Unidad 3

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 6

Excepciones y Aserciones

Excepciones y Aserciones

En Este Capítulo

- Introducción
- Excepciones
- Tipos de Excepciones
- Categorías
- Requerimientos de Java para Capturar o Especificar Excepciones
- Capturar y Manejar Excepciones
- Los bloques catch
- Manejo de sentencias try – catch
- Rescritura y Excepciones
- Creación de excepciones personalizadas
- Aserciones o aseveraciones
- Usos recomendados de las aserciones

Introducción

Las excepciones son un mecanismo utilizado por numerosos lenguajes de programación para describir lo que debe hacerse cuando ocurre algo inesperado. En general, algo inesperado suele ser algún tipo de error, por ejemplo, la llamada a un método con argumentos no válidos, el fallo de una conexión de red o la solicitud de apertura de un archivo que no existe.

Las aserciones son una forma de verificar ciertos supuestos sobre la lógica de un programa. Por ejemplo, si cree que, en un determinado punto, el valor de una variable siempre será positivo, una aserción puede comprobar si esto es verdad. Las aserciones suelen utilizarse para verificar supuestos sobre la lógica local dentro de un método y no para comprobar si se cumplen las expectativas externas.

Un aspecto importante de las aserciones es que pueden suprimirse enteramente al ejecutar el código. Esto permite habilitar las aserciones durante el desarrollo del programa, pero evitar la ejecución de las pruebas en el tiempo de ejecución, cuando el producto final se entrega al cliente.

Ésta es una diferencia importante entre aserciones y excepciones.

Excepciones

> *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, a los simples errores de programación, como tratar de acceder a un elemento de un vector fuera de sus límites. Cuando dicho error ocurre dentro de un método Java, el método crea un objeto del tipo Exception (aunque los errores graves y de máquina virtual generan un Error) y lo maneja en un sistema de ejecución especial dedicado a este fin dentro del lenguaje. 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 de Java, crear un objeto del tipo 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. Para ello se recorre en sentido inverso aquellos métodos que se encuentran en ejecución desde que comenzó el programa para encontrar el manejador apropiado. Esto es posible porque desde el comienzo del programa cada método invocado reserva espacio en el stack formando la denominada “pila de llamados”, la cual sirve de “lista” para recorrerla inversamente en busca del código que maneje la excepción. Si un método no maneja la excepción que acaba de ocurrir, simplemente se recorre en forma inversa el stack para revisar si el método anterior al que se acaba de revisar contiene el código que lo maneje. En caso de encontrarlo se va al siguiente y así sucesivamente hasta llegar a main. Si en main, el cual es el primer método que reserva espacio en el stack, no se encuentra el código que maneja la

excepción, el programa termina indicando por consola el motivo del error y el recorrido inverso que realizo en la pila de llamados indicando los métodos analizados en busca del código.

El código que maneja la excepción se considera adecuado para su manejo por la comparación de la variable de referencia que dicho código recibe como argumento. Esta variable debe ser del mismo tipo que el error que acaba de ocurrir. Esto quiere decir que la variable de referencia declarada deberá ser igual al objeto del tipo excepción ocurrida o al menos igual al de algún objeto del tipo Exception que se encuentre antecediéndolo en la misma cadena de herencia.

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 que "capture la excepción"), 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 convierten al código en algo muy complicado. Por ejemplo, suponiendo que existe una función que lee un archivo completo dentro de la memoria. En seudo código, la función se podría parecer a esto.

Ejemplo

```
1. leerArchivo {
2.        abrir el archivo;
3.        determinar su tamaño;
4.        asignar suficiente memoria;
5.        leer el archivo a la memoria;
6.        cerrar el archivo;
7. }
```

A primera vista esta función parece bastante sencilla, pero ignora todos aquellos errores potenciales, por ejemplo:

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

Para responder a estas cuestiones dentro de la función, se debería añadir mucho código para la detección y el manejo de errores. El aspecto final de la función se parecería a lo siguiente:

Ejemplo

```
1. codigodeError leerArchivo {
2. inicializar codigodeError = 0;
3. abrir el archivo;
4. if (archivoAbierto) {
5.              determinar la longitud del archivo;
6.              if (obtenerLongitudDelArchivo) {
7.                     asignar suficiente memoria;
8.                     if (obtenerSuficienteMemoria) {
9.                            leer el archivo a memoria;
10.                           if (falloDeLectura) {
11.                                  codigodeError = -1;
12.                           } else {
13.                                  codigodeError = -2;
14.                           }
15.                    } else {
16.                           codigodeError = -3;
17.                    }
18.                    cerrar el archivo;
19.                    if (archivoNoCerrado && codigodeError == 0) {
20.                           codigodeError = -4;
21.                    } else {
22.                           codigodeError = codigodeError and -4;
23.                    }
24.             } else {
25.                    codigodeError = -5;
26.             }
27.             return codigodeError;
28.      }
29.}
```

Con la detección de errores, las 7 líneas originales se han convertido en 29 líneas de código, por lo tanto ha aumentado más de un 400 %. Lo peor, existe tanta detección y manejo de errores y de retorno respecto a las 7 líneas originales que el código está totalmente enrarecido. Y aún peor, el flujo lógico del código también se pierde, haciendo difícil poder decidir si el código hace lo correcto (si se cierra el archivo realmente o si falla la asignación de memoria) e incluso es difícil asegurar que el código continúe haciendo las cosas correctas cuando se modifique el método tres meses después de haberlo escrito.

Muchos programadores "resuelven" este problema ignorándolo, motivo por el cual se informa de los errores cuando el programa no funciona. Sin embargo esto viola directamente un principio básico de las “buenas prácticas” en la programación orientada a objetos:

Toda clase debe ser capaz de manejar los errores que ocurren dentro de ella

Esto tiene varias consecuencias en la codificación, porque si un método no maneja el error dentro de él, se vuelve en reversa sobre el stack, lo cual no asegura que el próximo código a evaluar para el manejo del error se encuentre en el mismo objeto o dentro de otro que simplemente llamó al método donde ocurrió el error

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 el método leerArchivo utilizara excepciones en lugar de las técnicas de manejo de errores tradicionales se podría parecer a esto:

Ejemplo

```
1. leerArchivo {
2.       try {
3.             abrir el archivo;
4.             determinar su tamaño;
5.             asignar suficiente memoria;
6.             leer el archivo a la memoria;
7.             cerrar el archivo;
8.       } catch (falloAbrirArchivo) {
9.             hacerAlgo;
10.      } catch (falloDeterminacionTamaño) {
11.            hacerAlgo;
12.      } catch (falloAsignaciondeMemoria) {
13.            hacerAlgo;
14.      } catch (falloLectura) {
15.            hacerAlgo;
16.      } catch (falloCerrarArchivo) {
17.            hacerAlgo;
18.      }
19.}
```

Observar 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 es programa es de un 250%, comparado con más del 400% del ejemplo anterior.

Ventaja 2: Propagar los errores sobre el stack (pila) de llamadas

Una segunda ventaja de las excepciones es la posibilidad del propagar el error encontrado sobre la pila de llamadas a métodos. Suponiendo que el método leerArchivo 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 leerArchivo.

Ejemplo

```
1. metodo1 {
2.        llamar a metodo2;
3. }
4. metodo2 {
5.        llamar a metodo3;
6. }
7. metodo3 {
8.        llamar a leerArchivo;
9. }
```

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

```
1. metodo1 {
2.        numCodigoTipoDeError error;
3.        error = llamar a metodo2;
4.        if (error)
5.              procesodelError;
6.        else
7.              proceder;
8. }
9.
10. numCodigoTipoDeError metodo2 {
11.       numCodigoTipoDeError error;
12.       error = llamar a metodo3;
13.       if (error)
14.             return error;
15.       else
16.             proceder;
17. }
18.
19. numCodigoTipoDeError metodo3 {
20.       numCodigoTipoDeError error;
21.       error = llamar a leerArchivo;
22.       if (error)
23.             return error;
24.       else
25.             proceder;
26. }
```

Como se mencionó 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 (esto es, simplemente, no capturando la excepción), 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.

```
1. metodo1 {
2.       try {
3.             llamar a metodo2;
4.       } catch (excepcion) {
5.             procesodelError;
6.       }
7. }
8.
9. metodo2 throws excepcion {
10.      llamar a metodo3;
11.}
12.
13.metodo3 throws excepcion {
14.      llamar a leerArchivo;
15.}
```

Sin embargo, como se puede ver desde este pseudocódigo, requiere cierto esfuerzo por parte de los métodos que participan del proceso. Cualquier excepción a capturar que pueda ser lanzada dentro de un método forma parte de la firma o prototipo del método y debe ser especificado en la cláusula **throws** del mismo (la razón de esto es que Java debe saber que excepción puede lanzar cada método para poder realizar el "enlace" con el código que la maneje). Esta es la manera por la cual el método informa a quien lo invocó sobre las excepciones que puede lanzar, para que éste pueda decidir qué hacer con dicha excepción.

Observar de nuevo la diferencia del factor de aumento de código como así también la diferencia de claridad entre las dos técnicas de manejo de errores. El código que utiliza excepciones es más compacto y más fácil de entender.

Ventaja 3: Agrupar los tipos de errores y la diferenciación de éstos

Frecuentemente las excepciones se dividen en categorías o grupos. Por ejemplo, se puede suponer un grupo de excepciones, cada una de las cuales representara un tipo de error específico que pudiera ocurrir durante la manipulación de un vector:

- El índice está fuera del rango del tamaño del vector
- El elemento que se quiere insertar en el vector no es del tipo correcto
- El elemento que se está buscando no está en el vector

Además, se puede suponer que algunos métodos querrán manejar todas las excepciones de esa categoría (todas las excepciones de vector), y otros métodos podrían manejar sólo algunas excepciones específicas (como la excepción de índice no válido).

Como todas las excepciones lanzadas dentro de los programas Java son objetos, agrupar o categorizar las excepciones es una solución natural al modelar clases y cadenas de herencia. Las

excepciones Java deben ser subclases de la clase Throwable, o de cualquier subclase de ésta. Al igual que con otras clases de Java, se pueden crear subclases de Throwable y subclases de estas subclases.

Existen clases de excepción que no tienen subclase y otras que si las tienen. La herencia es un tipo de relación entre clases y por consiguiente, los objetos creados a partir de ellas. Esta relación permite agrupar conceptualmente los grupos de excepciones. Cada clase sin subclases representa un tipo específico de excepción y cada clase que tiene una o más subclases representa un grupo de excepciones relacionadas.

Por ejemplo, un grupo de clase que no tienen subclases son:

- InvalidIndexException
- ElementTypeException
- NoSuchElementException

Cada una representa un tipo específico de error que puede ocurrir cuando se manipula un vector. Un método puede capturar una excepción basada en su tipo específico (la clase que la define, una superclase de la cadena o sub cadena de herencia a la que pertenece o un interfaz que alguna de ellas implemente).

Por ejemplo, un manejador de excepción que sólo controle la excepción de índice no válido, tiene una sentencia **catch** (a través de la cual captura la excepción) como la siguiente:

```
catch (InvalidIndexException e) {
…
}
```

De esta manera, si suponemos que VectorException podría ser una superclase de la sub cadena de herencia que representa (agrupa a través de la herencia) cualquier error que pueda ocurrir durante la manipulación de un objeto vector, 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 vector, sin importar sus tipos específicos, un manejador de excepción especificaría un argumento de esta supuesta clase VectorException.

```
catch (VectorException e) {
…
}
```

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

```
catch (Exception e) {
…
```

```
}
```

Los manejadores de excepciones que son demasiado genéricos, como el mostrado anteriormente, 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 genéricos.

Como se ha 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.

Como se mencionó anteriormente, la clase Exception maneja condiciones de errores que no sean críticas que un programa pueda encontrar en el transcurso de su ejecución. En lugar de terminar abruptamente, a través de ella se puede escribir código que maneje las condiciones de error y continuar con el procesamiento normal del programa.

Cualquier condición anormal que influya sobre la ejecución de un programa mientras este se encuentre corriendo es un error o una excepción. Por ejemplo, las excepciones pueden ocurrir porque:

- Se trata de abrir un archivo que no existe
- La conexión de red falla
- Cuando se utilizan operandos que se van del rango posible de trabajo
- No se encuentra una clase que se quiere cargar con la JVM

En Java, la clase Error define que se considera una condición de error seria. Estas son aquellas que no se debería intentar recuperar puesto que ocurren ante problemas involucrados con la plataforma de ejecución y no con el procesamiento del programa. En la mayoría de los casos es conveniente que el programa termine cuando se presentan estas situaciones para ubicar el problema en la plataforma de ejecución.

Las excepciones de Java tiene el estilo de las de C++ para ayudar al programador a construir código más flexible. Cuando ocurre un error en un programa, el método que lo encuentra puede lanzar una excepción en dirección al código del método que lo invocó para señalar el problema que acaba de ocurrir. El método que realizó la invocación puede de esta manera “atrapar o capturar” la excepción lanzada y, en caso de poder hacerlo, recuperarse de ella manejando la condición de error. Esta forma de trabajo permite al programador generar código específico para el manejo de dichas excepciones.

Tipos de Excepciones

Se puede clasificar a las excepciones de diferente manera. Una de ellas es que se deriva de la atención especial que el compilador les brinda al exigir o no su gestión. Existen dos tipos:

- **Verificadas**: Las excepciones verificadas son aquellas que se espera que el programador gestione en el programa y se generan por condiciones externas que pueden afectar a un

programa en ejecución. Dentro de este tipo se encuentran, por ejemplo, cualquier error de E / S como accesos a archivos o comunicaciones por red (se retomará esta tema en el módulo de E / S).

- **No verificadas**: Las excepciones no verificadas pueden proceder de errores del código o situaciones que, en general, se consideran demasiado difíciles para que el programa las maneje de forma razonable. Se denominan así porque no se exige al programador que las verifique ni que haga nada cuando se producen. Estas excepciones son del tipo de las que probablemente ocurren como el resultado de defectos del código y ocurren siempre en tiempo de ejecución. Un ejemplo de este tipo de excepciones es el intento de acceder a un elemento más allá del final de un vector.

Cuando se tratan de excepciones verificadas, es relativamente fácil encontrar el lugar donde no se manejan los posibles errores. Sin embargo, cuando las excepciones no son de este tipo, los errores se pueden detectar cuando un programa termina abruptamente. Por ejemplo, el siguiente código genera una terminación anormal del programa.

Ejemplo

```
package introduccion;

public class HolaMundoExcepciones {
	public static void main(String args[]) {
		int i = 0;

		String saludos[] = { "Hola Mundo de Excepciones!",
				"No, no se rompió!",
				"Acá tampoco!!" };

		while (i < 4) {
			System.out.println(saludos[i]);
			i++;
		}
	}
}
```

Cuando se ejecuta este programa la salida que se obtiene es la siguiente:

```
Hola Mundo de Excepciones!
No, no se rompió!
Acá tampoco!!
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
	at introduccion.HolaMundoExcepciones.main(HolaMundoExcepciones.java:12)
```

Un programa termina con un mensaje de error cuando se lanza una excepción, de manera que el programa anterior muestra la salida anterior luego de ejecutar cuatro veces el ciclo.

El manejo de excepciones le permite al programa capturarlas, manejarlas y luego continuar con la ejecución normal del mismo. Es una estructura de manejo que provee el lenguaje para aquellos casos en los que no se sigue el flujo normal planificado para un programa. El ejemplo anterior muestra en un programa simple donde se puede dar este tipo de situación.

Estos casos especiales deben ser manejados en el momento que ocurren, en bloques de código separados, de manera que dichos bloques se puedan "asociar" a la ejecución normal del programa. Además, que sólo entren en acción cuando la situación anormal ocurre, generando en consecuencia código más simple y fácil de mantener.

Categorías

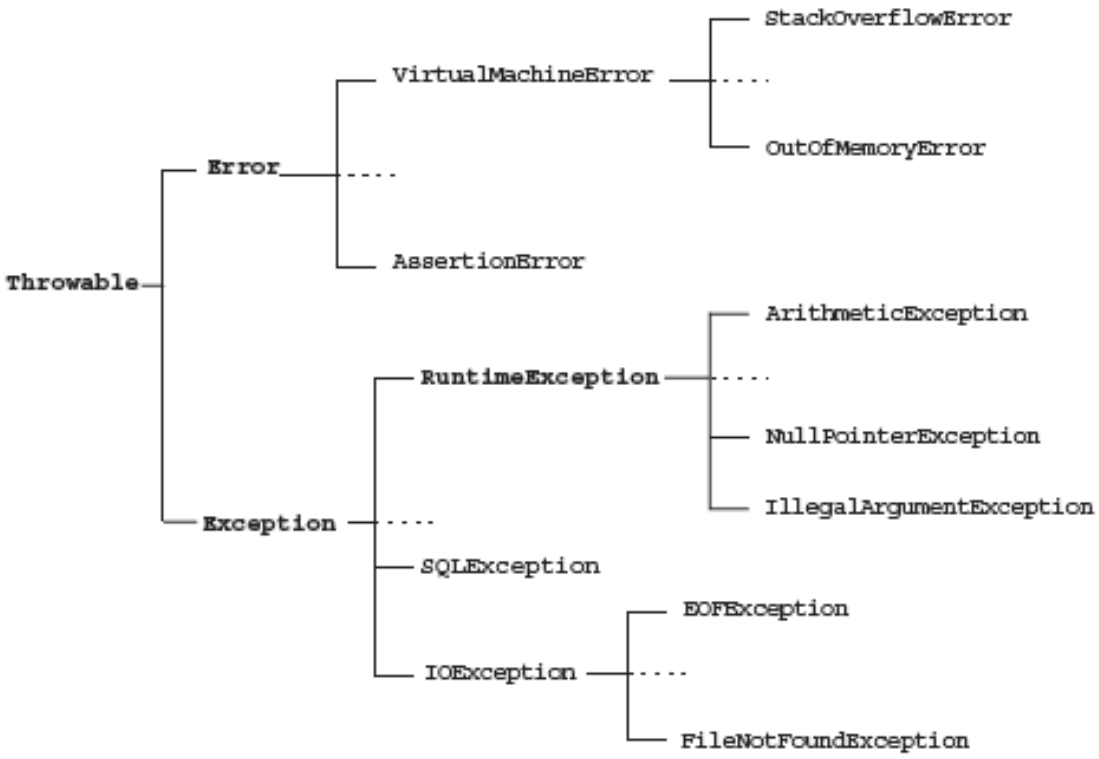
Anteriormente se mencionó que los errores se podían dividir en aquellos que son graves, los de plataforma, y los que se podían manejar en tiempo de ejecución. Para comprender y manejar ambas formas de ocurrir una determinada condición de error, hay que entenderlos en base a las categorías definidas para ellos a través de sus cadenas de herencia, las cuales permiten realizar agrupaciones según su funcionalidad.

Por ejemplo, toda clase que sea capaz de "lanzar" un objeto que sea manejado por la estructura de gestión de excepciones que provee Java debe ser subclase de Throwable, ya que esta actúa como superclase de todos los objetos de este tipo. Las subclases a partir de ella pueden crear instancias de objetos que se podrán "lanzar y capturar" utilizando el mecanismo de manejo de excepciones.

Los métodos definidos en la clase Throwable recuperan el mensaje de error asociado con la excepción e imprimen el recorrido que esta realice en el stack buscando un manejador además del lugar donde dicha excepción ocurrió. Existen dos subclases primordiales y una segunda división dentro de una de ellas, por el tipo de manejo que ofrecen para generar las categorías antes mencionadas a saber:

- Error
- Exception
  - RuntimeException
  - IOException

Las cadenas de herencias que Java provee para la generación de estas subclases se muestran en la siguiente figura



Como se pudo apreciar en el diagrama anterior, el lenguaje provee una cantidad de excepciones predefinidas. Algunas de las más comunes son las siguientes:

- ArithmeticException: es el resultado de dividir por cero una expresión de enteros. Se debe tener en cuenta que esta excepción no se lanza en el caso que la división sea de punto flotante

  Ejemplo

```
int i = 12 / 0
```

- NullPointerException: cualquier intento de acceder a un atributo o método de un objeto cuando no se creo una instancia del mismo, por ejemplo, se creo una variable de referencia del tipo de la clase que define el objeto pero no se lo creo con el operador **new**

  Ejemplo

```
Fecha f;
System.out.println(f.toString());
```

- NegativeArraySizeException: es el resultado de intentar crear un vector con un tamaño negativo en la definición de su dimensión

- ArrayIndexOutOfBoundsException: cuando se intenta acceder a un elemento de un vector más allá del límite que este tiene, se produce esta excepción

- SecurityException: por lo general esta excepción ocurre en un explorador de Internet y es lanzada por la clase SecurityManager cuando un applet realiza una operación no admitida. Si embargo, cualquier situación en la cual la seguridad establecida en esta clase es violada, también genera excepciones de este tipo. Por ejemplo, en el caso de los applets, esta situación se origina cuando:

    - Se trata de acceder a un archivo local en el cliente
    - Se abre un socket a un servidor distinto del cual bajo el applet
    - Se intenta ejecutar otro programa en el entorno de ejecución local

Requerimientos de Java para Capturar o Especificar Excepciones

> *Java requiere que un método o capture o especifique todas las excepciones verificadas 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. Posteriormente se mostrará como tratar con excepciones, a través de introducir un programa de ejemplo que explica cómo capturar excepciones y muestra cómo escribir un manejador de excepciones para el programa de ejemplo.

Especificar

Si se decide que un método no capture una excepción, se 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 de la firma o prototipo del método: aquellos que invocan a un método deben conocer las excepciones que ese método puede lanzar para poder decidir qué hacer son esas excepciones. Así, en la firma del método debe especificar las excepciones que el método puede lanzar.

Excepciones verificadas

Java tiene diferentes tipos de excepciones, incluyendo las excepciones de E/S, las excepciones en tiempo de ejecución, y las de su propia creación.

Las excepciones son en tiempo de ejecución en su mayoría, pero existen otras que se validan en tiempo de compilación. Las excepciones en tiempo de ejecución 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 vector 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 verificar todas las excepciones en tiempo de ejecución excede de los beneficios de capturarlas o especificarlas.

Las excepciones verificadas son aquellas que son verificadas por el compilador (esto es, el compilador comprueba que esas excepciones son capturadas o especificadas).

Algunas veces esto se considera como un ciclo 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.

Excepciones que pueden ser lanzadas desde el ámbito de un método

Java permite que un método lance una excepción desde una sentencia dentro de un método. Para ello debe cumplir con la regla de especificarla en la firma del método mediante la declaración `throws`, y utilizar la palabra reservada `throw` para llevar a cabo el mencionado lanzamiento.

Sin embargo, cuando se especifica la o las excepciones que puede lanzar se debe tener en cuenta en la declaración, que el método puede incluir llamados a otros métodos que a su vez lancen también excepciones. Esto se refleja declarando en la firma del métodos "todas" las excepciones que pueden ser lanzadas por él, las que se incluyen en el flujo de procesamiento y se lanzan mediante la palabra reservada `throw` y las que pueden ser lanzadas por los métodos invocados dentro de su ámbito y se lanzan nuevamente (se omite el manejo por no capturarlas o se capturan y se vuelven a lanzar en el ámbito del método).

Capturar y Manejar Excepciones

El manejo de excepciones se divide en un grupo de tres bloques:

- El bloque `try`
- Los bloques `catch`
- El bloque `finally`

Definición del bloque `try`

El primer paso en la escritura de una manejador de excepciones es poner la sentencia Java en 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` subsiguiente) asociado con él.

Definición de los bloques `catch`

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

Definición del bloque `finally`

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

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 tiene el siguiente formato.

Ejemplo

```
try {
sentencias Java
}
```

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

Para construir un manejador de excepción para un método de una clase, se necesita encerrar la sentencia que lanza la excepción en el método que realiza la declaración o la invocación, dentro de un bloque **`try`**.

Existe más de una forma de realizar esta tarea. Se puede 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 se puede poner todas las sentencias del método que realiza las invocaciones dentro de un sólo bloque **`try`** y asociar varios manejadores con él.

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

Los bloques `catch`

Como se mencionó anteriormente, 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`**.

Ejemplo

```
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 la siguiente:

```
catch (AlgunObjetoThrowable nombreVariable) {
```

```
            Sentencias Java
      }
```

Como se puede observar, la sentencia **catch** requiere *un sólo argumento formal*. El formato de esta sentencia es similar al de una declaración de método. El tipo del argumento AlgunObjetoThrowable declara el tipo de excepción que el manejador puede gestionar 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 lanzado un objeto, y estos sólo pueden ser los derivados de la clase Throwable. Por otro lado, nombreVariable es el nombre a través del cual el manejador puede referirse al objeto del tipo de la excepción capturada.

Manejo de sentencias try – catch

Hasta el momento se mencionó las posibilidades que tiene el mecanismo de manejo de excepciones en Java pero no se integró sus partes bajo las reglas que gobiernan su uso. A partir de este punto se tratará el tema.

Como se mencionó anteriormente, el código que puede lanzar una excepción en particular deberá colocarse dentro de un bloque **try** y a este se le debe adjuntar las lista de los correspondientes bloques **catch** con las posibles excepciones que se pudieran producir. Cada bloque de sentencias en un **catch** maneja la excepción en particular que este puede recibir como argumento. La forma en la cual se asocia un bloque cuando se produce la excepción es por el argumento definido para esta.

```
try {
      // código que puede lanzar una excepción
} catch (MyExceptionType miExcep) {
      // código que se ejecuta si la excepción
      // MyExceptionType es lanzada
} catch (Exception otraExcep) {
      // código que se ejecuta si se lanza cualquier
      // otra excepción
}
```

Los manejadores de excepciones pueden recibir como argumento una referencia a un objeto del tipo de una excepción y llamar al método getMessage() de la excepción utilizando el nombre de la referencia declarado para ella, por ejemplo, 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, en particular, 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 instancia si así lo requirieran,

Para buscar qué métodos implementar en una excepción, se puede comprobar la definición de la clase y las definiciones de las clases que la preceden en la cadena de herencia.

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.

Se debe tener en cuenta, como se mencionó anteriormente, que si una excepción no es manejada por un bloque **try** – **catch**, esta se lanza nuevamente al método que llamó al recibió la excepción, continuando este proceso a lo largo de todos los métodos registrados en el stack.

Si una excepción es lanzada nuevamente hacia atrás a lo largo de todos los métodos que hicieron los respectivos llamados hasta el que generó la excepción y llega a la función main(), el programa termina en forma anormal

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**.

Esta sentencia define un bloque que ***siempre*** se ejecuta, más allá que se haya capturado o no una excepción.

Ejemplo

```
try {
      metodo1();
      metodo2();
} catch (BrokenPipeException e) {
      CreaLogDelProblema(e);
} finally {
      realizarLiberacionRecursos();
}
```

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, si se lanza la excepción y se deriva a un **catch** o si no se lanza y se ejecuta normalmente el código.

El principal beneficio de utilizarla es liberar recursos o la gestión apropiada de los mismos.

En el ejemplo anterior, realizarLiberacionRecursos() se ejecuta más allá que ocurra una excepción o no cuando se ejecute tanto metodo1() como metodo2() (al código dentro del bloque **try** se lo denomina también “código protegido”).

La única posibilidad en la cual una sentencia **finally** y su bloque asociado no se ejecutan es cuando se ejecuta un System.exit() en algún lugar antes de su procesamiento en el código protegido, porque esto termina inmediatamente el programa, lo cual implica que el control del flujo se desvía de su secuencia normal. Si, por ejemplo, dentro del bloque **try** se encontrase una sentencia **return**, el código dentro del bloque **finally** se ejecuta igualmente antes que el método termine y retorne.

Para completar lo enunciado hasta el momento, se puede retomar el ejemplo del recorrido del vector que termina anormalmente e incluir en el código el manejo de la excepción que se lanza. El código resultante sería el siguiente.

Ejemplo

```
package vector;
public class HolaMundoExcepciones {

	public static void main(String[] args) {
		int i = 0;
		int j = 0;

		String greetings[] = { "Hola Mundo de Excepciones!",
				"No, no se rompió!", "Acá tampoco!!" };

		while (i < 4) {
			try {
				System.out.println(greetings[i]);
			} catch (ArrayIndexOutOfBoundsException e) {
				System.out.println("Reiniciando el valor del índice");
				i = -1;
			} finally {
				System.out.println("Esto se imprime siempre");
			}
			i++;
			j++;
			if (j == 20)
				i = 4;
		}
	}
}
```

Notar el uso de una segunda variable para detener la ejecución del programa, ya que al manejar la excepción cuando se produce y cambiar el valor de la variable de control del ciclo, este se convierte en un ciclo infinito.

La salida producida es la siguiente:

```
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
```

```
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Reiniciando el valor del índice
Esto se imprime siempre
```

Como se pudo apreciar en el ejemplo anterior, las excepciones se manejan dentro de bloques **try** – **catch** – **finally**. Lo que el lenguaje requiere para manejar estos bloques es que el código que los utilice debe manejar el lanzamiento de la excepción mientras el método se encuentra en el stack (recordar que el mecanismo de excepciones esta fuertemente asociado al recorrido a efectuar cuando se lanza una excepción a través del stack)

Por otra parte, el código que puede causar una excepción debe tener declarado en el prototipo del método la cláusula **throws** seguida de la o las excepciones que se puedan ocasionar dentro de su bloque de sentencias.

Un método puede tener más de una excepción en la cláusula **throws**, porque se pueden producir en su bloque de sentencias diferentes excepciones según el código que el método maneje. Si son varias excepciones, deberán estar separadas por comas después de la cláusula **throws**.

Ejemplo

```
package multiples;
import java.io.*;
public class MultiBaseA {
	public void metodoA()
		throws IOException, RuntimeException {
			// hace alguna E/S
		}
}
```

Rescritura y Excepciones

Cuando se sobrescriben métodos que lanzan excepciones, el método que realiza la rescritura deberá declarar también la cláusula **throws** con las mismas excepciones que el método que sobrescribe ya que estas forman parte del prototipo del método sobrescrito.

Por lo tanto, los métodos deben lanzar excepciones que pertenecen al mismo tipo de las excepciones que fueron declaradas para el método sobrescrito

Existe una excepción a esta regla, también pueden lanzar excepciones diferentes siempre y cuando dichas excepciones sean subclases de las declaradas en el método sobrescrito, no a la inversa. Esto implica que deben ser subclases de las excepciones declaradas del método sobrescrito, lo cual es fácil de verificar siguiendo las cadenas de herencia que las definen

Si un método de la superclase lanza múltiples excepciones, el método que lo sobrescribe en la subclase deberá lanzar al menos un subconjunto de excepciones apropiadas siguiendo las reglas ya enunciadas

Se pueden resumir las siguientes reglas de codificación:

- Una clase de excepción declarada en un método sobrescrito debe ser del mismo tipo de la que declare el método de la superclase (recordar que las subclases son también del mismo tipo que sus respectivas superclases para cualquier clase, particularmente cierto también para una clase de excepción). Por ejemplo, si el método de la superclase lanza una IOException, el método que la sobrescribe deberá lanzar una IOException o al menos una subclase de esta, como FileNotFoundException
- Se pueden declarar menos excepciones a lanzar que los declarados en el método de la superclase
- No se pueden agregar nuevas excepciones en los métodos que sobrescriben a los de la superclase, salvo que estas sean subclases de una de las declaradas en el método de la superclase que se rescribe.

Ejemplo

```
package rescritura;

public class BaseA {

	public void metodoA() throws RuntimeException {
		// cálculos numéricos
	}
}


package rescritura;
public class DerivadaA1 extends BaseA {

	public void metodoA() throws ArithmeticException {
		// cálculos numéricos
```

```
            }
}
```

```
package rescritura;
public class DerivadaA2 extends BaseA {

      public void metodoA() throws Exception { //Error
            // cálculos numéricos
      }
}
```

En este código se puede apreciar claramente que el error que se comete en la clase DerivadaA2 se debe a que Exception no es subclase de RuntimeException. Esto es una violación a las reglas enunciadas.

Avanzando un poco más en la complejidad de la codificación, se puede analizar los casos en los cuales se sobrescriben métodos que manejan múltiples excepciones.

Ejemplo

```
package multiples;
import java.io.*;
public class MultiBaseA {
      public void metodoA()
            throws IOException, RuntimeException {
                  // hace alguna E/S
            }
}
```

La clase de ejemplo MultiBaseA fue pensada para ser superclase, por lo tanto, los métodos que sobrescriban a metodoA ()deben cumplir con las reglas ya enunciadas

```
package multiples;
import java.io.*;

public class MultiDerivadaA1 extends MultiBaseA{
      public void metodoA()
            throws FileNotFoundException, UTFDataFormatException,
            ArithmeticException {
            // hace alguna E/S y manejo de
            // números
            }
}
```

Como las clases de excepción FileNotFoundException y UTFDataFormatException son subclases de IOException y ArithmeticException es subclase de RuntimeException la declaración del método methodA()es correcta y no se generan errores en tiempo de compilación

```
package multiples;
import java.io.*;
import java.sql.SQLException;
```

```
public class MultiDerivadaA2 extends MultiBaseA{
	public void metodoA()
		throws FileNotFoundException, UTFDataFormatException,
		ArithmeticException, SQLException {
		// hace alguna E/S,
		// manejo de números
		// y SQL
	}
}
```

En la clase MultiDerivadaA2 se genera un error por causa de la declaración de SQLException, ya que esta no es subclase de ninguna de las excepciones definidas en el método de la superclase

```
package multiples;

public class MultiDerivadaA3 extends MultiBaseA{
	public void metodoA() throws java.io.FileNotFoundException {
		// hace alguna E/S de archivo
		}
}
```

La clase MultiDerivadaA3 declara correctamente al método metodoA() porque si bien no se declara en la cláusula **throws** todas las excepciones definidas en el método de la superclase, si se define un subconjunto válido de ellas.

Creación de excepciones personalizadas

Una de las principales ventajas de manejar excepciones para el control de errores en el flujo de un programa es que las excepciones son clases en si mismas y tiene habilitado todo el manejo que se puede realizar con una clase. De esta manera, se puede sacar ventaja de este hecho creando excepciones propias tan sólo con generar una subclase de la clase Exception. Esto permite personalizar el tipo de excepción a manejar haciéndola adecuada al tipo de tratamiento que se quiera otorgar al error producido. Como ejemplo de creación de una clase de excepción, se muestra el siguiente código.

Ejemplo

```
package personalizada;

public class ExcepcionDeTiempoLimiteDelServidor extends Exception {
	private int puerto;
	public ExcepcionDeTiempoLimiteDelServidor(String mensaje, int puerto) {
		super(mensaje);
		this.puerto = puerto;
	}
		// Usar getMessage() para recuperar el string que
		// describe la excepción
	public int getPuerto() {
		return puerto;
	}
}
```

Uno de los detalles a tener en cuenta cuando se crea la excepción es construir adecuadamente la superclase. Por lo general, se le pasa un String con un mensaje que sea descriptivo del error que representa el objeto del tipo excepción, el cual puede recuperarse posteriormente cuando se lance este objeto con el método getMessage().

Si se quiere utilizar la excepción que se creo anteriormente, un ejemplo del código que se encontraría en una clase que defina los métodos que utilicen la excepción es el siguiente:

Ejemplo

```
package personalizada;

public class ClienteDeRed {
	private String servidorPorDefecto;
	private String servidorAlternativo;

	public void conectar(String nombreServidor) throws
			ExcepcionDeTiempoLimiteDelServidor {
		int exitoso;
		int puertoDeConexion = 80;
		exitoso = abrir(nombreServidor, puertoDeConexion);
		if (exitoso == -1) {
			throw new ExcepcionDeTiempoLimiteDelServidor(
				"No se pudo conectar",
				 puertoDeConexion);
			}
	}

	public void buscarServidor() {
		try {
			conectar(servidorPorDefecto);
		} catch (ExcepcionDeTiempoLimiteDelServidor e) {
			System.out.println("Tiempo límite del servidor alcanzado, " +
				"probando el alternativo");
			try {
				conectar(servidorAlternativo);
			} catch (ExcepcionDeTiempoLimiteDelServidor e1) {
				System.out.println("Error: " + e1.getMessage() +
						" conectándose al puerto " +
						e1.getPuerto());
			}
		}
	}

	private int abrir(String servidor, int puerto){
	:
	:
		return 0;
	}
}
```

Java 7

El nuevo `try` con manejo de recursos

Cuando se desarrollan aplicaciones empresariales que necesitan una funcionalidad amplia, el código depende en la mayoría de las situaciones de recursos que existen fuera de la máquina virtual de Java, lo que incluye cualquier cosa, desde un documento sobre el sistema de archivos, un registro en una base de datos, o una socket abierto para comunicarse con un equipo remoto. Trabajar con estas capacidades externas en el código significa la creación de objetos de Java que representan dichos recursos, incluidas las clases, tales como el objeto de conexión JDBC, o la clase de archivo en el paquete java.io. Si bien muchos de estos conceptos escapan as los objetivos actuales, se puede plantear su utilización y ampliarlo luego cuando se vean entradas y salidas. Sin embargo, con cada tecnología externa se debe tomar los recaudos de manejo de excepciones que cada una exija.

La necesidad de una administración automática de recursos en Java (ARM - Automatic Resource Management)

Cuando se interactúa con recursos externos se siguen, por lo general, los mismos pasos:

- Crear un objeto del tipo de la clase que maneje al recurso
- Comunicarse con dicho recurso
- Interactuar con el recurso
- Cerrar la comunicación con el recurso
- Liberar los recursos del sistema

Sin embargo es de público conocimiento que muchas aplicaciones sufren las complicaciones que se derivan del mal cierre o liberación de los recursos, lo cual tiene diferente importancia dependiendo del recurso externo accedido. Inclusive, en algunos casos, se producen degradaciones en la ejecución de las aplicaciones o errores mientras corre el programa que son difíciles de encontrar o recuperar para que este no se bloquee.

La interfaz java.lang.AutoCloseable

Para abordar el problema en la máquina virtual, Java 1.7 ha introducido una nueva interfaz denominada java.lang.AutoCloseable que define un método único, fácil de implementar.

```
void close() throws Exception
```

La idea detrás de su uso es relativamente sencilla. Cuando se coloca un recurso que implementa la interfaz dentro de un bloque try, al finalizar dicho bloque, Java invocará automáticamente al método de la interfaz, independientemente de si se produce una excepción durante la ejecución en curso.

Ejemplo

```
public class CierreAutomatico implements AutoCloseable {

	public void close() throws Exception {
		System.out.println(
			"CierreAutomatico. Ejecución del método close.");
	}
```

```
	public void gestionarRecurso() {
		// Realizar operaciones específicas con el recurso
		System.out.println(
			"CierreAutomatico. Ejecución del método gestionarRecurso.");
		}
}
```

Java 7 permite crear una instancia de la clase CierreAutomatico que implementa la interfaz AutoCloseable dentro de un bloque **try**-con-recursos especial.

Ejemplo

```
public class TryConRecursos {
	public static void main(String[] args) throws Exception{
		try (CierreAutomatico recurso1 = new CierreAutomatico();){
			recurso1.gestionarRecurso();
			}
	}
}
```

Cuando se ejecuta este código, el método close sobrescrito de la interfaz se invoca automáticamente al final del bloque **try**, ***independientemente si ocurre o no una excepción***. La salida es la siguiente:

```
CierreAutomatico. Ejecución del método gestionarRecurso.
CierreAutomatico. Ejecución del método close.
```

La sintaxis simplemente trata de crear o inicializar una instancia de la clase que implementa la interfaz AutoCloseable dentro de los paréntesis que siguen inmediatamente al **try**, lo que se conoce como la declaración de recursos:

```
	try (CierreAutomatico recurso1 = new CierreAutomatico();){
```

Una cosa que podría llamar la atención es el hecho que se esta utilizando la palabra clave **try** sin incluir ni un **catch** o un **finally**, que no sería válido con las versiones anteriores del JDK. Este concepto sigue siendo válido pero se debe agregar una nueva situación, la declaración de recursos. En la versión 7, el error se produce cuando un bloque **try** no tiene un **catch**, un **finally** o una declaración de recursos. En el caso de no cumplir con alguna de estas condiciones, el error obtenido será:

error: 'try' without 'catch', 'finally' or resource declarations

Por otro lado, tener en cuenta que declarar el uso de recursos no inhibe el uso de cualquier bloque que se necesite, ya sean varios **catch** o un **finally**. Sólo hay que tener en cuenta que cuando se ejecutan estos bloques, el método close de AutoCloseable **ya fue invocado**.

Declaración de múltiples recursos

Las declaraciones de múltiples recursos son también totalmente válidas. Por ejemplo, suponiendo una segunda clase que también implemente la interfaz, se pueden declarar ambas en el mismo bloque **try**-con-recursos.

Ejemplo

```
public class SegundoCierreAutomatico implements AutoCloseable {
```

```
	public void close() throws Exception {
		// Cerrar apropiadamente el recurso
		System.out.println(
			"SegundoCierreAutomatico. Ejecución del método close.");
		throw new UnsupportedOperationException(
			"Ha ocurrido un problema en SegundoCierreAutomatico");
		}

	public void manipulateResource() {
		// Realizar operaciones específicas sobre el recurso
		System.out.println(
		"SegundoCierreAutomatico. Ejecución del método gestionarRecurso.");
	}
}
```

Para su utilización, se deben poner ambos recursos como dos instrucciones cualesquiera del lenguaje dentro de los paréntesis del bloque **try**-con-recursos, lo cual implica que deben estar separadas por punto y coma.

Ejemplo

```
public class TryConRecursosMultiples {
	public static void main(String[] args) throws Exception{
		try (CierreAutomatico recurso1 = new CierreAutomatico();
				SegundoCierreAutomatico recurso2 = new
					SegundoCierreAutomatico()) {
			recurso1.gestionarRecurso();
			recurso2.manipulateResource();
			}
	}
}
```

Al ejecutar el programa, se puede apreciar el orden de invocación del método close:

```
CierreAutomatico. Ejecución del método gestionarRecurso.
SegundoCierreAutomatico. Ejecución del método gestionarRecurso.
SegundoCierreAutomatico. Ejecución del método close.
CierreAutomatico. Ejecución del método close.
```

Uso de constructores en los recursos del bloque try-con-recursos

Como se vio anteriormente, el orden de invocación del método close es inverso a la creación de los recursos. Para clarificar el tema se puede hacer uso de constructores y validar la afirmación.

Ejemplo

```
public class CierreAutomatico implements AutoCloseable {

	public CierreAutomatico() {
		System.out.println("Constructor de CierreAutomatico");
	}

	public void close() throws Exception {
		System.out.println(
			"CierreAutomatico. Ejecución del método close.");
	}
```

```
	public void gestionarRecurso() {
		// Realizar operaciones específicas con el recurso
		System.out.println(
			"CierreAutomatico. Ejecución del método gestionarRecurso.");
		}
}


public class SegundoCierreAutomatico implements AutoCloseable {

	public void close() throws Exception {
		// Cerrar apropiadamente el recurso
		System.out.println(
			"SegundoCierreAutomatico. Ejecución del método close.");
		throw new UnsupportedOperationException(
			"Ha ocurrido un problema en SegundoCierreAutomatico");
	}

	public void manipularRecurso() {
		// Realizar operaciones específicas sobre el recurso
		System.out.println(
		"SegundoCierreAutomatico. Ejecución del método gestionarRecurso.");
	}
}


public class TryConRecursosConstructores {
	public static void main(String[] args) throws Exception{
		try (CierreAutomatico recurso1 = new CierreAutomatico();
				SegundoCierreAutomatico recurso2 = new
				SegundoCierreAutomatico()) {
			recurso1.gestionarRecurso();
			recurso2.manipularRecurso();
			}
	}
}
```


Al ejecutar el programa, se obtiene la salida que verifica la afirmación realizada:

```
Constructor de CierreAutomatico
Constructor de SegundoCierreAutomatico
CierreAutomatico. Ejecución del método gestionarRecurso.
SegundoCierreAutomatico. Ejecución del método gestionarRecurso.
SegundoCierreAutomatico. Ejecución del método close.
CierreAutomatico. Ejecución del método close.
```

Cuando dos excepciones son arrojadas por el bloque `try`-con-recursos

Como se mencionó anteriormente, antes de ejecutar cualquiera de los bloques **`catch`** o un bloque **`finally`**, Java invoca automáticamente al método close para cerrar los recursos. Esto deja abierta la siguiente interrogante, ¿qué pasa si se lanza una excepción en un método, pero cuando se invoca al método close también se lanza una excepción? Las clases con un escenario de este tipo se muestran a continuación.

Ejemplo

```
public class CierreAutomatico implements AutoCloseable {

	public void close() throws Exception {
		System.out.println(
			"CierreAutomatico. Ejecución del método close.");
		throw new UnsupportedOperationException(
			"Ha ocurrido un problema en CierreAutomatico");
	}
	public void gestionarRecurso() throws Exception {
		// Realizar operaciones específicas con el recurso
		System.out.println(
			"CierreAutomatico. Ejecución del método gestionarRecurso.");
		throw new Exception(
			"Ha ocurrido un problema en CierreAutomatico");
		}
}



public class SegundoCierreAutomatico implements AutoCloseable {

	public void close() throws Exception {
		// Cerrar apropiadamente el recurso
		System.out.println(
			"SegundoCierreAutomatico. Ejecución del método close.");
		throw new UnsupportedOperationException(
			"Ha ocurrido un problema en SegundoCierreAutomatico");
	}

	public void manipularRecurso() {
		// Realizar operaciones específicas sobre el recurso
		System.out.println(
		"SegundoCierreAutomatico. Ejecución del método gestionarRecurso.");
		throw new NullPointerException(
			"Ha ocurrido un problema en SegundoCierreAutomatico");
	}
}
```

En este punto cabe la pregunta de cuál es la excepción que se ejecuta primero y por qué.

Manejo de excepciones suprimidas en Java 7

La clave de este escenario en particular es la "excepción suprimida". Siempre que se produce una excepción dentro del bloque y es seguida por una excepción en la instrucción **try**-con-recursos, sólo la excepción que se produce en el bloque de la instrucción **try** es elegible para ser capturada por el código de manejo de excepciones. Todas las demás excepciones se consideran ***excepciones suprimidas***, un concepto que es nuevo en Java 7.

Ejemplo

```
public class UsaCierreAutomatico {
	public static void main(String[] args) {
		try (CierreAutomatico recurso1 = new CierreAutomatico();
```

```
                        SegundoCierreAutomatico recurso2 = new
                              SegundoCierreAutomatico()) {
                        recurso1.gestionarRecurso();
                        recurso2.manipularRecurso();
                  } catch (Exception e) {
                        e.printStackTrace();
                        System.out.println("Ver excepciones suprimidas");
                        for (Throwable throwable : e.getSuppressed()) {
                              System.out.println(throwable);
                        }
                  }
            }
}
```

Cuando se ejecuta el programa, se obtiene la siguiente salida:

```
CierreAutomatico. Ejecución del método gestionarRecurso.
SegundoCierreAutomatico. Ejecución del método close.
CierreAutomatico. Ejecución del método close.
java.lang.Exception: Ha ocurrido un problema en
CierreAutomatico.gestionarRecurso
      at suprimidas.CierreAutomatico.gestionarRecurso(CierreAutomatico.java:13)
      at suprimidas.UsaCierreAutomatico.main(UsaCierreAutomatico.java:7)
      Suppressed: java.lang.UnsupportedOperationException: Ha ocurrido un
problema en SegundoCierreAutomatico.close
            at
suprimidas.SegundoCierreAutomatico.close(SegundoCierreAutomatico.java:9)
            at suprimidas.UsaCierreAutomatico.main(UsaCierreAutomatico.java:9)
      Suppressed: java.lang.UnsupportedOperationException: Ha ocurrido un
problema en CierreAutomatico.close
            at suprimidas.CierreAutomatico.close(CierreAutomatico.java:7)
            at suprimidas.UsaCierreAutomatico.main(UsaCierreAutomatico.java:9)
Ver excepciones suprimidas
java.lang.UnsupportedOperationException: Ha ocurrido un problema en
SegundoCierreAutomatico.close
java.lang.UnsupportedOperationException: Ha ocurrido un problema en
CierreAutomatico.close
```

Se puede apreciar que el método e.getSuppressed retorna un vector con las excepciones suprimidas. En realidad, es uno de los nuevos métodos incorporados para el manejo de estas situaciones. Es importante comprender que al suprimir las excepciones no se las ignora, sino que se suprimen para la correcta ejecución del mecanismo de manejo de excepciones. Para hacer frente a este concepto de excepciones suprimidas, dos nuevos métodos y un constructor se han añadido a la clase java.lang.Throwable en Java 7.

```
public void addSuppressed (Throwable exception)
```

Anexa a la excepción especificada de las excepciones que fueron reprimidas con el fin de ofrecer esta excepción.

```
public final Throwable[] getSuppressed()
```

Devuelve un vector que contiene todas las excepciones que fueron suprimidas, por lo general por la instrucción try-con los recursos, a fin de entregar esta excepción.

```
protected Throwable(String message, Throwable cause,
        boolean enableSuppression, boolean writableStackTrace)
```

Tener en cuenta que este constructor se duplica en la clase java.lang.Exception en Java 7, por el hecho de que los constructores no son heredados por las subclases.

Aserciones o aseveraciones

Una aserción es una instrucción que contiene una expresión booleana la cual el programador sabe que en un momento dado de la ejecución del programa se debe evaluar a verdadero (de ahí la aserción o afirmación).

Este es un método habitual para la verificación formal de algoritmos. Básicamente se trata de afirmaciones respecto de precondiciones, post condiciones e invariantes a lo largo del flujo de un programa en puntos determina dos del mismo. Esto implica que verificando que es cierta la expresión booleana propuesta en la afirmación, se comprueba que el programa se ejecuta dentro de los límites que el programador demarca y además reduce la posibilidad de errores.

Las aserciones no son nada nuevo en programación, de hecho estos ya estaban previstos en la especificación inicial de Oak (el lenguaje precursor de Java) pero fueron desestimados por que no había tiempo suficiente para hacer una implementación satisfactoria.

Las aserciones admiten dos tipos de sintaxis:

- assert (Expresión con resultado [boolean]) ;
- assert (Expresión con resultado [boolean]) : (Expresión Descriptiva del error) ;

Se debe definir una expresión que dará como resultado una expresión booleana (verdadero o falso), dicha expresión es precisamente la que será aseverada por Java, si esta expresión resulta verdadera el ciclo de ejecución del programa continuará normalmente. Sin embargo, si resulta falsa la expresión, la ejecución del programa será interrumpida indicando el error de aseveración ("assertion").

La única diferencia que existe entre las declaraciones antes mencionadas, es que la segunda de éstas define una expresión adicional que permite agregar información extra acerca de la aseveración levantada por el programa, dicha declaración puede ser desde una variable hasta un método que de como resultado un valor no nulo (**void**), este resultado es convertido a un String que es pasado al constructor de la aseveración para ser desplegado como información adicional al momento de ejecución; la segunda declaración de aseveración simplemente verifica la validez de la primer expresión sin proporcionar detalles específicos del error.

En ambos casos, si la expresión booleana se evalúa como **false**, se genera un error de aserción (AssertionError). Este error no debería capturarse y el programa debería finalizar de forma anómala. Se puede verificar la clase para manejar este tipo de errores deriva de Error y no de

Exception, lo cual implica que fue diseñada para manejar errores a nivel de la máquina virtual y no del programa.

Si se utiliza el segundo formato, la segunda expresión, que puede ser de cualquier tipo, se convierte en un tipo String y se utiliza para complementar el mensaje que aparece en la pantalla cuando se notifica la aserción.

Nota: Dado que la palabra clave assert es relativamente nueva, el mecanismo de aserción puede interrumpir el código escrito para la versión 1.3 o versiones anteriores del JDK si éste usa la palabra assert como etiqueta o nombre de variable. Puede utilizar la opción -source 1.3 para indicar al compilador javac que compile el código para la versión 1.3.

Las aseveraciones están pensadas para la comprobación de invariantes (condiciones que se cumplen siempre en un determinado lugar del código), por lo que tienen mas interés en la etapa de desarrollo. Por esto se puede desactivar y activar la comprobación de las mismas. Por defecto la comprobación esta desactivada y se proporcionan dos opciones para el intérprete del JDK (java).

- java -enableassertions (-ea), para activar la comprobación.
- java -disableassertions (-da), para desactivar la comprobación.

Si estos modificadores se escriben tal cual, se activará o desactivará la comprobación de aseveraciones para la clase que se pretende ejecutar. Pero si lo que se quiere es activar/desactivar la comprobación en un determinado paquete o de una determinada clase:

- java -enableassertions:simple... Notas
  - (Activa aserciones en el paquete simple, por los puntos ...)
- java -enableassertions:simple Notas
  - (Activa aserciones la clase simple.Notas, por que no lleva puntos)

Y lo mismo para desactivar:

- java -disablessertions: simple... Notas
- java -disableassertions: simple Notas

También se puede activar para unos y desactivar para otros:

- java -ea:simple.Nota... -da: simple.Meses simple.OtraClase

En resumen la sintaxis es la siguiente:

- java [-enableassertions | -ea] [:<package name>"..." | :<class name>]
- java [-disableassertions | -da] [:<package name>"..." | :<class name>]

Ejemplo

```
package simple;
public class Notas {

	private String[] alumnos = { "Pedro", "Juan", "Helena", "Mónica",
			"Sebastián" };
	private int[] notas = { 7, 4, 6, 7, 8 };
```

```
	public String controlDeNotas() {
		for (int i = 0; i < notas.length; i++) {
			System.out.println(notas[i]);
			assert (notas[i] > 5) : determinarNota(i);
		}
		return "Control de Notas correcto";
	}

	public String determinarNota(int indice) {
		int noAprobado = notas[indice];
		String alumno = alumnos[indice];
		String resultado = "El alumno "
				+ alumno + " no aprueba con " + noAprobado +
				" de nota";
		return resultado;
	}

	public static void main(String[] args) {
		Notas examenes = new Notas();
		System.out.println(examenes.controlDeNotas());
	}
}
```

Con las aseveraciones activadas, el resultado obtenido es:

```
7
4
Exception in thread "main" java.lang.AssertionError: El alumno Juan no aprueba
con 4 de nota
	at Notas.controlDeNotas(Notas.java:10)
	at Notas.main(Notas.java:25)
```

Usos recomendados de las aserciones

Las aserciones pueden aportar datos valiosos sobre las suposiciones y expectativas que maneja el programador. Por este motivo, resultan especialmente útiles cuando otros programadores trabajan con el código para operaciones de mantenimiento.

En general, las aserciones deberían utilizarse para verificar la lógica interna de un solo método o un pequeño grupo de métodos fuertemente vinculados entre sí. No deberían utilizarse para comprobar si el código se utiliza correctamente, sino para verificar que se cumplan sus propias expectativas internas.

Invariantes internas

Las invariantes internas se dan cuando se cree que una situación se va a producir en todos los casos o en ningún caso y se escribe el código de acuerdo con esta convicción.

Ejemplo

```
		if (y > 0) {
			// hacer esto
```

```
} else {
        // hacer aquello
}
```

Si parte de la premisa de que x tiene el valor 0, y que nunca puede ser un valor negativo, y resulta que sí adopta un valor negativo, el código hará algo imprevisto. Casi con toda seguridad se comportará de forma incorrecta. Y lo que es peor, dado que el código no se detiene ni comunica el problema, no lo detectará hasta mucho después, cuando el daño causado sea aún mayor. En tales casos, resulta muy difícil determinar el origen del problema.

Para estas situaciones, la adición de una aserción al abrir el bloque **else** ayuda a garantizar la veracidad de la premisa y permite ver rápidamente el error cuando ésta no se cumple o cuando alguien cambia el código y la premisa deja de ser cierta. Una vez introducida la aserción, el código debería ser parecido al siguiente.

Ejemplo

```
if (x > 0) {
        // hacer esto
} else {
        assert (x == 0);
        // hacer aquello a menos que x sea negativo
}
```

Invariantes del control de flujo

Las invariantes del flujo de control reflejan suposiciones similares a las de las invariantes internas, pero tienen que ver con la dirección que sigue el flujo de ejecución, más que con el valor o las relaciones de las variables.

Por ejemplo, es posible que piense que ha enumerado cada valor posible de la variable de control en una sentencia switch y que, por tanto, nunca se ejecutará una sentencia default. Esto debería verificarse agregando una sentencia de aserción similar a ésta:

Ejemplo

```
switch (mes) {
case 1:
        System.out.println("Enero");
        break;
case 2:
        System.out.println("Febrero");
        break;
case 3:
        System.out.println("Marzo");
        break;
case 4:
        System.out.println("Abril");
        break;
case 5:
        System.out.println("Mayo");
```

```
                break;
        case 6:
                System.out.println("Junio");
                break;
        case 7:
                System.out.println("Julio");
                break;
        case 8:
                System.out.println("Agosto");
                break;
        case 9:
                System.out.println("Septiembre");
                break;
        case 10:
                System.out.println("Octubre");
                break;
        case 11:
                System.out.println("Noviembre");
                break;
        case 12:
                System.out.println("Diciembre");
                break;
        default:
                assert (false) : "Ese no es un mes válido!";
        }
```

Cuando se ejecuta este código, el programa termina anormalmente mostrando por consola un error como el siguiente:

```
Exception in thread "main" java.lang.AssertionError: Ese no es un mes válido!
        at simple.Meses.imprimeMes(Meses.java:19)
        at simple.Meses.main(Meses.java:25)
```

Post condiciones e invariantes de clase

Las post condiciones son suposiciones sobre el valor o las relaciones que presentan las variables al finalizar un método. Un ejemplo sencillo de prueba de post condición sería verificar que después de ejecutar un método para quitar un elemento de una pila (sacar), ésta contiene uno menos de los que tenía antes de llamar al método, a menos que la pila ya estuviese vacía.

Ejemplo

```
package postcondiciones;

public class Pila {
        private int vec[] = new int[10];
        private int indice = 0;
        private int elementos = 0;

        public void agregar(int valor) throws ExcepcionPila {
                if (indice == 9)
                        throw new ExcepcionPila("Pila llena");
                vec[indice] = valor;
```

```
            elementos = ++indice;
        }

        public int sacar() throws ExcepcionPila {
            int resultado = 0;
            if (indice == 0)
                throw new ExcepcionPila("Pila vacía");
            elementos = --indice;
            resultado = vec[indice];
            assert (elementos == indice);
            return resultado;
        }

        public static void main(String[] args) {
            Pila p = new Pila();
            try {
                p.agregar(8);
                p.agregar(4);
                p.agregar(3);
                p.agregar(4);
                p.sacar();
                p.sacar();
                p.sacar();
                p.sacar();
            } catch (ExcepcionPila e) {
                e.printStackTrace();
            }
            System.out.println();
        }

}
```


Observe que, si el método “sacar” no generase una excepción en caso de recibir una llamada para actuar sobre una pila vacía, sería más difícil expresar esta aserción, porque el tamaño original “cero” seguiría siendo cero en el resultado final. Observe también que, en la prueba de precondición, es decir, aquella que determina si se ha llamado al método para actuar sobre una pila vacía, no se utiliza ninguna aserción. Esto es porque, si se produce tal situación, no se debe a un error de la lógica local del método, sino a la forma en que se está utilizando el objeto. En general, estas pruebas sobre el comportamiento externo no deberían utilizar aserciones, sino simples excepciones. Esto garantiza que la prueba se realizará siempre y que no se inhabilitará, como puede ocurrir con las aserciones.

Una invariante de clase es aquella que puede comprobarse al final de cada llamada a un método de una clase. En el caso de una clase Pila, una condición invariante es aquella en que el número de elementos nunca es negativo.

Usos inapropiados de las aserciones

No utilizar las aserciones para comprobar los parámetros de un método **public**. Es el método el que debería comprobar cada parámetro y generar la excepción apropiada, por ejemplo, IllegalArgumentException o NullPointerException. La razón por la que no debería usarse

aserciones para comprobar parámetros es porque es posible desactivarlas a pesar que el programador quiera seguir realizando la comprobación.

En las pruebas de las aserciones, no utilizar métodos que puedan provocar efectos secundarios, ya que dichas pruebas pueden desactivarse en el momento de la ejecución. Si la aplicación depende de estos efectos secundarios, se comportará de forma diferente cuando las pruebas de aserción se desactiven.