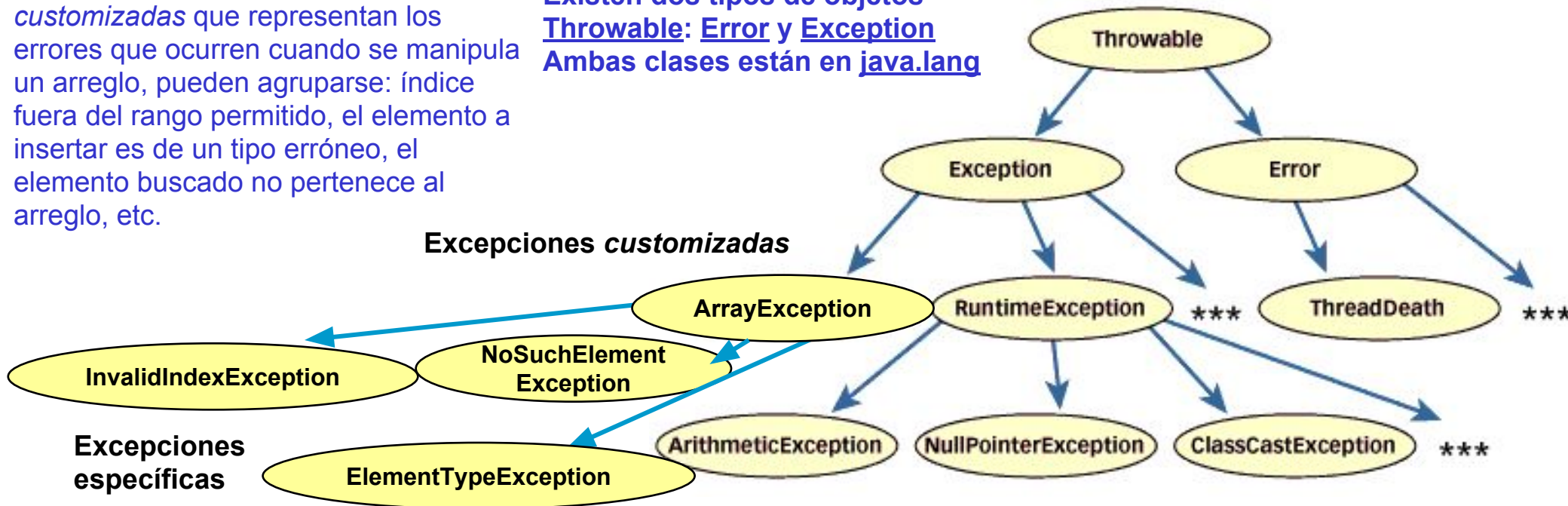
Throwable es la clase base de todos los errores y excepciones en JAVA. Solamente los objetos que son instancias de **Throwable** o de alguna de sus subclases pueden ser disparados por la JVM o por la sentencia **throw**. A su vez el tipo del argumento de la cláusula **catch** solamente puede ser **Throwable** o alguna de sus subclases.

Exception es el tipo base de todos los objetos que pueden dispararse desde cualquier método de la API JAVA o desde nuestros propios métodos cuando ocurren condiciones anormales en la aplicación. En algunos casos pueden preverse y recuperarse mediante un código específico.

Error representa problemas serios, relacionados con la computadora, la memoria o el procesador. Los errores son disparados por la JVM y los programadores no pueden hacer nada.

ArrayException: son excepciones *customizadas* que representan los errores que ocurren cuando se manipula un arreglo, pueden agruparse: índice fuera del rango permitido, el elemento a insertar es de un tipo erróneo, el elemento buscado no pertenece al arreglo, etc.

Existen dos tipos de objetos
Throwable: **Error** y **Exception**
Ambas clases están en java.lang



El nombre de la excepción representa el problema que ocurre y la idea es que sea lo más autoexplicativo posible. Existen clases de excepciones en diferentes paquetes: `java.util`, `java.net`, `java.io`, etc

RuntimeException

- Representan **errores de lógica de programación** que el programador no puede anticipar, ni necesitan recuperarse ni identificarse. Ejemplos: excepciones aritméticas como división por cero; excepciones de punteros al intentar acceder a un objeto a través una referencia nula; excepciones de índices al intentar acceder a una posición fuera del rango de un arreglo; excepciones de *casting*, etc.
- Este tipo de excepciones son subclase de **RuntimeException**. Estas excepciones pueden ocurrir en cualquier lugar de un programa y típicamente podrían ser numerosas, es por ello que son ***no-verificables en compilación*** o ***unchecked exceptions***, el compilador no fuerza a especificarlas, no puede detectarlas estáticamente.
- Son disparadas automáticamente por la JVM. Por ejemplo: **NullPointerException, ClassCastException, ArrayIndexOutOfBoundsException, etc**
- Este tipo de excepciones ayudan al proceso de *debugging* del código, los errores deben ser corregidos.

Componentes de un Manejador de Excepciones

① El bloque try

```
try {  
    sentencias JAVA  
}
```

Las sentencias JAVA que pueden disparar excepciones deben estar encerradas dentro de un bloque **try**.

Es posible,

- a. encerrar individualmente cada una de las sentencias JAVA que pueden disparar excepciones en un bloque *try* propio y proveer manejadores de excepciones individuales
- ó
- b. agrupar las sentencias que pueden disparar excepciones en un único bloque *try* y asociarle múltiples manejadores.

```
PrintWriter out=null;  
try{  
    out=new PrintWriter(new FileWriter("outFile.txt"));  
  
    for (int i=0; i< CANT; i++)  
        out.println("Valor en: "+ i +" = "+v.elementAt(i));  
}
```

El constructor de **FileWriter** dispara una **IOException** si no puede abrir el archivo

El método **elementAt()** de **vector** dispara una **ArrayIndexOutOfBoundsException** si el índice es muy chico (número negativo) ó muy grande

RuntimeException

Componentes de un Manejador de Excepciones

② El bloque **catch** (opcional)

- Son los **manejadores de excepciones**.
- La forma de asociar manejadores de excepciones con un bloque **try** es proveyendo uno o más bloques **catch** inmediatamente después del bloque **try**.

```
try{  
    //sentencias que pueden disparar excepciones  
} catch (SQLException e) {  
    System.err.println("Excepción capturada..." + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Excepción capturada..." + e.getMessage());  
}
```

Manejadores de Excepciones Especializados

- Si se dispara una excepción dentro del bloque **try**, el mecanismo de manejo de excepciones comienza a buscar el primer manejador de excepciones con un argumento que coincida con el tipo de excepción disparada. La coincidencia entre el tipo de la excepción disparada y la de su manejador puede no ser exacta. El tipo de las excepciones del manejador puede ser cualquier superclase de la excepción disparada.
- Luego, se ejecuta el bloque **catch** y la excepción se considera manejada/recuperada. Solamente se ejecuta el bloque **catch** que coincide con la excepción disparada.
- Si adentro del bloque **try** la invocación a diferentes métodos dispara el mismo tipo de excepción, solamente necesitamos un único manejador de excepciones.

Componentes de un Manejador de Excepciones

② El bloque catch (Continuación)

```
class Molestia extends Exception {}  
class Estornudo extends Molestia {}
```

```
public class SerHumano{  
    public static void main(String[] args) {  
        try {  
            throw new Estornudo();  
        } catch(Estornudo s) {  
            System.err.println("Manejador de Estornudo");  
        } catch(Molestia a) {  
            System.err.println("Manejador de Molestia");  
        }  
    }  
} // Fin de SerHumano
```

Podríamos eliminar el primer **catch** y dejar solamente el segundo:

```
catch(Molestia a) {  
    System.err.println("Manejador de Molestia");  
}
```

Captura las excepciones de tipo Molestia y todas las derivadas de Molestia

¿Qué ocurre si se invierte el orden de los manejadores?

NO COMPILA!!! pues se especificó un manejador inalcanzable (Estornudo), la excepción es manejada por el catch Molestia

Componentes de un Manejador de Excepciones

② El bloque catch (Continuación)

```
catch (FileNotFoundException e) {  
    //código del manejador  
}
```

Manejador de excepciones específico

Capturar un error basado en su grupo ó tipo general especificando alguna de las superclases de excepciones:

```
catch (IOException e) {  
    //código del manejador  
}
```

Se puede averiguar la excepción específica usando el parámetro **e**

Se puede establecer un manejador muy general que capture cualquier tipo de excepciones. Debe ubicarse al final de la lista de manejadores.

```
catch (Exception e) {  
    //código del manejador  
}
```

Los manejadores de excepciones muy generales hacen el código propenso a errores pues capturan y manejan excepciones que no fueron previstas. No son útiles para recuperación de errores

Componentes de un Manejador de Excepciones

② El bloque catch (Continuación)

La clase **Throwable** superclase de **Exception** provee un conjunto de métodos útiles para obtener información de la excepción disparada:

String getMessage(): devuelve un mensaje detallado de la excepción.

String getLocalizedMessage(): idem getMessage(), pero adaptado a la región.

String toString(): devuelve una descripción corta del Throwable incluyendo el mensaje (si existe).

void printStackTrace()
void printStackTrace(PrintStream)
void printStackTrace(java.io.PrintWriter)

Imprimen el **Throwable** y el *stack-trace* del **Throwable**. El *stack-trace* muestra la secuencia de métodos invocados que condujo al punto dónde se disparó la excepción. La primera versión, imprime en la salida de error estándar y en la segunda y tercera es posible especificar dónde queremos imprimir.

Ejemplo

```
public class TesteaExcepciones {
```

```
    public static void f() throws MiExcepcion {  
        System.err.println( "Origen de MiExcepcion desde f()" );  
        throw new MiExcepcion();  
    }
```

```
    public static void g() throws MiExcepcion {  
        System.err.println( "Origen de MiExcepcion desde g()" );  
        throw new MiExcepcion();  
    }
```

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

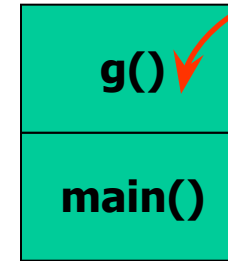
```
        try {  
            f();  
        } catch (MiExcepcion e) {  
            e.printStackTrace();  
        }
```

```
        try {  
            g();  
        } catch (MiExcepcion e) {  
            e.printStackTrace();  
        }
```

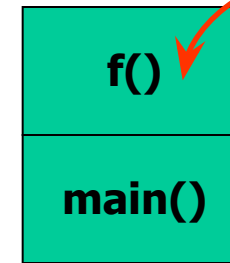
```
    }
```

```
}// Fin de la clase TesteaExcepciones
```

Dispara la MiExcepcion



Dispara la MiExcepcion



Origen de MiExcepcion desde f()

MiExcepcion

at TesteaExcepciones.f(TesteaExcepciones.java:6)

at TesteaExcepciones.main(TesteaExcepciones.java:14)

Origen de MiExcepcion desde g()

MiExcepcion

at TesteaExcepciones.g(TesteaExcepciones.java:10)

at TesteaExcepciones.main(TesteaExcepciones.java:20)

Componentes de un Manejador de Excepciones

③ El bloque **finally**

El último paso para definir un manejador de excepciones es liberar recursos antes que el control sea pasado a otra parte del programa. Esto se hace escribiendo el código necesario para liberar recursos adentro del bloque **finally**.

El sistema de ejecución de JAVA siempre ejecuta las sentencias del bloque *finally* independientemente de lo que sucedió en el bloque *try*.

```
finally {  
if (out != null){  
    System.out.println("Cerrando PrintWriter");  
    out.close();  
} else {  
    System.out.println("PrintWriter no fue abierto");  
}  
}
```

```
PrintWriter out=null;  
try{  
out=new PrintWriter(new FileWriter("OutFile.txt"));  
  
for (int i=0; i< CANT; i++)  
out.println("Valor en: "+ i + " = "+v.elementAt(i));  
}
```

Si no está el bloque **finally**: ¿Cómo se cierra el PrintWriter si no se provee un manejador de excepciones para **ArrayIndexOutOfBoundsException**?

Ejemplo Completo

```
public void writeList() {  
    PrintWriter out=null;  
    try {  
        out=new PrintWriter(new FileWriter("OutFile.txt"));  
        for (int i=0; i< CANT; i++)  
            out.println("Valor en: "+ i +" = "+ v.elementAt(i));  
    } catch (IOException e) {  
        System.err.println("Excepción capturada..." + e.getMessage());  
    } finally {  
        if (out !=null){  
            System.out.println("Cerrando PrintWriter");  
            out.close();  
        } else {  
            System.out.println("PrintWriter no fue abierto");  
        }  
    }  
}
```

v es una variable de instancia privada de tipo **Vector**.

CANT es una constante de clase inicializada en 20

El bloque **try** tiene tres formas posibles de terminar:


- El **constructor de FileWriter** falla y dispara una **IOException** por ej. si el usr no tiene permiso de escritura, el disco está lleno, etc.
- La sentencia **v.elementAt(i)** falla y dispara una **ArrayIndexOutOfBoundsException**.
- No sucede ninguna falla y el bloque **try** termina exitosamente.

Especificación de Excepciones

- JAVA fuerza a usar una sintaxis que permite informarle al programador las excepciones que podrían disparar los métodos que usa y de esta manera puede anticiparse a los errores y manejarlos. **Especificación de Excepciones**.
- La **Especificación de Excepciones** es parte de la declaración del método y se escribe después de la lista de argumentos. Es parte de la interface pública del método. Se usa la palabra clave **throws** seguida por una lista de tipos de excepciones que podrían dispararse en el alcance de dicho método.
- Si un método **NO** captura ni maneja las **excepciones checked** disparadas dentro de su alcance, el compilador JAVA fuerza al método a especificarlas en su declaración, **propagarlas**.
- En algunas situaciones es mejor que un método propague las excepciones, por ejemplo si se está implementando un paquete de clases, es posible que no se puedan prever las necesidades de todos los usuarios del paquete. En este caso es mejor no capturar las excepciones y permitirle a los métodos que usan las clases que manejen las excepciones que podrían dispararse.

```
public void writeList() throws IOException {  
    PrintWriter out=null;  
    out=new PrintWriter(new FileWriter("OutFile.txt"));  
  
    for (int i=0; i<CANT; i++)  
        out.println("Valor en: "+ i + " = "+v.elementAt(i))  
    out.close();  
}
```

No es necesario tratar ni propagar (especificar) la excepción **ArrayIndexOutOfBoundsException** porque es de tipo **RuntimeException**



El bloque finally

```
public void writeList() throws IOException {  
    PrintWriter out=null;  
    try{  
        out=new PrintWriter(new FileWriter("OutFile.txt"));  
  
        for (int i=0; i<CANT; i++)  
            out.println("Valor en: "+ i + " = "+v.elementAt(i));  
    }  
    finally{  
        if (out !=null){  
            System.out.println("Cerrando PrintWriter");  
            out.close();  
        } else {  
            System.out.println("PrintWriter no fue abierto");  
        }  
    }  
}
```

- El bloque **finally** debe tener un bloque **try**, el **catch** es opcional.
- Lo relevante del bloque **finally** es liberar los recursos que podrían haberse alocado en el bloque **try** independientemente de si se disparó o no una excepción

¿Cómo disparar Excepciones ?

La palabra clave **throw** es usada por todos los métodos que crean objetos excepción y requiere como único argumento un objeto **Throwable**.

El método **pop()** usa la cláusula **throws** para declarar que dentro de su alcance se puede disparar una **EmptyStackException**

```
public Object pop() throws EmptyStackException {  
    Object obj;  
    if (size == 0)  
        throw new EmptyStackException();  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    size--;  
    return obj;  
}
```

- Se crea un objeto en la *heap* que representa el error y la referencia la tiene la cláusula **throw**.
- El objeto **EmptyStackException** es retornado por el método **pop()**.

El método **pop()** chequea si hay algún elemento en la pila. Si está vacía, instancia un objeto **EmptyStackException** y lo lanza. Algún manejador en un contexto superior manejará el error.

Re-disparar una Excepción

```
try{  
    // código del bloque try  
} catch (FileNotFoundException e) {  
    System.out.println("Excepción de archivo no encontrado");  
    logger.log(E);  
    throw e; ← Re-dispara la excepción capturada  
} catch (IOException e) {  
    //código del manejador de excepciones  
}
```

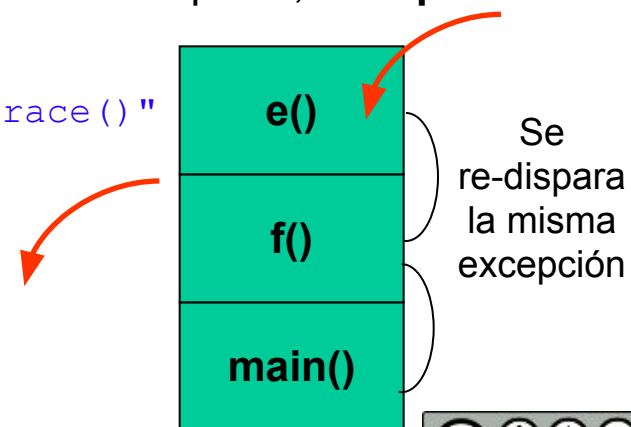
- Re-disparar una excepción causa que la excepción busque un manejador de excepciones en un contexto de más alto nivel. En este caso, el bloque catch maneja parcialmente la excepción ocurrida y la re-lanza para que un manejador de más alto nivel finalice su manejo.
- Las cláusulas **catch** del mismo bloque **try** son ignoradas.
- Se conserva todo acerca del objeto excepción, de manera tal que el manejador del contexto de más alto nivel que capture la excepción, pueda extraer información.
- Si la excepción que se re-dispara es la actual, la información que se imprime con el método **printStackTrace()** pertenece al origen de la excepción, no al lugar dónde se re-disparó.
- Es posible re-disparar una excepción diferente a la capturada. En este caso, la información del lugar de origen de la excepción se pierde y se tiene la información perteneciente al nuevo **throw**.

Re-disparar la misma Excepción

```
public class ReDisparar{  
    public static void e() throws Exception{  
        System.err.println("Origen de la excepción en e()");  
        throw new Exception("disparada en e()");  
    }  
    public static void f() throws Exception{  
        try {  
            e();  
        } catch (Exception e) {  
            System.err.println("Adentro de f(),  
                               e.printStackTrace()");  
            e.printStackTrace();  
            throw e;  
        }  
    }  
    public static void main(String[] args) {  
        try {  
            f();  
        } catch (Exception e) {  
            System.err.println("Capturada en el main(), e.printStackTrace()");  
            e.printStackTrace();  
        }  
    }  
} // Fin de ReDisparar
```

Origen de la excepción en e()
Adentro de f(), e.printStackTrace()
java.lang.Exception: disparada en e()
 at ReDisparar.e(ReDisparar.java:18)
 at ReDisparar.f(ReDisparar.java:22)
 at ReDisparar.main(ReDisparar.java:31)
Capturada en el main(), e.printStackTrace()
java.lang.Exception: disparada en e()
 at ReDisparar.e(ReDisparar.java:18)
 at ReDisparar.f(ReDisparar.java:22)
 at ReDisparar.main(ReDisparar.java:31)

Se dispara la excepción, **Exception**



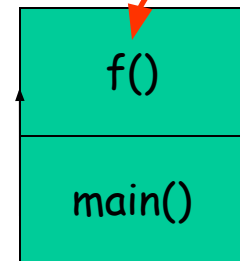
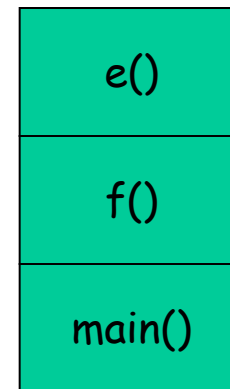
Re-disparar otra Excepción

```
public class ReDisparar{
    public static void e() throws Exception{
        System.err.println( "Origen de la excepción en e()" );
        throw new Exception("disparada en e()");
    }
    public static void f() throws Exception{
        try {
            e();
        } catch (Exception e) {
            System.err.println( "Adentro de f(),
                                e.printStackTrace()" );
            e.printStackTrace();
            throw new MiExcepcion( "MiExcepcion()" );
        }
    }
    public static void main(String[] args) {
        try {
            f();
        } catch (Exception e) {
            System.err.println( "Capturada en el main(),
                                e.printStackTrace()" );
            e.printStackTrace();
        }
    }
} // Fin de ReDisparar
```

Origen de la excepción en e()
Adentro de f(), e.printStackTrace()
java.lang.Exception: disparada en e()
at ReDisparar.e(ReDisparar.java:18)
at ReDisparar.f(ReDisparar.java:22)
at ReDisparar.main(ReDisparar.java:32)
Capturada en el main(), e.printStackTrace()
MiExcepcion: MiExcepcion()
at ReDisparar.f(ReDisparar.java:27)
at ReDisparar.main(ReDisparar.java:32)

Se dispara la excepción, **Exception**

Se re-dispara **MiExcepcion**



De la excepción MiExcepcion solamente sabe que se originó en f() y no en e()

Restricciones en Excepciones

Sobreescritura de métodos

- Cuando se sobreescribe un método solamente se pueden disparar las excepciones especificadas en la versión de la clase base del método. La utilidad de esta restricción es que el código que funciona correctamente para un objeto de la clase base, seguirá funcionando para un objeto de la clase derivada (principio fundamental de la OO)
- La **interface de especificación de excepciones** de un método puede reducirse y sobreescribirse en la herencia, pero nunca ampliarse. Es exactamente opuesto a lo que ocurre en la herencia con los especificadores de acceso de una clase.

```
public class A{
    public void f() throws AException{
        //código de f()
    }
    public void g() throws BException, CException{
        //código de g()
    }
}
```

```
public class B extends A{
    public void f(){
        //código de f()
    }
    public void g() throws DException{
        //código de g()
    }
}
```

```
class AException extends Exception {}
class BException extends AException {}
class CException extends AException {}
class DException extends BException {}
```

¿Está bien?SI!!!!

```
public class Test{
    public static void main(String[] args){
        try {
            A x= new A(); A x=new B();
            x.f();
            x.g();
        } catch (BException e) {}
        catch (CException e) {}
        catch (AException e) {}
    }
}
```

Restricciones en Excepciones

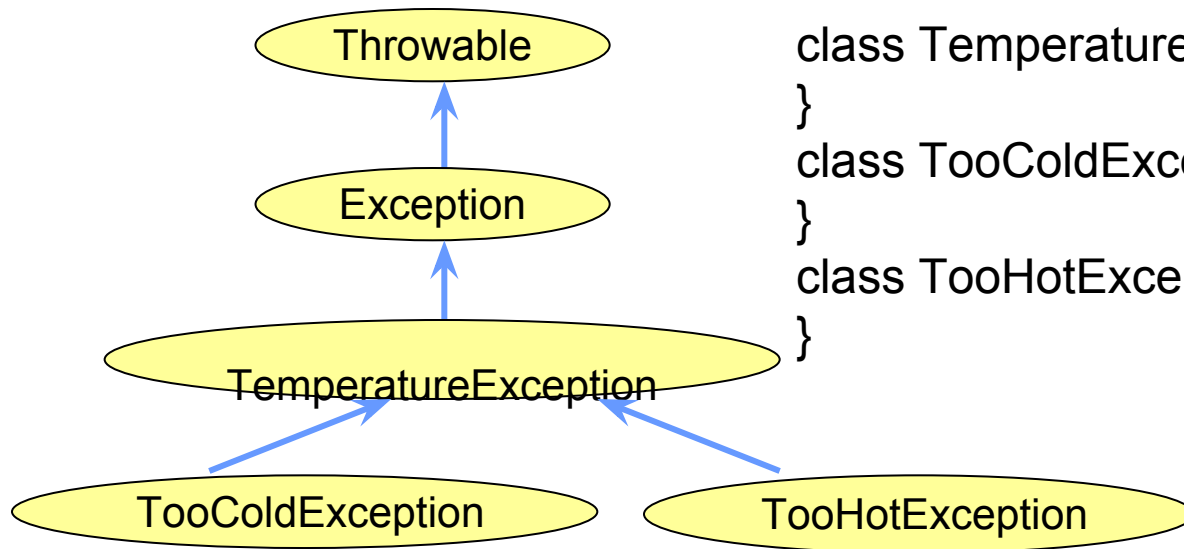
Constructores

- Los constructores no se sobrescriben.
- Los constructores de una subclase pueden disparar excepciones diferentes a las excepciones disparadas por el constructor de la superclase.
- Hay que ser cuidadoso de dejar el objeto que se intenta construir y no se puede, en un estado seguro.

Crear Excepciones Propias

Ejemplo: Café Virtual

Cuando se diseña un paquete de clases JAVA, éstas deben interactuar bien y sus interfaces deben ser fáciles de entender y de usar. Para ello es bueno diseñar clases de excepciones. Las condiciones excepcionales que pueden ocurrir cuando el cliente toma una taza de café son las siguientes: el café está muy frío o muy caliente.



```
class TemperatureException extends Exception {  
}  
class TooColdException extends TemperatureException {  
}  
class TooHotException extends TemperatureException {  
}
```

Convención de nombres: es una buena práctica agregar el texto Exception a todos los nombres de clases que heredan directa ó indirectamente de la clase **Exception**.

Ejemplo: Café Virtual

```
class VirtualPerson {  
    private static final int tooCold = 65;  
    private static final int tooHot = 85;  
  
    public void drinkCoffee(CoffeeCup cup) throws  
        TooColdException, TooHotException {  
  
        int temperature = cup.getTemperature();  
  
        if (temperature <= tooCold) {  
            throw new TooColdException();  
        }  
  
        else if (temperature >= tooHot) {  
            throw new TooHotException();  
        }  
    }  
}
```

Se declaran las excepciones que puede disparar el método **drinkCoffee()**



Se crea un
objeto
excepción y
se dispara

```
class CoffeeCup {  
    // 75 grados Celsius: es la temperatura ideal del café  
    private int temperature = 75;  
  
    public void setTemperature(int val){  
        temperature = val;  
    }  
  
    public int getTemperature() {  
        return temperature;  
    }  
}
```

Ejemplo: Café Virtual

```
class VirtualCafe {  
    public static void serveCustomer(VirtualPerson cust, CoffeeCup cup) {  
        try {  
            cust.drinkCoffee(cup);  
            System.out.println("El Café está OK.");  
        } catch (TooColdException e) {  
            System.out.println("El Café está muy frío.");  
        } catch (TooHotException e) {  
            System.out.println("El Café está muy caliente.");  
        }  
    }  
}
```

El método drinkCoffee() puede disparar las excepciones: **TooHotException** ó **TooColdException**

```
catch (TemperatureException e) {  
    System.out.println("El Café no está OK");  
}
```

- Se recomienda el uso de manejadores de excepciones especializados.
- Los manejadores genéricos (que agrupan muchos tipos de excepciones) no son útiles para recuperación de errores, dado que el manejador tiene que determinar qué tipo de excepción ocurrió para elegir la mejor estrategia para recuperar el error.
- Los manejadores genéricos pueden hacer que el código sea más propenso a errores, dado que se capturan y manejan excepciones que pueden no haber sido previstas por el programador.

Incorporar información a las Excepciones

- Las excepciones además de transferir el control desde una parte del programa a otra, permiten transferir información.
- Es posible agregar información a un objeto excepción acerca de la condición anormal que se produjo
- La cláusula *catch* permite obtener información interrogando directamente al objeto excepción.
- La clase *Exception* permite especificar mensajes de tipo String a un objeto excepción y, recuperarlos vía el método ***getMessage()*** (sobre el objeto excepción).
- Es posible agregar a un objeto *Exception* información de un tipo distinto que String. Para ello, es necesario agregar a la subclase de *Exception* datos y métodos de acceso a los mismos.

Incorporar información a las Excepciones

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

Dos constructores para
UnusualTasteException

Un programa que dispara una excepción de tipo *UnusualTasteException* puede hacerlo de las dos formas siguiente:

- a) **throw new UnusualTasteException()**
- b) **throw new UnusualTasteException("El Café parece Té")**

```
try {  
    //código JAVA que dispara excepciones  
} catch (UnusualTasteException e) {  
    String s = e.getMessage();  
    System.out.println(s);  
}
```

Se obtiene
información del
objeto excepción

Incorporar información a las Excepciones

```
abstract class TemperatureException extends Exception {  
    private int temperature;  
    public TemperatureException(int temperature) {  
        this.temperature = temperature;  
    }  
    public int getTemperature() {  
        return temperature;  
    }  
}
```

Datos y métodos de acceso a la información asociada a la excepción

```
class TooColdException extends TemperatureException {  
    public TooColdException(int temperature) {  
        super(temperature);  
    }  
}
```

```
class TooHotException extends TemperatureException {  
    public TooHotException(int temperature) {  
        super(temperature);  
    }  
}
```


Ejemplo - Café Virtual

```
class VirtualPerson {  
  
    private static final int tooCold = 65;  
    private static final int tooHot = 85;  
  
    public void drinkCoffee(CoffeeCup cup) throws  
        TooColdException, TooHotException {  
  
        int temperature = cup.getTemperature();  
        if (temperature <= tooCold) {  
            throw new TooColdException(temperature);  
        }  
        else if (temperature >= tooHot) {  
            throw new TooHotException(temperature);  
        }  
        //...  
    }  
    //...  
}
```

```
class VirtualCafe {  
  
    public static void serveCustomer(VirtualPerson cust,  
        CoffeeCup cup) {  
        try {  
            cust.drinkCoffee(cup);  
        } catch (TooColdException e) {  
            int temp = e.getTemperature();  
            if (temp > 55 && temp <= 65) {  
            } else if (temp > 0 && temp <= 55) {  
            } else if (temp <= 0) {  
                //código JAVA  
            }  
        } catch (TooHotException e) {  
            int temp = e.getTemperature();  
            if (temp >= 85 && temp < 100) {  
            } else if (temp >= 100 && temp < 2000) {  
            } else if (temp >= 2000) {  
            }  
        }  
    }  
}
```