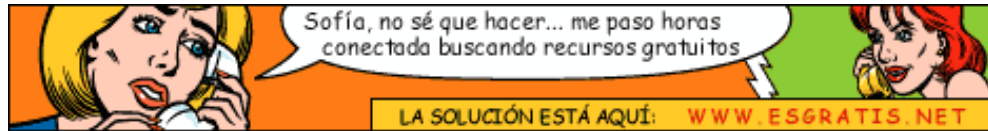




TutorJava recomienda...



Manejo de Errores Usando Excepciones Java

Autor-Traductor: [Juan Antonio Palos \(Ozito\)](#)

Puedes encontrar la Version Original en Ingles en (<http://java.sun.com>)


Leer comentarios (0) | [Escribir comentario](#) | Puntuación: (1 voto)

[Vota](#)

Indice de contenidos

- [Manejo de Errores Utilizando Excepciones](#)
- [¿Qué es un Excepción y Por Qué Debo Tener Cuidado?](#)
 - Ventaja 1: Separar el Manejo de Errores del Código "Normal"
 - Ventaja 2: Propagar los Errores sobre la Pila de Llamadas
 - Ventaja 3: Agrupar Errores y Diferenciación
 - ¿ Y ahora qué?
- [Primer Encuentro con las Excepciones Java](#)
- [Requerimientos Java para Capturar o Especificar Excepciones](#)
 - Capturar
 - Especificar
 - Excepciones Chequeadas
 - Excepciones que pueden ser lanzadas desde el ámbito de un método
- [Tratar con las Excepciones Java](#)
 - El ejemplo: ListOfNumbers
 - Capturar y Manejar Excepciones
 - Especificar las Excepciones que pueden ser Lanzadas por un Método
- [El Ejemplo ListOfNumbers](#)
- [Capturar y Manejar Excepciones](#)
 - El Bloque try
 - Los bloques catch
 - El bloque finally
 - Poniéndolo Todo Junto
- [El Bloque Try](#)
- [Los Bloques catch](#)
 - Ocurre una IOException
 - Capturar Varios Tipos de Excepciones con Un Manejador
- [El Bloque finally](#)
 - ¿Es realmente necesaria la sentencia finally?
- [Poniéndolo todo Junto](#)
 - Escenario 1:Ocurre una excepción IOException
 - Escenario 2: Ocurre una excepción ArrayIndexOutOfBoundsException
 - Escenario 3: El bloque try sale normalmente
- [Especificar las Excepciones Lanzadas por un Método](#)
- [La Sentencias throw](#)
 - La clausula throws
- [La Clase Throwable y sus Subclases](#)
 - Error

- Exception
- Excepciones en Tiempo de Ejecución
- [Crear Clases de Excepciones](#)
 - ¿Qué puede ir mal?
 - Elegir el Tipo de Excepción Lanzada
 - Elegir una Superclase
 - Convenciones de Nombres
- [Excepciones en Tiempo de Ejecución - La Controversia](#)
- [Cambios en el JDK 1.1 que afectan a las Excepciones](#)
- [Cambios en el ejemplo ListOfNumbers](#)
- [Cambios en la Clase Throwable](#)
 - Nuevos Métodos

Leer comentarios (0) 

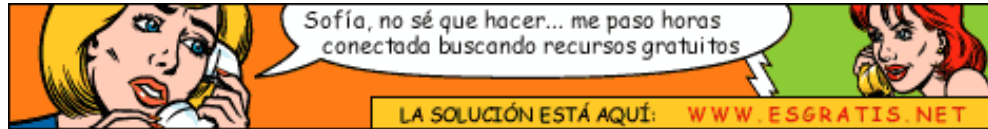
[Escribir comentario](#) 

Puntuación:  (1 voto)

[Vota](#) 



TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Manejo de Errores Utilizando Excepciones](#)

Manejo de Errores Utilizando Excepciones

Existe una regla de oro en el mundo de la programación: en los programas ocurren errores.

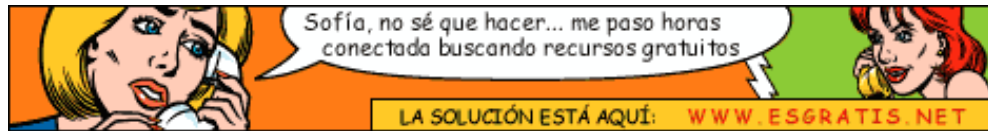
Esto es sabido. Pero ¿qué sucede realmente después de que ha ocurrido el error? ¿Cómo se maneja el error? ¿Quién lo maneja?, ¿Puede recuperarlo el programa?

El lenguaje Java utiliza excepciones para proporcionar capacidades de manejo de errores. En esta lección aprenderás qué es una excepción, cómo lanzar y capturar excepciones, qué hacer con una excepción una vez capturada, y cómo hacer un mejor uso de las excepciones heredadas de las clases proporcionadas por el entorno de desarrollo de Java.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- ¿Qué es un Excepción y Por Qué Debo Tener Cuidado?
 - Ventaja 1: Separar el Manejo de Errores del Código "Normal"
 - Ventaja 2: Propagar los Errores sobre la Pila de Llamadas
 - Ventaja 3: Agrupar Errores y Diferenciación
 - ¿Y ahora qué?

¿Qué es un Excepción y Por Qué Debo Tener Cuidado?

El término excepción es una forma corta de la frase "suceso excepcional" y puede definirse de la siguiente forma.

Definición:

Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.

Muchas clases de errores pueden utilizar excepciones -- desde serios problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites. Cuando dicho error ocurre dentro de un método Java, el método crea un objeto 'exception' y lo maneja fuera, en el sistema de ejecución. Este objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando ocurrió el error. El sistema de ejecución es el responsable de buscar algún código para manejar el error. En terminología Java, crear un objeto exception y manejarlo por el sistema de ejecución se llama lanzar una excepción.

Después de que un método lance una excepción, el sistema de ejecución entra en acción para buscar el manejador de la excepción. El conjunto de "algunos" métodos posibles para manejar la excepción es el conjunto de métodos de la pila de llamadas del método donde ocurrió el error. El sistema de ejecución busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el "manejador de excepción" adecuado.

Un manejador de excepción es considerado adecuado si el tipo de la excepción lanzada es el mismo que el de la excepción manejada por el manejador. Así la excepción sube sobre la pila de llamadas hasta que encuentra el manejador apropiado y una de las llamadas a métodos maneja la excepción, se dice que el manejador de excepción elegido captura la excepción.

Si el sistema de ejecución busca exhaustivamente por todos los métodos de la pila de llamadas sin encontrar el manejador de excepción adecuado, el sistema de ejecución finaliza (y consecuentemente y el programa Java también).

Mediante el uso de excepciones para manejar errores, los programas Java tienen las siguientes ventajas frente a las técnicas de manejo de errores tradicionales.

- Ventaja 1: Separar el Manejo de Errores del Código "Normal"
- Ventaja 2: Propagar los Errores sobre la Pila de Llamadas
- Ventaja 3: Agrupar los Tipos de Errores y la Diferenciación de éstos

Ventaja 1: Separar el Manejo de Errores del Código "Normal"

En la programación tradicional, la detección, el informe y el manejo de errores se convierte en un código muy liado. Por ejemplo, supongamos que tenemos una función que lee un fichero completo dentro de la memoria. En pseudo-código, la función se podría parecer a esto.

```
leerFichero {
```

```

abrir el fichero;
determinar su tamaño;
asignar suficiente memoria;
leer el fichero a la memoria;
cerrar el fichero;
}

```

A primera vista esta función parece bastante sencilla, pero ignora todos aquellos errores potenciales.

- ¿Qué sucede si no se puede abrir el fichero?
- ¿Qué sucede si no se puede determinar la longitud del fichero?
- ¿Qué sucede si no hay suficiente memoria libre?
- ¿Qué sucede si la lectura falla?
- ¿Qué sucede si no se puede cerrar el fichero?

Para responder a estas cuestiones dentro de la función, tendríamos que añadir mucho código para la detección y el manejo de errores. El aspecto final de la función se parecería esto.

```

codigodeError leerFichero {
    inicializar codigodeError = 0;
    abrir el fichero;
    if (ficheroAbierto) {
        determinar la longitud del fichero;
        if (obtenerLongitudDelFichero) {
            asignar suficiente memoria;
            if (obtenerSuficienteMemoria) {
                leer el fichero a memoria;
                if (falloDeLectura) {
                    codigodeError = -1;
                }
            } else {
                codigodeError = -2;
            }
        } else {
            codigodeError = -3;
        }
        cerrar el fichero;
        if (ficheroNoCerrado && codigodeError == 0) {
            codigodeError = -4;
        } else {
            codigodeError = codigodeError and -4;
        }
    } else {
        codigodeError = -5;
    }
    return codigodeError;
}

```

Con la detección de errores, las 7 líneas originales (en negrita) se han convertido en 29 líneas de código-- a aumentado casi un 400 %. Lo peor, existe tanta detección y manejo de errores y de retorno que en las 7 líneas originales y el código está totalmente atestado. Y aún peor, el flujo lógico del código también se pierde, haciendo difícil poder decir si el código hace lo correcto (si ¿se cierra el fichero realmente si falla la asignación de memoria?) e incluso es difícil asegurar que el código continúe haciendo las cosas correctas cuando se modifique la función tres meses después de haberla escrito. Muchos programadores "resuelven" este problema ignorándolo-- se informa de los errores cuando el programa no funciona.

Java proporciona una solución elegante al problema del tratamiento de errores: las excepciones. Las excepciones le permiten escribir el flujo principal de su código y tratar los casos excepcionales en otro lugar. Si la función leerFichero utilizara excepciones en lugar de las técnicas de manejo de errores tradicionales se podría parecer a esto.

```

leerFichero {
    try {
        abrir el fichero;
    }
}

```

```

        determinar su tamaño;
        asignar suficiente memoria;
        leer el fichero a la memoria;
        cerrar el fichero;
    } catch (falloAbrirFichero) {
        hacerAlgo;
    } catch (falloDeterminacionTamaño) {
        hacerAlgo;
    } catch (falloAsignaciondeMemoria) {
        hacerAlgo;
    } catch (falloLectura) {
        hacerAlgo;
    } catch (falloCerrarFichero) {
        hacerAlgo;
    }
}

```

Observa que las excepciones no evitan el esfuerzo de hacer el trabajo de detectar, informar y manejar errores. Lo que proporcionan las excepciones es la posibilidad de separar los detalles oscuros de qué hacer cuando ocurre algo fuera de la normal.

Además, el factor de aumento de código de este programa es de un 250% -- comparado con el 400% del ejemplo anterior.

Ventaja 2: Propagar los Errores sobre la Pila de Llamadas

Una segunda ventaja de las excepciones es la posibilidad de propagar el error encontrado sobre la pila de llamadas a métodos. Supongamos que el método leerFichero es el cuarto método en una serie de llamadas a métodos anidadas realizadas por un programa principal: metodo1 llama a metodo2, que llama a metodo3, que finalmente llama a leerFichero.

```

metodo1 {
    call metodo2;
}
metodo2 {
    call metodo3;
}
metodo3 {
    call leerFichero;
}

```

Supongamos también que metodo1 es el único método interesado en el error que ocurre dentro de leerFichero. Tradicionalmente las técnicas de notificación del error forzarían a metodo2 y metodo3 a propagar el código de error devuelto por leerFichero sobre la pila de llamadas hasta que el código de error llegue finalmente a metodo1 -- el único método que está interesado en él.

```

metodo1 {
    codigodeErrorType error;
    error = call metodo2;
    if (error)
        procesodelError;
    else
        proceder;
}
codigodeErrorType metodo2 {
    codigodeErrorType error;
    error = call metodo3;
    if (error)
        return error;
    else
        proceder;
}
codigodeErrorType metodo3 {
    codigodeErrorType error;
    error = call leerFichero;
    if (error)

```



```

        return error;
    else
        proceder;
}

```

Como se aprendió anteriormente, el sistema de ejecución Java busca hacia atrás en la pila de llamadas para encontrar cualquier método que esté interesado en manejar una excepción particular. Un método Java puede "esquivar" cualquier excepción lanzada dentro de él, por lo tanto permite a los métodos que están por encima de él en la pila de llamadas poder capturarlo. Sólo los métodos interesados en el error deben preocuparse de detectarlo.

```

metodo1 {
    try {
        call metodo2;
    } catch (excepcion) {
        procesodelError;
    }
}
metodo2 throws excepcion {
    call metodo3;
}
metodo3 throws excepcion {
    call leerFichero;
}

```

Sin embargo, como se puede ver desde este pseudo-código, requiere cierto esfuerzo por parte de los métodos centrales. Cualquier excepción chequeada que pueda ser lanzada dentro de un método forma parte del interface de programación público del método y debe ser especificado en la cláusula throws del método. Así el método informa a su llamador sobre las excepciones que puede lanzar, para que el llamador pueda decidir concienzuda e inteligentemente qué hacer con esa excepción.

Observa de nuevo la diferencia del factor de aumento de código y el factor de ofuscación entre las dos técnicas de manejo de errores. El código que utiliza excepciones es más compacto y más fácil de entender.

Ventaja 3: Agrupar Errores y Diferenciación

Frecuentemente las excepciones se dividen en categorías o grupos. Por ejemplo, podríamos imaginar un grupo de excepciones, cada una de las cuales representara un tipo de error específico que pudiera ocurrir durante la manipulación de un array: el índice está fuera del rango del tamaño del array, el elemento que se quiere insertar en el array no es del tipo correcto, o el elemento que se está buscando no está en el array. Además, podemos imaginar que algunos métodos querrían manejar todas las excepciones de esa categoría (todas las excepciones de array), y otros métodos podría manejar sólo algunas excepciones específicas (como la excepción de índice no válido).

Como todas las excepciones lanzadas dentro de los programas Java son objetos de primera clase, agrupar o categorizar las excepciones es una salida natural de las clases y las superclases. Las excepciones Java deben ser ejemplares de la clase Throwable, o de cualquier descendiente de ésta. Como de las otras clases Java, se pueden crear subclases de la clase Throwable y subclases de estas subclases. Cada clase 'hoja' (una clase sin subclases) representa un tipo específico de excepción y cada clase 'nodo' (una clase con una o más subclases) representa un grupo de excepciones relacionadas.

InvalidIndexException, ElementTypeException, y NoSuchElementException son todas clases hojas. Cada una representa un tipo específico de error que puede ocurrir cuando se manipula un array. Un método puede capturar una excepción basada en su tipo específico (su clase inmediata o interface). Por ejemplo, un manejador de excepción que sólo controle la excepción de índice no válido, tiene una sentencia catch como esta.

```

catch (InvalidIndexException e) {
    . . .
}

```

ArrayException es una clase nodo y representa cualquier error que pueda ocurrir durante la manipulación de un objeto array, incluyendo aquellos errores representados específicamente por una de sus subclases. Un método puede capturar una excepción basada en este grupo o tipo general especificando cualquiera de las superclases de la excepción en la sentencia catch. Por ejemplo, para capturar todas las excepciones de array, sin importar sus tipos específicos, un manejador de excepción especificaría un argumento ArrayException.

```

catch (ArrayException e) {

```

```
    . . .  
}
```

Este manejador podría capturar todas las excepciones de array, incluyendo `InvalidIndexException`, `ElementTypeException`, y `NoSuchElementException`. Se puede descubrir el tipo de excepción preciso que ha ocurrido comprobando el parámetro del manejador `e`. Incluso podríamos seleccionar un manejador de excepciones que controlara cualquier excepción con este manejador.

```
catch (Exception e) {  
    . . .  
}
```

Los manejadores de excepciones que son demasiado generales, como el mostrado aquí, pueden hacer que el código sea propenso a errores mediante la captura y manejo de excepciones que no se hubieran anticipado y por lo tanto no son manejadas correctamente dentro de manejador. Como regla no se recomienda escribir manejadores de excepciones generales.

Como has visto, se pueden crear grupos de excepciones y manejarlas de una forma general, o se puede especificar un tipo de excepción específico para diferenciar excepciones y manejarlas de un modo exacto.

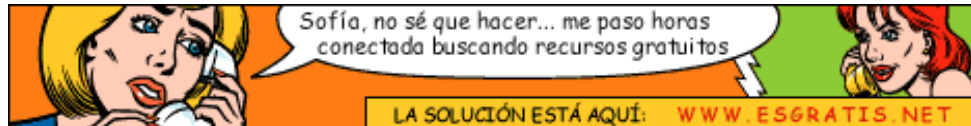
¿ Y ahora qué?

Ahora que has entendido qué son las excepciones y las ventajas de su utilización en los programas Java, es hora de aprender cómo utilizarlas.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Primer Encuentro con las Excepciones Java](#)

Primer Encuentro con las Excepciones Java

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException
must be caught, or it must be declared in throws clause of this method.
    fis = new FileInputStream(filename);
           ^
```

El mensaje de error anterior es uno de los dos mensajes similares que verás si intentas compilar la clase [InputFile](#), porque la clase `InputFile` contiene llamadas a métodos que lanzan excepciones cuando se produce un error. El lenguaje Java requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas desde dentro del ámbito de ese método. (Los detalles sobre lo que ocurre los puedes ver en la próxima página [Requerimientos de Java para Capturar o Especificar](#).)

Si el compilador detecta un método, como los de `InputFile`, que no cumplen este requerimiento, muestra un error como el anterior y no compila el programa.

Echemos un vistazo a `InputFile` en más detalle y veamos que sucede.

La clase `InputFile` envuelve un canal `FileInputStream` y proporciona un método, `getLine()`, para leer una línea en la posición actual del canal de entrada.

```
// Nota: Esta clase no se compila por diseño!
import java.io.*;

class InputFile {

    FileInputStream fis;

    InputFile(String filename) {
        fis = new FileInputStream(filename);
    }

    String getLine() {
        int c;
        StringBuffer buf = new StringBuffer();

        do {
            c = fis.read();
            if (c == '\n')                // nueva línea en UNIX
                return buf.toString();
            else if (c == '\r') {         // nueva línea en Windows 95/NT
                c = fis.read();
                if (c == '\n')
                    return buf.toString();
                else {
                    buf.append((char)'\r');
                    buf.append((char)c);
                }
            }
        } else
```

```

        buf.append((char)c);
    } while (c != -1);

    return null;
}
}

```

El compilador dará el primer error en la primera línea que está en negrita. Esta línea crea un objeto `FileInputStream` y lo utiliza para abrir un fichero (cuyo nombre se pasa dentro del constructor del `FileInputStream`).

Entonces, ¿Qué debe hacer el `FileInputStream` si el fichero no existe? Bien, eso depende de lo que quiera hacer el programa que utiliza el `FileInputStream`. Los implementadores de `FileInputStream` no tenían ni idea de lo que quiere hacer la clase `InputFile` si no existe el fichero. ¿Debe `FileInputStream` terminar el programa? ¿Debe intentar un nombre alternativo? o ¿debería crear un fichero con el nombre indicado? No existe un forma posible de que los implementadores de `FileInputStream` pudieran elegir una solución que sirviera para todos los usuarios de `FileInputStream`. Por eso ellos lanzaron una excepción. Esto es, si el fichero nombrado en el argumento del constructor de `FileInputStream` no existe, el constructor lanza una excepción `java.io.FileNotFoundException`. Mediante el lanzamiento de esta excepción, `FileInputStream` permite que el método llamador maneje ese error de la forma que considere más apropiada.

Como puedes ver en el listado, la clase `InputFile` ignora completamente el hecho de que el constructor de `FileInputStream` puede lanzar un excepción. Sin embargo, El lenguaje Java requiere que un método o bien lance o especifique todas las excepciones chequeadas que pueden ser lanzadas desde dentro de su ámbito. Como la Clase `InputFile` no hace ninguna de las dos cosas, el compilador rehusa su compilación e imprime el mensaje de error.

Además del primer error mostrado arriba, se podrá ver el siguiente mensaje de error cuando se compile la clase `InputFile`.

```

InputFile.java:15: Warning: Exception java.io.IOException must be caught,
or it must be declared in throws clause of this method.
    while ((c = fis.read()) != -1) {
                ^

```

El método `getLine()` de la clase `InputFile` lee una línea desde el `FileInputStream` que fue abierto por el constructor de `InputFile`. El método `read()` de `FileInputStream` lanza la excepción `java.io.IOException` si por alguna razón no pudiera leer el fichero. De nuevo, la clase `InputFile` no hace ningún intento por capturar o especificar esta excepción lo que se convierte en el segundo mensaje de error.

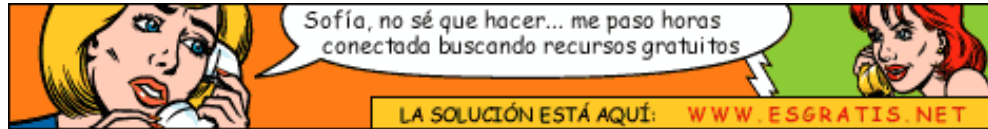
En este punto, tenemos dos opciones. Se puede capturar las excepciones con los métodos apropiados en la clase `InputFile`, o se puede esquivarlas y permitir que otros métodos anteriores en la pila de llamadas las capturen. De cualquier forma, los métodos de `InputFile` deben hacer algo, o capturar o especificar las excepciones, antes de poder compilar la clase `InputFile`. Aquí tiene la clase [`InputFileDeclared`](#), que corrige los errores de `InputFile` mediante la especificación de las excepciones.

La siguiente página le describe con más detalles los [Requerimientos de Java para Capturar o Especificar](#). Las páginas siguientes le enseñarán cómo cumplir estos requerimientos.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Requerimientos Java para Capturar o Especificar Excepciones](#)
 - [Capturar](#)
 - [Especificar](#)
 - [Excepciones Chequeadas](#)
 - [Excepciones que pueden ser lanzadas desde el ámbito de un método](#)

Requerimientos Java para Capturar o Especificar Excepciones

Como se mencionó anteriormente, Java requiere que un método o capture o especifique todas las excepciones chequeadas que se pueden lanzar dentro de su ámbito. Este requerimiento tiene varios componentes que necesitan una mayor descripción.

Capturar

Un método puede capturar una excepción proporcionando un manejador para ese tipo de excepción. La página siguiente, [Tratar con Excepciones](#), introduce un programa de ejemplo, le explica cómo capturar excepciones, y le muestra cómo escribir un manejador de excepciones para el programa de ejemplo.

Especificar

Si un método decide no capturar una excepción, debe especificar que puede lanzar esa excepción. ¿Por qué hicieron este requerimiento los diseñadores de Java? Porque una excepción que puede ser lanzada por un método es realmente una parte del interface de programación público del método: los llamadores de un método deben conocer las excepciones que ese método puede lanzar para poder decidir inteligente y concienzudamente qué hacer son esas excepciones. Así, en la firma del método debe especificar las excepciones que el método puede lanzar.

La siguiente página, [Tratar con Excepciones](#), le explica la especificación de excepciones que un método puede lanzar y le muestra cómo hacerlo.

Excepciones Chequeadas

Java tiene diferentes tipos de excepciones, incluyendo las excepciones de I/O, las excepciones en tiempo de ejecución, y las de su propia creación. Las que nos interesan a nosotros para esta explicación son las excepciones en tiempo de ejecución, Estas excepciones son aquellas que ocurren dentro del sistema de ejecución de Java. Esto incluye las excepciones aritméticas (como dividir por cero), excepciones de puntero (como intentar acceder a un objeto con una referencia nula), y excepciones de indexación (como intentar acceder a un elemento de un array con un índice que es muy grande o muy pequeño).

Las excepciones en tiempo de ejecución pueden ocurrir en cualquier parte de un programa y en un programa típico pueden ser muy numerosas. Muchas veces, el costo de chequear todas las excepciones en tiempo de ejecución excede de los beneficios de capturarlas o especificarlas. Así el compilador no requiere que se capturen o especifiquen estas excepciones, pero se puede hacer.

Las excepciones chequeadas son excepciones que no son excepciones en tiempo de ejecución y que son chequeadas por el compilador (esto es, el compilador comprueba que esas excepciones son capturadas o especificadas).

Algunas veces esto se considera como un bucle cerrado en el mecanismo de manejo de excepciones de Java y los programadores se ven tentados a convertir todas las excepciones en excepciones en tiempo de ejecución. En general, esto no está recomendado.

[La controversia -- Excepciones en Tiempo de Ejecución](#) contiene una explicación detallada sobre cómo utilizar las excepciones en tiempo de ejecución.

Excepciones que pueden ser lanzadas desde el ámbito de un método

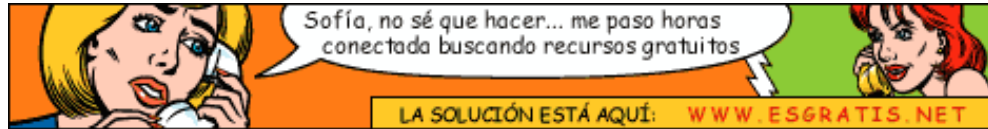
Esta sentencia podría parecer obvia a primera vista: sólo hay que fijarse en la sentencia `throw`. Sin embargo, esta sentencia incluye algo más no sólo las excepciones que pueden ser lanzadas directamente por el método: la clave está en la frase dentro del ámbito `de`. Esta frase incluye cualquier excepción que pueda ser lanzada mientras el flujo de control permanezca dentro del método. Así, esta sentencia incluye.

- excepciones que son lanzadas directamente por el método con la sentencia **`throw`** de Java, y
- las excepciones que son lanzadas por el método indirectamente a través de llamadas a otros métodos.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Tratar con las Excepciones Java](#)
 - [El ejemplo: ListOfNumbers](#)
 - [Capturar y Manejar Excepciones](#)
 - [Especificar las Excepciones que pueden ser Lanzadas por un Método](#)

Tratar con las Excepciones Java

[Primer Encuentro con las Excepciones de Java](#) describió brevemente cómo fue introducido en las excepciones Java: con un error del compilador indicando que las excepciones deben ser capturadas o especificadas. Luego [Requerimientos de Java para la Captura o Especificación](#) explicó qué significan exactamente los mensajes de error y por qué los diseñadores de Java decidieron hacer estos requerimientos. Ahora vamos a ver cómo capturar una excepción y cómo especificar otra.

[El ejemplo: ListOfNumbers](#)

Las secciones posteriores, sobre como capturar y especificar excepciones, utilizan el mismo ejemplo. Este ejemplo define e implementa un clase llamada ListOfNumbers. Esta clase llama a dos clases de los paquetes de Java que pueden lanzar excepciones. [Capturar y Manejar Excepciones](#) mostrará cómo escribir manejadores de excepciones para las dos excepciones, y [Especificar las Excepciones Lanzadas por un Método](#) mostrará cómo especificar esas excepciones en lugar de capturarlas.

[Capturar y Manejar Excepciones](#)

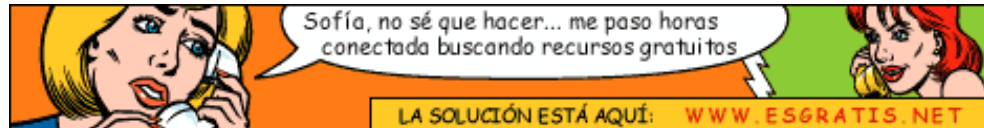
Una vez que te has familiarizado con la clase ListOfNumbers y con las excepciones que pueden ser lanzadas, puedes aprender cómo escribir manejadores de excepción que puedan capturar y manejar esas excepciones.

Esta sección cubre los tres componentes de una manejador de excepción -- los bloques try, catch, y finally -- y muestra cómo utilizarlos para escribir un manejador de excepción para el método writeList() de la clase ListOfNumbers. Además, esta sección contiene una página que pasea a lo largo del método writeList() y analiza lo que ocurre dentro del método en varios escenarios.

[Especificar las Excepciones que pueden ser Lanzadas por un Método](#)

Si no es apropiado que un método capture y maneje una excepción lanzada por un método que él ha llamado, o si el método lanza su propia excepción, debe especificar en la firma del método que éste puede lanzar una excepción. Utilizando la clase ListOfNumbers, esta sección le muestra cómo especificar las excepciones lanzadas por un método.





Manejo de Errores Usando Excepciones Java



En esta página:

- [El Ejemplo ListOfNumbers](#)

El Ejemplo ListOfNumbers

Las dos secciones siguientes que cubren la captura y especificación de excepciones utilizan este ejemplo.

```
import java.io.*;
import java.util.Vector;

class ListOfNumbers {
    private Vector vector;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        vector = new Vector(size);
        for (i = 0; i < size; i++)
            vector.addElement(new Integer(i));
    }

    public void writeList() {
        PrintStream pStr = null;

        System.out.println("Entering try statement");
        int i;
        pStr = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + vector.elementAt(i));

        pStr.close();
    }
}
```

Este ejemplo define e implementa una clase llamada `ListOfNumbers`. Sobre su construcción, esta clase crea un `Vector` que contiene diez elementos enteros con valores secuenciales del 0 al 9. Esta clase también define un método llamado `writeList()` que escribe los números de la lista en un fichero llamado "OutFile.txt".

El método `writeList()` llama a dos métodos que pueden lanzar excepciones. Primero la siguiente línea invoca al constructor de `FileOutputStream`, que lanza una excepción `IOException` si el fichero no puede ser abierto por cualquier razón.

```
pStr = new PrintStream(new BufferedOutputStream(new
FileOutputStream("OutFile.txt")));
```

Segundo, el método `elementAt()` de la clase `Vector` lanza una excepción `ArrayIndexOutOfBoundsException` si se le pasa un índice cuyo valor sea demasiado pequeño (un número negativo) o demasiado grande (mayor que el número de elementos que contiene realmente el `Vector`). Aquí está cómo `ListOfNumbers` invoca a `elementAt()`.


```
pStr.println("Value at: " + i + " = " + victor.elementAt(i));
```

Si se intenta compilar la clase ListOfNumbers, el compilador dará un mensaje de error sobre la excepción lanzada por el constructor de FileOutputStream, pero no muestra ningún error sobre la excepción lanzada por elementAt().

Esto es porque la excepción lanzada por FileOutputStream, es una excepción chequeada y la lanzada por elementAt() es una ejecución de tiempo de ejecución. Java sólo requiere que se especifiquen o capturen las excepciones chequeadas. Para más información, puedes ver [Requerimientos de Java para Capturar o Especificar](#).

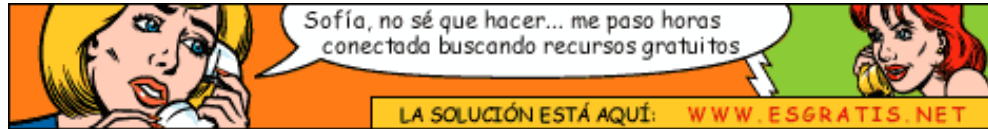
La siguiente sección, [Captura y Manejo de Excepciones](#), le mostrará cómo escribir un manejador de excepción para el método writeList() de ListOfNumbers.

Después de esto, una sección llamada [Especificar las Excepciones Lanzadas por un Método](#), mostrará cómo especificar que el método writeList() lanza excepciones en lugar de capturarlas.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Capturar y Manejar Excepciones](#)
 - [El Bloque try](#)
 - [Los bloques catch](#)
 - [El bloque finally](#)
 - [Poniéndolo Todo Junto](#)

Capturar y Manejar Excepciones

Las siguientes páginas muestran cómo construir un manejador de excepciones para el método `writeList()` descrito en [El ejemplo: ListOfNumbers](#). Las tres primeras páginas listadas abajo describen tres componentes diferentes de un manejador de excepciones y le muestran cómo pueden utilizarse esos componentes en el método `writeList()`. La cuarta página trata sobre el método `writeList()` resultante y analiza lo que ocurre dentro del código de ejemplo a través de varios escenarios.

[El Bloque try](#)

El primer paso en la escritura de un manejador de excepciones es poner la sentencia Java dentro de la cual se puede producir la excepción dentro de un bloque `try`. Se dice que el bloque `try` gobierna las sentencias encerradas dentro de él y define el ámbito de cualquier manejador de excepciones (establecido por el bloque `catch` subsecuente) asociado con él.

[Los bloques catch](#)

Después se debe asociar un manejador de excepciones con un bloque `try` proporcionándole uno o más bloques `catch` directamente después del bloque `try`.

[El bloque finally](#)

El bloque `finally` de Java proporciona un mecanismo que permite a sus métodos limpiarse a sí mismos sin importar lo que sucede dentro del bloque `try`. Se utiliza el bloque `finally` para cerrar ficheros o liberar otros recursos del sistema.

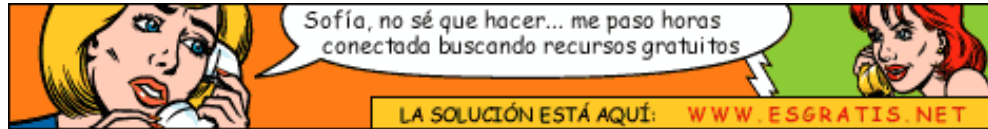
[Poniéndolo Todo Junto](#)

Las secciones anteriores describen cómo construir los bloques de código `try`, `catch`, y `finally` para el ejemplo `writeList()`. Ahora, pasearemos sobre el código para investigar que sucede en varios escenarios.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [El Bloque Try](#)

El Bloque Try

El primer paso en la construcción de un manejador de excepciones es encerrar las sentencias que podrían lanzar una excepción dentro de un bloque try. En general, este bloque se parece a esto.

```
try {  
    sentencias Java  
}
```

El segmento de código etiquetado sentencias java está compuesto por una o más sentencias legales de Java que podrían lanzar una excepción.

Para construir un manejador de excepción para el método writeList() de la clase ListOfNumbers, se necesita encerrar la sentencia que lanza la excepción en el método writeList() dentro de un bloque try.

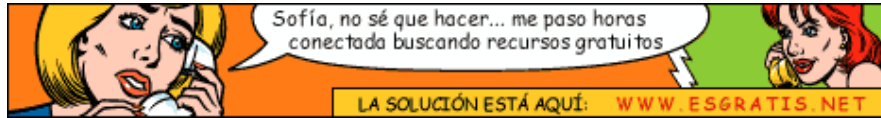
Existe más de una forma de realizar esta tarea. Podríamos poner cada una de las sentencias que potencialmente pudieran lanzar una excepción dentro de su propio bloque try, y proporcionar manejadores de excepciones separados para cada uno de los bloques try. O podríamos poner todas las sentencias de writeList() dentro de un sólo bloque try y asociar varios manejadores con él. El siguiente listado utiliza un sólo bloque try para todo el método porque el código tiende a ser más fácil de leer.

```
PrintStream pstr;  
  
try {  
    int i;  
  
    System.out.println("Entering try statement");  
    pStr = new PrintStream(  
        new BufferedOutputStream(  
            new FileOutputStream("OutFile.txt")));  
  
    for (i = 0; i < size; i++)  
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));  
}
```

Se dice que el bloque try gobierna las sentencias encerradas dentro del él y define el ámbito de cualquier manejador de excepción (establecido por su subsecuente bloque catch) asociado con él. En otras palabras, si ocurre una excepción dentro del bloque try, esta excepción será manejada por el manejador de excepción asociado con esta sentencia try.

Una sentencia try debe ir acompañada de al menos un bloque catch o un bloque finally.





Manejo de Errores Usando Excepciones Java



En esta página:

- [Los Bloques catch](#)
 - [Ocurre una IOException](#)
 - [Capturar Varios Tipos de Excepciones con Un Manejador](#)

Los Bloques catch

Como se aprendió en la [página anterior](#), la sentencia try define el ámbito de sus manejadores de excepción asociados. Se pueden asociar manejadores de excepción a una sentencia try proporcionando uno o más bloques catch directamente después del bloque try.

```
try {  
    . . .  
} catch ( . . . ) {  
    . . .  
} catch ( . . . ) {  
    . . .  
}
```

No puede haber ningún código entre el final de la sentencia try y el principio de la primera sentencia catch. La forma general de una sentencia catch en Java es esta.

```
catch (AlgunObjetoThrowable nombreVariable) {  
    Sentencias Java  
}
```

Como puedes ver, la sentencia catch requiere un sólo argumento formal. Este argumento parece un argumento de una declaración de método. El tipo del argumento AlgunObjetoThrowable declara el tipo de excepción que el manejador puede manejar y debe ser el nombre de una clase heredada de la clase [Throwable](#) definida en el paquete [java.lang](#). (Cuando los programas Java lanzan una excepción realmente están lanzando un objeto, sólo pueden lanzarse los objetos derivados de la clase Throwable. Aprenderás cómo lanzar excepciones en la lección [¿Cómo Lanzar Excepciones?.](#))

nombreVariable es el nombre por el que el manejador puede referirse a la excepción capturada. Por ejemplo, los manejadores de excepciones para el método writeList() (mostrados más adelante) llaman al método getMessage() de la excepción utilizando el nombre de excepción declarado e.

```
e.getMessage()
```

Se puede acceder a las variables y métodos de las excepciones en la misma forma que accede a los de cualquier otro objeto. getMessage() es un método proporcionado por la clase Throwable que imprime información adicional sobre el error ocurrido. La clase Throwable también implementa dos métodos para rellenar e imprimir el contenido de la pila de ejecución cuando ocurre la excepción. Las subclases de Throwable pueden añadir otros métodos o variables de ejemplar. Para buscar qué métodos implementar en una excepción, se puede comprobar la definición de la clase y las definiciones de las clases antecesoras.

El bloque catch contiene una serie de sentencias Java legales. Estas sentencias se ejecutan cuando se llama al manejador de excepción. El sistema de ejecución llama al manejador de excepción cuando el manejador es el primero en la pila de llamadas cuyo tipo coincide con el de la excepción lanzada.

El método writeList() de la clase de ejemplo ListOfNumbers utiliza dos manejadores de excepción para su sentencia try, con un manejador para cada uno de los tipos de excepciones que pueden lanzarse dentro del bloque try -- `ArrayIndexOutOfBoundsException` y `IOException`.

```
try {  
    . . .  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Caught IOException: " + e.getMessage());  
}
```

Ocurre una IOException

Supongamos que ocurre una excepción IOException dentro del bloque try. El sistema de ejecución

inmediatamente toma posesión e intenta localizar el manejador de excepción adecuado. El sistema de ejecución empieza buscando al principio de la pila de llamadas. Sin embargo, el constructor de `FileOutputStream` no tiene un manejador de excepción apropiado por eso el sistema de ejecución comprueba el siguiente método en la pila de llamadas -- el método `writeList()`. Este método tiene dos manejadores de excepciones: uno para `ArrayIndexOutOfBoundsException` y otro para `IOException`.

El sistema de ejecución comprueba los manejadores de `writeList()` por el orden en el que aparecen después del bloque `try`. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una `ArrayIndexOutOfBoundsException`, pero la excepción que se ha lanzado era una `IOException`. Una excepción `IOException` no puede asignarse legalmente a una `ArrayIndexOutOfBoundsException`, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de `writeList()` es una `IOException`. La excepción lanzada por el constructor de `FileOutputStream` también es una `IOException` y por eso puede ser asignada al argumento del manejador de excepciones de `IOException`. Así, este manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia.

```
Caught IOException: OutFile.txt
```

El sistema de ejecución sigue un proceso similar si ocurre una excepción `ArrayIndexOutOfBoundsException`. Para más detalles puedes ver.

Poniéndolo todo Junto que te lleva a través de método `writeList()` después de haberlo completado (queda un paso más) e investiga lo que sucede en varios escenarios.

Capturar Varios Tipos de Excepciones con Un Manejador

Los dos manejadores de excepción utilizados por el método `writeList()` son muy especializados. Cada uno sólo maneja un tipo de excepción. El lenguaje Java permite escribir manejadores de excepciones generales que pueden manejar varios tipos de excepciones.

Como ya sabes, las excepciones Java son objetos de la clase `Throwable` (son ejemplares de la clase `Throwable` a de alguna de sus subclases). Los paquetes Java contienen numerosas clases derivadas de la clase `Throwable` y así construyen un árbol de clases `Throwable`.

El manejador de excepción puede ser escrito para manejar cualquier clase heredada de `Throwable`. Si se escribe un manejador para una clase 'hoja' (una clase que no tiene subclases), se habrá escrito un manejador especializado: sólo maneja excepciones de un tipo específico. Si se escribe un manejador para una clase 'nodo' (una clase que tiene subclases), se habrá escrito un manejador general: se podrá manejar cualquier excepción cuyo tipo sea el de la clase nodo o de cualquiera de sus subclases.

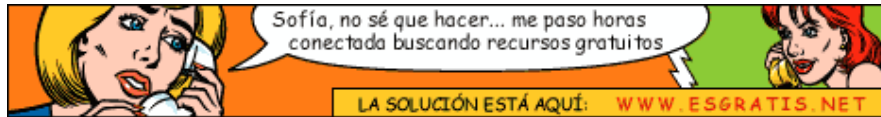
Modifiquemos de nuevo el método `writeList()`. Sólo esta vez, escribamoslo para que maneje las dos excepciones `IOException` y `ArrayIndexOutOfBoundsException`. El antecesor más cercano de estas dos excepciones es la clase `Exception`. Así un manejador de excepción que quisiera manejar los dos tipos se parecería a esto.

```
try {  
    . . .  
} catch (Exception e) {  
    System.err.println("Exception caught: " + e.getMessage());  
}
```

La clase `Exception` está bastante arriba en el árbol de herencias de la clase `Throwable`. Por eso, además de capturar los tipos de `IOException` y `ArrayIndexOutOfBoundsException` este manejador de excepciones, puede capturar otros muchos tipos. Generalmente hablando, los manejadores de excepción deben ser más especializados.

Los manejadores que pueden capturar la mayoría o todas las excepciones son menos utilizados para la recuperación de errores porque el manejador tiene que determinar qué tipo de excepción ha ocurrido de todas formas (para determinar la mejor estrategia de recuperación). Los manejadores de excepciones que son demasiado generales pueden hacer el código más propenso a errores mediante la captura y manejo de excepciones que no fueron anticipadas por el programador y para las que el manejador no está diseñado.





Manejo de Errores Usando Excepciones Java



En esta página:

- [El Bloque finally](#)
 - [¿Es realmente necesaria la sentencia finally?](#)

El Bloque finally

El paso final en la creación de un manejador de excepción es proporcionar un mecanismo que limpie el estado del método antes (posiblemente) de permitir que el control pase a otra parte diferente del programa. Se puede hacer esto encerrando el código de limpieza dentro de un bloque finally.

El bloque try del método `writeList()` ha estado trabajando con un `PrintStream` abierto. El programa debería cerrar ese canal antes de permitir que el control salga del método `writeList()`. Esto plantea un problema complicado, ya que el bloque try del `writeList()` tiene tres posibles salidas.

1. La sentencia `new FileOutputStream` falla y lanza una `IOException`.
2. La sentencia `vector.elementAt(i)` falla y lanza una `ArrayIndexOutOfBoundsException`.
3. Todo tiene éxito y el bloque `try` sale normalmente.

El sistema de ejecución siempre ejecuta las sentencias que hay dentro del bloque finally sin importar lo que suceda dentro del bloque try. Esto es, sin importar la forma de salida del bloque try del método `writeList()` debido a los escenarios 1, 2 ó 3 listados arriba, el código que hay dentro del bloque finally será ejecutado de todas formas.

Este es el bloque finally para el método `writeList()`. Limpia y cierra el canal `PrintStream`.

```
finally {
    if (pStr != null) {
        System.out.println("Closing PrintStream");
        pStr.close();
    } else {
        System.out.println("PrintStream not open");
    }
}
```

¿Es realmente necesaria la sentencia finally?

La primera necesidad de la sentencia finally podría no aparecer de forma inmediata. Los programadores se preguntan frecuentemente "¿Es realmente necesaria la sentencia finally o es sólo azúcar para mi Java?" En particular los programadores de C++ dudan de la necesidad de esta sentencia porque C++ no la tiene.

Esta necesidad de la sentencia finally no aparece hasta que se considera lo siguiente: ¿cómo se podría cerrar el `PrintStream` en el método `writeList()` si no se proporcionara un manejador de excepción para la `ArrayIndexOutOfBoundsException` y ocurre una `ArrayIndexOutOfBoundsException`? (sería sencillo y legal omitir un manejador de excepción para `ArrayIndexOutOfBoundsException` porque es una excepción en tiempo de ejecución y el compilador no alerta de que `writeList()` contiene una llamada a un método que puede lanzar una).

La respuesta es que el `PrintStream` no se cerraría si ocurriera una excepción `ArrayIndexOutOfBoundsException` y `writeList()` no proporcionara un manejador para ella -- a menos que `writeList()` proporcionara una sentencia finally.

Existen otros beneficios de la utilización de la sentencia finally. En el ejemplo de `writeList()` es posible proporcionar un código de limpieza sin la intervención de una sentencia finally. Por ejemplo, podríamos poner el código para cerrar el `PrintStream` al final del bloque try y de nuevo dentro del manejador de excepción para `ArrayIndexOutOfBoundsException`, como se muestra aquí.

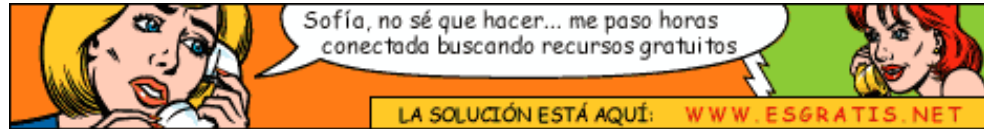
```
try {
    . . .
    pStr.close(); // No haga esto, duplica el código
} catch (ArrayIndexOutOfBoundsException e) {
    pStr.close(); // No haga esto, duplica el código
    System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```


Sin embargo, esto duplica el código, haciéndolo difícil de leer y propenso a errores si se modifica más tarde. Por ejemplo, si se añade código al bloque try que pudiera lanzar otro tipo de excepción, se tendría que recordar el cerrar el PrintStream dentro del nuevo manejador de excepción (lo que se olvidará seguro si se parece a mí).





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Poniéndolo todo Junto](#)
 - [Escenario 1: Ocurre una excepción IOException](#)
 - [Escenario 2: Ocurre una excepción ArrayIndexOutOfBoundsException](#)
 - [Escenario 3: El bloque try sale normalmente](#)

Poniéndolo todo Junto

Cuando se juntan todos los componentes, el método `writeList()` se parece a esto.

```
public void writeList() {
    PrintStream pStr = null;

    try {
        int i;

        System.out.println("Entrando en la Sentencia try");
        pStr = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: " +
e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (pStr != null) {
            System.out.println("Cerrando PrintStream");
            pStr.close();
        } else {
            System.out.println("PrintStream no está abierto");
        }
    }
}
```

El bloque `try` de este método tiene tres posibilidades de salida diferentes.

1. La sentencia **`new FileOutputStream`** falla y lanza una `IOException`.
2. La sentencia **`victor.elementAt(i)`** falla y lanza una `ArrayIndexOutOfBoundsException`.
3. Todo tiene éxito y la sentencia **`try`** sale normalmente.

Esta página investiga en detalle lo que sucede en el método `writeList` durante cada una de esas posibilidades de salida.

Escenario 1: Ocurre una excepción `IOException`

La sentencia `new FileOutputStream("OutFile.txt")` puede fallar por varias razones: el usuario no tiene permiso de escritura sobre el fichero o el directorio, el sistema de ficheros está lleno, o no existe el directorio. Si cualquiera de estas situaciones es verdadera el constructor de

FileOutputStream lanza una excepción IOException.

Cuando se lanza una excepción IOException, el sistema de ejecución para inmediatamente la ejecución del bloque try. Y luego intenta localizar un manejador de excepción apropiado para manejar una IOException.

El sistema de ejecución comienza su búsqueda al principio de la pila de llamadas. Cuando ocurrió la excepción, el constructor de FileOutputStream estaba al principio de la pila de llamadas. Sin embargo, este constructor no tiene un manejador de excepción apropiado por lo que el sistema comprueba el siguiente método que hay en la pila de llamadas -- el método writeList(). Este método tiene dos manejadores de excepciones: uno para ArrayIndexOutOfBoundsException y otro para IOException.

El sistema de ejecución comprueba los manejadores de writeList() por el orden en el que aparecen después del bloque try. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una ArrayIndexOutOfBoundsException, pero la excepción que se ha lanzado era una IOException. Una excepción IOException no puede asignarse legalmente a una ArrayIndexOutOfBoundsException, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de writeList() es una IOException. La excepción lanzada por el constructor de FileOutputStream también es una IOException y por eso puede ser asignada al argumento del manejador de excepciones de IOException. Así, este manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia.

```
Caught IOException: OutFile.txt
```

Después de que se haya ejecutado el manejador de excepción, el sistema pasa el control al bloque finally. En este escenario particular, el canal PrintStream nunca se ha abierto, así el pStr es null y no se cierra. Después de que se haya completado la ejecución del bloque finally, el programa continúa con la primera sentencia después de este bloque.

La salida completa que se podrá ver desde el programa ListOfNumbers cuando se lanza un excepción IOException es esta.

```
Entrando en la sentencia try
Caught IOException: OutFile.txt
PrintStream no está abierto
```

Escenario 2: Ocurre una excepción ArrayIndexOutOfBoundsException

Este escenario es el mismo que el primero excepto que ocurre un error diferente dentro del bloque try. En este escenario, el argumento pasado al método elementAt() de Vector está fuera de límites. Esto es, el argumento es menor que cero o mayor que el tamaño del array. (De la forma en que está escrito el código, esto es realmente imposible, pero supongamos que se ha introducido un error cuando alguien lo ha modificado).

Como en el escenario 1, cuando ocurre una excepción el sistema de ejecución para la ejecución del bloque try e intenta localizar un manejador de excepción apropiado para ArrayIndexOutOfBoundsException. El sistema busca como lo hizo anteriormente. Llega a la sentencia catch en el método writeList() que maneja excepciones del tipo ArrayIndexOutOfBoundsException. Como el tipo de la excepción corresponde con el de el manejador, el sistema ejecuta el manejador de excepción.

Después de haber ejecutado el manejador de excepción, el sistema pasa el control al bloque finally. En este escenario particular, el canal PrintStream si que se ha abierto, así que el bloque finally lo cerrará. Después de que el bloque finally haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida completa que dará el programa ListOfNumbers si ocurre una excepción ArrayIndexOutOfBoundsException.

```
Entrando en la sentencia try
Caught ArrayIndexOutOfBoundsException: 10 >= 10
Cerrando PrintStream
```

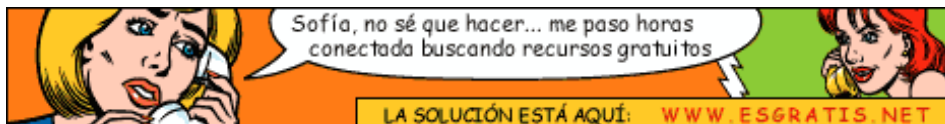
Escenario 3: El bloque try sale normalmente

En este escenario, todas las sentencias dentro del ámbito del bloque try se ejecutan de forma satisfactoria y no lanzan excepciones. La ejecución cae al final del bloque try y el sistema pasa el control al bloque finally. Como todo ha salido satisfactorio, el PrintStream abierto se cierra cuando el bloque finally consigue el control. De nuevo, Después de que el bloque finally haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida cuando el programa ListOfNumbers cuando no se lanzan excepciones.

Entrando en la sentencia try
Cerrando PrintStream





Manejo de Errores Usando Excepciones Java



En esta página:

- [Especificar las Excepciones Lanzadas por un Método](#)

Especificar las Excepciones Lanzadas por un Método

Le sección anterior mostraba como escribir un manejador de excepción para el método `writeList()` de la clase `ListOfNumbers`. Algunas veces, es apropiado capturar las excepciones que ocurren pero en otras ocasiones, sin embargo, es mejor dejar que un método superior en la pila de llamadas maneje la excepción. Por ejemplo, si se está utilizando la clase `ListOfNumbers` como parte de un paquete de clases, probablemente no se querrá anticipar las necesidades de todos los usuarios de su paquete. En este caso, es mejor no capturar las excepciones y permitir que alguien la capture más arriba en la pila de llamadas.

Si el método `writeList()` no captura las excepciones que pueden ocurrir dentro de él, debe especificar que puede lanzar excepciones. Modifiquemos el método `writeList()` para especificar que puede lanzar excepciones. Como recordatorio, aquí tienes la versión original del método `writeList()`.

```
public void writeList() {
    System.out.println("Entrando en la sentencia try");
    int i;
    pStr = new PrintStream(
        new BufferedOutputStream(
            new FileOutputStream("OutFile.txt")));

    for (i = 0; i < size; i++)
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));
}
```

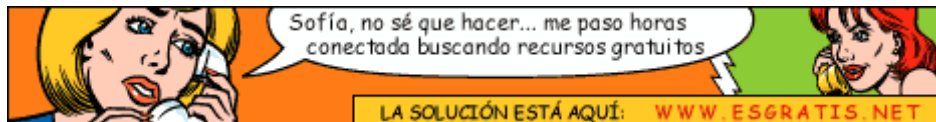
Como recordarás, la sentencia `new FileOutputStream("OutFile.txt")` podría lanzar un excepción `IOException` (que no es una excepción en tiempo de ejecución). La sentencia `victor.elementAt(i)` puede lanzar una excepción `ArrayIndexOutOfBoundsException` (que es una subclase de la clase `RuntimeException`, y es una excepción en tiempo de ejecución).

Para especificar que `writeList()` lanza estas dos excepciones, se añade la cláusula `throws` a la firma del método de `writeList()`. La cláusula `throws` está compuesta por la palabra clave `throws` seguida por una lista separada por comas de todas las excepciones lanzadas por el método. Esta cláusula va después del nombre del método y antes de la llave abierta que define el ámbito del método. Aquí tienes un ejemplo.

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
```

Recuerda que la excepción `ArrayIndexOutOfBoundsException` es una excepción en tiempo de ejecución, por eso no tiene porque especificarse en la sentencia `throws` pero puede hacerse si se quiere.





Manejo de Errores Usando Excepciones Java



En esta página:

- [La Sentencias throw](#)
 - [La clausula throws](#)

La Sentencias throw

Todos los métodos Java utilizan la sentencia throw para lanzar una excepción.

Esta sentencia requiere un sólo argumento, un objeto Throwable. En el sistema Java, los objetos lanzables son ejemplares de la clase [Throwable](#) definida en el paquete [java.lang](#). Aquí tienes un ejemplo de la sentencia throw.

```
throw algunObjetoThrowable;
```

Si se intenta lanzar un objeto que no es 'lanzable', el compilador rehusa la compilación del programa y muestra un mensaje de error similar a éste.

```
testing.java:10: Cannot throw class java.lang.Integer; it must be a subclass
of class java.lang.Throwable.
    throw new Integer(4);
    ^
```

La página siguiente, [La clase Throwable y sus Subclases](#), cuentan más cosas sobre la clase Throwable.

Echemos un vistazo a la sentencia throw en su contexto. El siguiente método está tomado de una clase que implementa un objeto pila normal. El método pop() saca el elemento superior de la pila y lo devuelve.

```
public Object pop() throws EmptyStackException {
    Object obj;

    if (size == 0)
        throw new EmptyStackException();

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

El método pop() comprueba si hay algún elemento en la pila. Si la pila está vacía (su tamaño es igual a cero), ejemplariza un nuevo objeto de la clase EmptyStackException y lo lanza. Esta clase está definida en el paquete java.util. En páginas posteriores podrás ver cómo crear tus propias clases de excepciones. Por ahora, todo lo que necesitas recordar es que se pueden lanzar objetos heredados desde la clase Throwable.

La clausula throws

Habrás observado que la declaración del método pop() contiene esta clausula.

```
throws EmptyStackException
```

La clausula throws especifica que el método puede lanzar una excepción EmptyStackException. Como ya sabes, el lenguaje Java requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas dentro de su ámbito.

Se puede hacer esto con la clausula throws de la declaración del método.

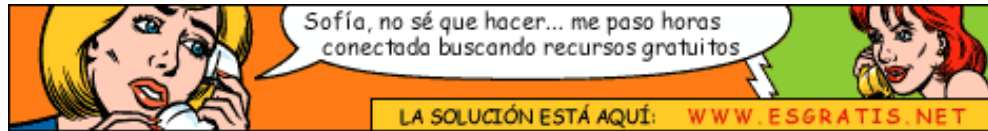
Para más información sobre estos requerimientos puedes ver [Requerimientos Java para Capturar o Especificar](#).

También puedes ver, [Especificar las Excepciones lanzadas por un Método](#) para obtener más detalles sobre cómo un método puede lanzar excepciones.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [La Clase Throwable y sus Subclases](#)
 - [Error](#)
 - [Exception](#)
 - [Excepciones en Tiempo de Ejecución](#)

La Clase Throwable y sus Subclases

Como se aprendió en la página anterior, sólo se pueden lanzar objetos que estén derivados de la clase Throwable. Esto incluye descendientes directos (esto es, objetos de la clase Throwable) y descendiente indirectos (objetos derivados de hijos o nietos de la clase Throwable).

Este diagrama ilustra el árbol de herencia de la clase Throwable y sus subclases más importantes.



Como se puede ver en el diagrama, la clase Throwable tiene dos descendientes directos: Error y Exception.

Error

Cuando falla un enlace dinámico, y hay algún fallo "hardware" en la máquina virtual, ésta lanza un error. Típicamente los programas Java no capturan los Errores. Pero siempre lanzarán errores.

Exception

La mayoría de los programas lanzan y capturan objetos derivados de la clase Exception.

Una Excepción indica que ha ocurrido un problema pero que el problema no es demasiado serio.

La mayoría de los programas que escribirás lanzarán y capturarán excepciones.

La clase Exception tiene muchos descendientes definidos en los paquetes Java. Estos descendientes indican varios tipos de excepciones que pueden ocurrir. Por ejemplo, `IllegalAccessException` señala que no se puede encontrar un método particular, y `NegativeArraySizeException` indica que un programa intenta crear un array con tamaño negativo.

Una subclase de Exception tiene un significado especial en el lenguaje Java: `RuntimeException`.

Excepciones en Tiempo de Ejecución

La clase `RuntimeException` representa las excepciones que ocurren dentro de la máquina virtual Java (durante el tiempo de ejecución). Un ejemplo de estas excepciones es `NullPointerException`, que ocurre cuando un método intenta acceder a un miembro de un objeto a través de una referencia nula. Esta excepción puede ocurrir en cualquier lugar en que un programa intente desreferenciar una referencia a un objeto. Frecuentemente el coste de chequear estas excepciones sobrepasa los beneficios de capturarlas.

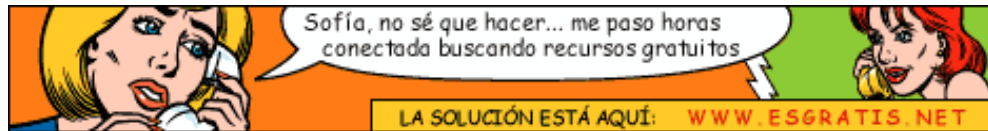
Como las excepciones en tiempo de ejecución están omnipresentes e intentar capturar o especificarlas todas en todo momento podría ser un ejercicio infructuoso (y un código infructuoso, imposible de leer y de mantener), el compilador permite que estas excepciones no se capturen ni se especifiquen.

Los paquetes Java definen varias clases `RuntimeException`. Se pueden capturar estas excepciones al igual que las otras. Sin embargo, no se requiere que un método especifique que lanza excepciones en tiempo de ejecución. Además puedes crear sus propias subclases de `untimeException`.

[Excepciones en Tiempo de Ejecución -- La Controversia](#) contiene una explicación detallada sobre cómo utilizar las excepciones en tiempo de ejecución.



TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Crear Clases de Excepciones](#)
 - [¿Qué puede ir mal?](#)
 - [Elegir el Tipo de Excepción Lanzada](#)
 - [Elegir una Superclase](#)
 - [Convenciones de Nombres](#)

Crear Clases de Excepciones

Cuando diseñes un paquete de clases java que colabore para proporcionar alguna función útil a sus usuarios, deberás trabajar duro para asegurarte de que las clases interactúan correctamente y que sus interfaces son fáciles de entender y utilizar. Deberías estar mucho tiempo pensando sobre ello y diseñar las excepciones que esas clases pueden lanzar.

Supon que estás escribiendo una clase con una lista enlazada que estás pensando en distribuir como freeware. Entre otros métodos la clase debería soportar estos.

objectAt(int n)

Devuelve el objeto en la posición **n** de la lista.

firstObject()

Devuelve el primer objeto de la lista.

indexOf(Object o)

Busca el Objeto especificado en la lista y devuelve su posición en ella.

¿Qué puede ir mal?

Como muchos programadores utilizarán tu clase de lista enlazada, puedes estar seguro de que muchos de ellos la utilizarán mal o abusarán de los métodos de la clase. También, alguna llamada legítima a los métodos de la clase podría dar algún resultado indefinido. No importa, con respecto a los errores, querrá que tu clase sea lo más robusta posible, para hacer algo razonable con los errores, y comunicar los errores al programa llamador. Sin embargo, no puedes anticipar como quiere cada usuario de sus clase enlazada que se comporten sus objetos ante la adversidad. Por eso, lo mejor que puedes hacer cuando ocurre un error es lanzar una excepción.

Cada uno de los métodos soportados por la lista enlazada podría lanzar una excepción bajo ciertas condiciones, y cada uno podría lanzar un tipo diferente de excepción. Por ejemplo.

objectAt()

lanzará una excepción si se pasa un entero al método que sea menor que 0 o mayor que el número de objetos que hay realmente en la lista.

firstObject()

lanzará una excepción si la lista no contiene objetos.

indexOf()

lanzará una excepción si el objeto pasado al método no está en la lista.

Pero ¿qué tipo de excepción debería lanzar cada método? ¿Debería ser una excepción proporcionada por el entorno de desarrollo de Java? O ¿Deberían ser excepciones propias?

Elegir el Tipo de Excepción Lanzada

Tratándose de la elección del tipo de excepción a lanzar, tienes dos opciones.

1. Utilizar una escrita por otra persona. Por ejemplo, el entorno de desarrollo de Java proporciona muchas clases de excepciones que podrías utilizar.

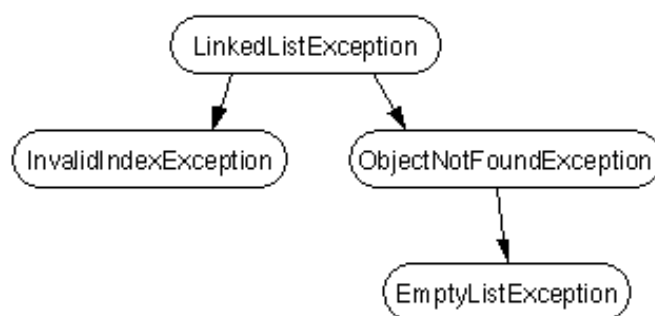
2. Escribirlas tu mismo.

Necesitarás escribir tus propias clases de excepciones si respondes "Si" a alguna de las siguientes preguntas. Si no es así, probablemente podrás utilizar alguna excepción ya escrita.

- ¿Necesitas un tipo de excepción que no está representada por lo existentes en el entorno de desarrollo de Java?
- ¿Ayudaría a sus usuarios si pudieran diferenciar sus excepciones de las otras lanzadas por clases escritas por otros vendedores?
- ¿Lanza el código más una excepción relacionada?
- Si utilizas excepciones de otros, ¿Podrán sus usuarios tener acceso a estas excepciones? Una pregunta similar es "¿Debería tu paquete ser independiente y auto-contenedor?"

La clase de lista enlazada puede lanzar varias excepciones, y sería conveniente poder capturar todas las excepciones lanzadas por la lista enlazada con un manejador. Si planeas distribuir la lista enlazada en un paquete, todo el código relacionado debe empaquetarse junto. Así para la lista enlazada, deberías crear tu propio árbol de clases de excepciones.

El siguiente diagrama ilustra una posibilidad del árbol de clases para su lista enlazada.



LinkedListException es la clase padre de todas las posibles excepciones que pueden ser lanzadas por la clase de la lista enlazada. Los usuarios de esta clase pueden escribir un sólo manejador de excepciones para manejarlas todas con una sentencia catch como esta.

```
catch (LinkedListException) {
    . . .
}
```

O, podrías escribir manejadores más especializados para cada una de las subclases de LinkedListException.

Elegir una Superclase

El diagrama anterior no indica la superclase de la clase LinkedListException. Como ya sabes, las excepciones de Java deben ser ejemplares de la clase Throwable o de sus subclases.

Por eso podría tentarte hacer LinkedListException como una subclase de la clase Throwable.

Sin embargo, el paquete java.lang proporciona dos clases Throwable que dividen los tipos de problemas que pueden ocurrir en un programa java: Errores y Excepción. La mayoría de los applets y de las aplicaciones que escribes lanzan objetos que son Excepciones. (Los errores están reservados para problemas más serios que pueden ocurrir en el sistema.)

Teóricamente, cualquier subclase de Exception podría ser utilizada como padre de la clase LinkedListException. Sin embargo, un rápido examen de esas clases muestra que o son demasiado especializadas o no están relacionadas con LinkedListException para ser apropiadas.

Así que el padre de la clase LinkedListException debería ser Exception.

Como las excepciones en tiempo de ejecución no tienen por qué ser especificadas en la cláusula throws de un método, muchos desarrolladores de paquetes se preguntan.

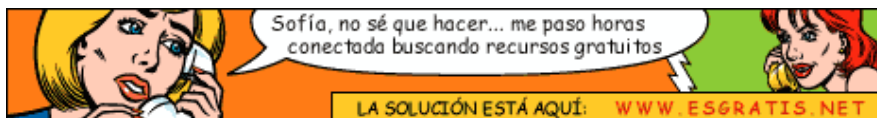
"¿No es más sencillo hacer que todas mis excepciones sean heredadas de Runtime Exception?" La respuesta a esta pregunta con más detalle en [Excepciones en Tiempo de Ejecución -- La Controversia](#). La línea inferior dice que no deberías utilizar subclases de RuntimeException en tus clases a menos que tus excepciones sean realmente en tiempo de ejecución! Para la mayoría de nosotros, esto significa "NO, tus excepciones no deben descender de la clase RuntimeException."

Convenciones de Nombres

Es una buena práctica añadir la palabra "Exception" al final del nombre de todas las clases heredadas (directa o indirectamente) de la clase Exception. De forma similar, los nombres de las

clases que se hereden desde la clase Error deberían terminar con la palabra "Error".





Manejo de Errores Usando Excepciones Java



En esta página:

- [Excepciones en Tiempo de Ejecución - La Controversia](#)

Excepciones en Tiempo de Ejecución - La Controversia

Como el lenguaje Java no requiere que los métodos capturen o especifiquen las excepciones en tiempo de ejecución, es una tentación para los programadores escribir código que lance sólo excepciones de tiempo de ejecución o hacer que todas sus subclases de excepciones hereden de la clase `RuntimeException`. Estos atajos de programación permiten a los programadores escribir código Java sin preocuparse por los consiguientes.

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException must be caught,
or it must be declared in throws clause of this method.
```

```
    fis = new FileInputStream(filename);
    ^
```

errores del compilador y sin preocuparse por especificar o capturar ninguna excepción.

Mientras esta forma parece conveniente para los programadores, esquiva los requerimientos de Java de capturar o especificar y pueden causar problemas a los programadores que utilicen tus clases.

¿Por qué decidieron los diseñadores de Java forzar a un método a especificar todas las excepciones chequeadas no capturadas que pueden ser lanzadas dentro de su ámbito?

Como cualquier excepción que pueda ser lanzada por un método es realmente una parte del interface de programación público del método: los llamadores de un método deben conocer las excepciones que el método puede lanzar para poder decidir concienzuda e inteligentemente qué hacer con estas excepciones. Las excepciones que un método puede lanzar son como una parte del interface de programación del método como sus parámetros y devuelven un valor.

La siguiente pregunta podría ser: "Bien ¿Si es bueno documentar el API de un método incluyendo las excepciones que pueda lanzar, por qué no especificar también las excepciones de tiempo de ejecución?".

Las excepciones de tiempo de ejecución representan problemas que son detectados por el sistema de ejecución. Esto incluye excepciones aritméticas (como la división por cero), excepciones de punteros (como intentar acceder a un objeto con un referencia nula), y las excepciones de indexación (como intentar acceder a un elemento de un array a través de un índice demasiado grande o demasiado pequeño).

Las excepciones de tiempo de ejecución pueden ocurrir en cualquier lugar del programa y en un programa típico pueden ser muy numerosas. Típicamente, el coste del chequeo de las excepciones de tiempo de ejecución excede de los beneficios de capturarlas o especificarlas.

Así el compilador no requiere que se capturen o especifiquen las excepciones de tiempo de ejecución, pero se puede hacer.

Las excepciones chequeadas representan información útil sobre la operación legalmente especificada sobre la que el llamador podría no tener control y el llamador necesita estar informado sobre ella -- por ejemplo, el sistema de ficheros está lleno, o el ordenador remoto ha cerrado la conexión, o los permisos de acceso no permiten esta acción.

¿Qué se consigue si se lanza una excepción `RuntimeException` o se crea una subclase de `RuntimeException` sólo porque no se quiere especificarla? Simplemente, se obtiene la posibilidad de lanzar una excepción sin especificar lo que se está haciendo. En otras palabras, es una forma de evitar la documentación de las excepciones que puede lanzar un método.

¿Cuándo es bueno esto? Bien, ¿cuándo es bueno evitar la documentación sobre el comportamiento de los métodos? La respuesta es "NUNCA".

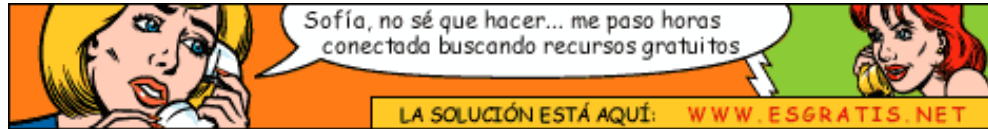
Reglas del Pulgar:

- Se puede detectar y lanzar una excepción de tiempo de ejecución cuando se encuentra un error en la máquina virtual, sin embargo, es más sencillo dejar que la máquina virtual lo detecte y lo lance. Normalmente, los métodos que escribas lanzarán excepciones del tipo `Exception`, no del tipo `RuntimeException`.
- De forma similar, puedes crear una subclase de `RuntimeException` cuando estas creando un error en la máquina virtual (que probablemente no lo hará), De otro modo utilizará la clase `Exception`.
- No lances una excepción en tiempo de ejecución o crees una subclase de `RuntimeException` simplemente porque no quieres preocuparte de especificarla.





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Cambios en el JDK 1.1 que afectan a las Excepciones](#)

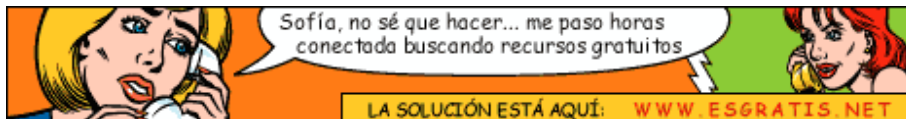
Cambios en el JDK 1.1 que afectan a las Excepciones

[Manejo de Errores Utilizando Excepciones](#) Aunque el mecanismo básico del manejo de excepciones no ha cambiado en el JDK 1.1, se han añadido muchas nuevas clases de excepciones y errores. Comprueba la [Documentación online del API](#).

[El ejemplo ListOfNumbers](#) El ejemplo ListOfNumbers utiliza métodos obsoletos. Puedes ver [Cambios en el JDK 1.1: El ejemplo ListOfNumbers](#)

[La clase Throwable y sus subclases](#) El JDK 1.1 requiere algún cambio menor en la clase Throwable debido a la internacionalización. Puedes ver: [Cambios en el JDK 1.1: la Clase Throwable](#)





Manejo de Errores Usando Excepciones Java



En esta página:

- [Cambios en el ejemplo ListOfNumbers](#)

Cambios en el ejemplo ListOfNumbers

El ejemplo [ListOfNumbers](#) escribe su salida en un `PrintStream`. Crear un objeto `PrintStream` está desfasado en el JDK 1.1 para asegurarse de que los programadores utilizan una nueva clase, `PrintWriter`, en vez de un `PrintStream`.

La clase `PrintWriter` es muy similar a la clase `PrintStream`, y en el caso del ejemplo `ListOfNumbers`, se puede utilizar un `PrintWriter` en vez de un `PrintStream` sin otros cambios en el código. Aquí tienes una versión del JDK 1.1 del ejemplo `ListOfNumbers`.

```
import java.io.*;
import java.util.Vector;

class ListOfNumbers {
    private Vector vector;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        vector = new Vector(size);
        for (i = 0; i < size; i++)
            vector.addElement(new Integer(i));
    }

    public void writeList() {
        PrintWriter pWriter = null;

        try {
            int i;

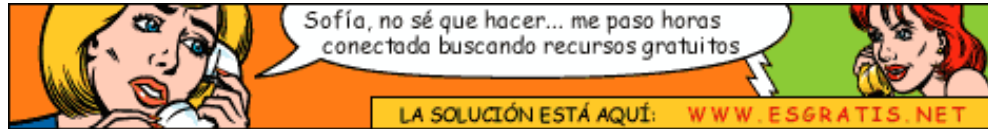
            System.out.println("Entering try statement");
            pWriter = new PrintWriter(
                new BufferedOutputStream(
                    new FileOutputStream("OutFile.txt")));

            for (i = 0; i < size; i++)
                pWriter.println("Value at: " + i + " = " + vector.elementAt(i));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Caught ArrayIndexOutOfBoundsException: " +
e.getMessage());
        } catch (IOException e) {
            System.err.println("Caught IOException: " + e.getMessage());
        } finally {
            if (pWriter != null) {
                System.out.println("Closing PrintWriter");
                pWriter.close();
            } else {
                System.out.println("PrintWriter not open");
            }
        }
    }
}
```





TutorJava recomienda...



Manejo de Errores Usando Excepciones Java



En esta página:

- [Cambios en la Clase Throwable](#)
 - [Nuevos Métodos](#)

Cambios en la Clase Throwable

La versión 1.1 del JDK añade muchas características que hacen sencillo para los programadores la creación de programas internacionalizados. También se han hecho necesarios algunos cambios en la clase Throwable para convertirla en un ciudadano internacional. De hecho, todos los cambios hechos en esta clase están relacionados con la internacionalización.

Nuevos Métodos

Estos métodos se han añadido a la clase Throwable en el JDK 1.1.

```
String getLocalizedMessage()  
printStackTrace(PrintWriter)
```

