Exceptions y Assertions

# Temas:

1. Usar las sentencias throw y throws
2. Usar la sentencia try con las clausulas multi-catch y finally
3. Auto-cerrar recursos con try-with-resources
4. Crear excepciones personalizadas
5. Provar invariantes usando assertions

En este capítulo aprenderás acerca del soporte que tiene Java para manejar excepciones en detalle. Primero aprenderás los conceptos básicos del manejo de excepciones y después aprenderás como lanzar (throw), atrapar (catch) y relanzar excepciones. También aprenderás acerca de las características recién agregadas como el try-with-resources y multi-catch. En seguida aprenderás cómo definir tus propias clases para excepciones personalizadas. Finalmente, discutiremos los temas relacionados con assertions y te enseñaremos cómo usarlas en tus programas. Muchos de los ejemplos en este capítulo usarán funciones I/O para ilustrar los conceptos del manejo de excepciones.

# Introducción al manejo de Excepciones

Como programadores somos optimistas, solo escribimos el código para resolver el problema a mano y esperamos que todo funcione sin problemas. Sin embargo, algunas cosas pueden salir mal, entonces debemos anticiparnos siempre a los errores y excepciones y escribir código para manejar condiciones excepcionales.

Java tiene un manejo de excepciones integrado. El lenguaje Java soporta el manejo de excepciones en la forma de las palabras clave throw, throws, try, catch, y finally. Observa el siguiente código para entender la sintaxis básica de estas palabras.

[ExceptionSintax.java](src/java/oca2/cap11/ExceptionSintax.java)

## Lanzar Excepciones

El código siguiente es un ejemplo muy simple en donde quieres duplicar el texto escrito en la línea de comandos. Asume que el usuario debe escribir algún texto como argumento en la línea de comandos para duplicarlo, o de lo contrario necesitas informar al usuario acerca de la condición de error.

[Echo.java](src/java/oca2/cap11/Echo.java)

En este caso imprimes el error en la consola usando una sentencia println(). Este es un programa sin importancia y el error ocurre en el método main(), entonces el manejo del error es fácil. En este caso, puedes terminar el programa después de imprimir el error en la consola. Sin embargo, si estás muy profundo en las funciones en la llamada de una aplicación compleja, necesitas una mejor manera de indicar que una condición excepcional ha ocurrido, para después informar al que llama acerca de esta condición. Además, puedes también necesitar recuperar la aplicación de la condición de error, en lugar de terminarla. Entonces necesitas manejar la excepción o relanzarla hacia arriba en la pila de llamadas hasta que algún llamador la maneje. (Vamos a revisar el tema de relanzar excepciones después) Por el momento, cambiarás el programa para lanzar una excepción en lugar de imprimir el mensaje de error, en un programa separado:

[Echo1.java](src/java/oca2/cap11/Echo1.java)

Ya que no hay código que atrape y maneje esta excepción, esta excepción termina con el programa. En este caso tú explícitamente lanzaste una excepción. Las excepciones también pueden lanzarse cuando escribes código o llamas a la API de Java. Vamos a ver un ejemplo ahora.

## Excepciones no manejadas

Considera el programa que sigue, que intenta leer un valor entero que el usuario escribe en la consola y después lo vuelve a imprimir a la consola. Para leer un entero de la consola usarás el método readInt() de la clase java.util.Scanner. Para instanciar la clase Scanner, pasas como argumento a System.in en el constructor, que está referenciado a *system input stream.*

[ScanInt.java](src/java/oca2/cap11/ScanInt.java)

Si corres este programa y escribes en la consola un entero, digamos 10, el programa trabaja correctamente e imprime el entero de regreso a la consola. Sin embargo si escribes un string “diez” en lugar del valor entero, el programa se terminará después de lanzar una excepción como esta: java.util.InputMismatchException.

Si lees la documentación del método nextInt(), verás que este método puede lanzar esta excepción, si el token siguiente no coincide con una expresión regular de valor entero, o está fuera de rango. En este programa simple, asumes que el usuario siempre escribirá un entero como se espera y si esto falla, una excepción será lanzada. Si aquí el programa lanza una excepción y se deja sin manejar, el programa terminará de manera anormal después de lanzar el rastro del stack, como se mostró aquí.

Un *stack trace* muestra la lista de métodos (con números de línea) que fueron llamados antes de que el control alcanzara la sentencia donde fue lanzada la excepción. Como programador, encontrarás útil seguir el rastro del flujo de control para poder corregir el programa que apunta a esta excepción.

Entonces ¿Cómo puedes manejar esta situación? Necesitas poner este código entre los bloques *try-catch* y después manejar la excepción.

## Sentencias try-catch

Java provee las palabras clave *try* y *catch* para manejar excepciones que pueden lanzarse en el código. El siguiente ejemplo es el programa anterior mejorado para atrapar la excepción.

[ScanInt2.java](src/java/oca2/cap11/ScanInt2.java)

Ahora analicemos este código. El bloque seguido por la palabra clave try limita el segmento de código a donde tú esperas que alguna excepción pueda ser lanzada. Si alguna excepción es lanzada por el bloque try, java en tiempo de ejecución busca el manejador que coincida con la excepción lanzada (que discutiremos más adelante). En este caso, un manejador de excepciones para InputMismatchException está presente, que es el exactamente el mismo tipo que la excepción que se va a lanzar. Esta excepción que coincide exactamente, está disponible fuera del bloque try in la forma de un bloque al que precede la palabra catch, y este bloque catch será ejecutado. En el bloque catch atrapas la excepción, entonces tú estás manejando la excepción aquí: provees una descripción del error clara para los humanos en lugar de mostrar la rara pila de seguimiento como en el ejemplo anterior, por lo tanto estás proveyendo una buena salida para el programa.

## Acceso programático a la pila de seguimiento *stack trace*

Observaste que la pila de seguimiento es útil para encontrar y corregir el programa, por lo cual si estás manejando la excepción en un bloque catch, puedes usar el método printStackTrace() que imprimirá la pila de seguimiento en el bloque catch. Vamos a agregar la siguiente sentencia al bloque catch del programa anterior:

ime.printStackTrace();

Puedes incluso acceder a cada una de las entradas del *stack trace*. Todas las excepciones tienen un método llamado getStackTrace() que retorna un array de StackTraceElements. Entonces considera escribir este método en el bloque catch:

[Código de ejemplo](src/java/oca2/cap11/ScanInt2.java)

## Múltiples bloques *catch*

En el siguiente código usas un objeto Scanner para leer un entero de la consola. Nota que puedes usar un objeto Scanner para leer Strings también.

[ScanInt3.java](src/java/oca2/cap11/ScanInt3.java)

Si pasas como argumento un string que no coincide con una expresión regular de un entero, entonces obtendrás un InputMismatchException, y si pasas como argumento una String vacía “” entonces obtendrás un NoSuchElementException.

Si observas la documentación para el método Scanner.nextInt() encontrarás que puede lanzar también una IllegalStateException (esta excepción es lanzada si el método nextInt() es llamado en un objeto Scanner que ya está cerrado). Entonces, en estas situaciones, podemos proveer manejadores de excepciones para:

* InputMismatchException
* NoSuchElementException
* IllegalStateException

Veamos el siguiente código que muestra cómo hacerlo.

[ScanInt4.java](src/java/oca2/cap11/ScanInt4.java)

Como puedes ver en la salida, ya que la String dada está vacía, entonces lanzará una excepción como sigue; NoSuchElementException. Es atrapada en el manejador catch para esta excepción, y el código provisto dentro de este bloque será ejecutado para una salida buena del programa.

Nota cómo provees más de un manejador catch apilado para arriba: provees excepciones específicas (tipos derivados) para después seguir hacia abajo con excepciones más generales. Si provees una excepción derivada después de su excepción clase base obtendrás un error de compilación. Todavía no conoces pero NoSuchElementExcepton es la clase base de InputMismatchException. Veamos qué pasa si intentas invertir el orden de los manejadores catch, entre la primera excepción y la segunda. Obtendrás un error de compilación.

NOTA: cuando provees múltiples manejadores catch, maneja excepciones específicas antes de manejar excepciones generales. Si provees una clase derivada de una excepción después de su clase base, tu código no compilará.

## Bloques Multi-Catch

Has visto que no puedes invertir el orden de los manejadores catch para una clase derivada y su clase base. Sin embargo es posible combinar estos dos manejadores juntos. Java 7 provee una característica llamada multi-catch en donde puedes combinar múltiples manejadores catch. Observa el siguiente código.

[ScanInt5.java](src/java/oca2/cap11/ScanInt5.java)

Nota como combinas los manejadores catch juntos usando el operador OR “|” aquí (Es el mismo operador que usas para operaciones OR a nivel de bits).

A diferencia de la combinación de las excepciones combinadas NoSuchElementException y IllegalStateException, no puedes combinar las cláusulas NoSuchElementException y InputMismatchException. Como ya lo hemos discutido, una clase es la clase base de la otra y por esta razón no puedes atrapar ambas en el mismo bloque multi-catch. Si lo intentas obtendrás un error de compilación: “Alternatives in a multi-catch statement cannot be related by subclassing”.

Entonces la alternativa a este problema es proveer un catch para la clase base y esto es suficiente, ya que la clase base también manejará las excepciones de sus clases derivadas.

NOTA: Solo puedes combinar manejadores catch para excepciones que no comparten una relación parent-child entre ellas.

¿Cómo sabes si es mejor combinar el manejo de excepciones o apilarlas? Esta es una elección de diseño que debes considerar siguiendo estos aspectos:

1. ¿Las excepciones serán lanzadas por razones similares o diferentes?
2. ¿Es el manejo del código similar o diferente?

Si tu respuesta es “similar” para ambas preguntas es mejor combinar en un multi-catch, si tu respuesta es “diferente” para cualquiera de las dos preguntas, es mejor separar y apilar.

¿Qué pasa con el código anterior? ¿Es mejor combinar o separar los manejadores para las dos excepciones en el multi-catch? Puedes ver que las excepciones se manejan en el mismo bloque catch, pero las razones de estas dos excepciones son considerablemente diferentes. La excepción InputMismatchException se lanza por un error de programación cuando llamas al método nextInt() después de llamar al método close() en la clase Scanner. Por lo que en este caso es mejor separar los manejadores de estas dos excepciones.

## Manejadores Catch Generales

Como te habrás dado cuenta, muchas excepciones pueden ser lanzadas cuando usas la API I/O. Hemos discutido solo en la llamada de un solo método nextInt() de la clase Scanner, necesitas manejar tres excepciones:

1. InputMismatchException
2. NoSuchElementException
3. IllegalStateException.

Si mantienes el manejo de excepciones específicas como aquí puede no resultar en una condición excepcional cuando corres el programa, mucho de tu código consiste en bloques try-catch! Es la mejor manera de decir ¿“maneja otras excepciones”? Sí, puedes proveer un manejador de excepciones más general.

El código en el programa [ScanInt3.java](src/java/oca2/cap11/ScanInt3.java), si lo mejoramos con un manejador de excepciones general. Ahora provee un manejador para la clase base Exception (de donde se derivan todas las excepciones). Entonces, si el bloque try lanza cualquier otra excepción, y si la excepción es de una clase derivada de la clase Exception, este catch general la manejará. Es una práctica recomendada atrapar excepciones específicas, y proveer un manejador de excepciones generales para asegurar que otras excepciones serán manejadas también.

## Excepciones encadenadas

Cuando quieres atrapar una excepción y lanzar otra, puedes encadenar la primera excepción a la excepción lanzada. En otras palabras, cuando lanzas una excepción, puedes asociar otra excepción que la causa.

Cuando creas un objeto Exception puedes usar un constructor que toma otra excepción como argumento; este argumento pasado es la excepción encadenada a el objeto excepción que fue creado. Aquí también hay un constructor sobrecargado que toma un mensaje descriptivo como argumento adicional. Por ejemplo, la siguiente ejemplo hay dos constructores sobrecargados de la clase Exception

* Exception(Trowable cause)
* Exception(String detailMsg, Throwable cause)

Constructores similares están disponibles para otras clases como Throwable, Error, y RuntimeException. El siguiente código ilustra excepciones encadenadas.

[ChainedException.java](src/java/oca2/cap11/ChainedException.java)

Los métodos relacionados con las excepciones encadenadas son getCause() e initCause() en la clase Throwable.

El método getCause() retorna un objeto Throwable. Este retorna una excepción encadenada al objeto excepción donde este método es invocado. Esta excepción encadenada es la excepción original para esta excepción. Si ninguna excepción está encadenada a esta, retornará null.

El método initCause(Throwable causeException) fija la excepción encadenada para el objeto excepción en el método que es llamado. Si la excepción encadenada ya está fijada actualmente cuando creas el objeto, llamar a este método resultará en una IllegalStateException. Este método puede ser llamado solo una vez; cualquier intento de llamarlo más de una vez lanzará una excepción IllegalStateException.

Nota que las excepciones pueden ser encadenadas a cualquier nivel de profundidad.

## Bloques Finally

Hay un método close() provisto en la clase Scanner, y necesitas cerrarla. En las clases [ScanInt1](src/java/oca2/cap11/ScanInt.java). [ScanInt2](src/java/oca2/cap11/ScanInt2.java) y [ScanInt3](src/java/oca2/cap11/ScanInt3.java) nota que abres un objeto Scanner pero no lo cierras. Entonces este programa tiene un desperdicio de recursos!! la palabra “recurso” se refiere a cualquiera de las clases que adquiere algo de recursos del sistema, como red, archivos, base de datos, y otros. Pero ¿Cómo sabes se estas clases necesitan ser cerradas? Buena pregunta. La respuesta es si la clase implementa la interface java.io.Closeable, entonces puedes llamar al método close() de esta clase; de otra manera resultará en un desperdicio de recursos.

NOTA: El garbage collector (gc) es responsable de devolver solo recursos de memoria. Si estás usando alguna clase que requiere de recursos del sistema, es tu responsabilidad regresarlos llamando al método close() en el objeto.

El siguiente código llama al método close() de la clase Scanner en su método main(); Haremos un código corto por lo que manejaremos solo una excepción general.

[ScanInt6.java](src/java/oca2/cap11/ScanInt6.java)

Como puedes ver en la salida, si ingresas un número no entero, el programa no cierra el Scanner; ¿Cómo puedes arreglar esto? Una forma es llamar al método close() en el bloque catch. Esta solución funciona pero no es elegante. Sabes que puedes tener múltiples bloques catch y por lo tanto tendrías que proveer la llamada al método close() en cada uno de ellos. Por lo que el mejor camino para cerrar los recursos es usar un bloque finally. Observa el siguiente código.

[ScanInt7.java](src/java/oca2/cap11/ScanInt7.java)

En este caso, un bloque finally es provisto después del bloque catch. Este bloque finally será ejecutado si ocurre una excepción o no. Por lo tanto, el bloque finally es un buen lugar para cerrar los recursos llamando al método close() para asegurar que este recurso será devuelto al sistema siempre.

NOTA: si llamas al método System.exit() dentro de un método, este terminará de manera anormal el programa. Entonces, si el método llamado tiene un bloque finally, no será llamado y los recursos serán desperdiciados. Por esta razón, es una mala práctica llamar System.exit() para terminar el programa.

NOTA: puedes tener un bloque finally directamente después de un bloque try, sin bloque catch; esta característica es raramente usada pero es muy útil a veces.

## Puntos a recordar

* Puedes atrapar excepciones y envolverlas dentro de una excepción más genérica, y lanzarlas en un nivel más alto en la pila de llamadas. Cuando atrapas una excepción y creas excepciones más generales, puedes retener la referencia a la excepción original. Esto se llama [Exception chaining](src/java/oca2/cap11/ChainingEx.java).
* Encadenar excepciones es útil por propósitos de corrección de código. Cuando obtienes una excepción general puedes checar si está encadenada una excepción de más bajo nivel e intentar entender por qué esta excepción ocurrió.
* El bloque finally siempre se ejecuta, independientemente que el try haya lanzado una excepción o no. Considera el siguiente [código](src/java/oca2/cap11/TryFinally.java). ¿Qué valor retornará el código, true o false? Este método siempre retornará falso porque el bloque finally siempre es ejecutado. De hecho, si usas la opción –Xlint obtendrás una advertencia “finally clause cannot complete normally”

## Relanzamiento preciso de excepciones

Considera el siguiente programa:

[PreciseRetrow.java](src/java/oca2/cap11/PreciseRethrow.java)

Si intentas compilar este programa en versiones anteriores a Java 1.7 obtendrás el siguiente error: unreported exception Exception; must be caught or declared to be thrown.

En este programa, el método Integer.parseInt() puede lanzar un NumberFormatException. Sin embargo, el bloque catch está atrapando una excepción más general de tipo Exception. Dentro del bloque catch, la excepción es relanzada. Ahora, la cláusula del método foo() indica que se puede lanzar la excepción NumberFormatException, lo que es correcto porque es la única excepción que el método parseInt() puede lanzar. Sin embargo, ya que el tipo estático de la excepción relanzada es de tipo Exception, el compilador se quejará ya que la cláusula del método foo() deberá declararse también como Exception.

Java 7 te permite ser más específico cuando relanzas una excepción. Si relanzas una excepción desde un bloque catch, puedes lanzar un tipo que el bloque try puede lanzar pero ningún bloque catch la atrapa. También, la excepción relanzada necesita no ser la misma que la del parámetro del cach; esta debe ser un subtipo. Por lo tanto la clase PreciseRethrow compilará sin advertencias ni errores. Aunque por supuesto el programa fallará después de lanzar NumberFormatException ya que “ten” no es un entero.

# Try-with-resources

Es un error bastante común de los programadores Java que olviden liberar recursos, incluso en el bloque final. También, si estás tratando con múltiples recursos, es tedioso recordar llamar al método close() en el bloque finally. Java presentó una nueva característica llamada try-with-resources para ayudarte a hacer tu vida fácil. El siguiente código hace uso de esta característica y mejora la versión del código anterior.

[TryWithResources.java](src/java/oca2/cap11/TryWithResources.java)

El programa tiene un comportamiento similar al programa parecido. En la sentencia try-with-resources obtienes los recursos dentro de los parámetros, después de la palabra clave try, pero antes del bloque try. También no provees el bloque finally. El compilador internamente traduce este try-with-resources en un bloque try-finally (por supuesto, el compilador retendrá todos los bloques catch que hayas declarado). Puedes adquirir muchos recursos en esta característica; cada recurso debe estar separado por punto y coma “;”.

Puedes proveer sentencias try-with-resources sin ningún catch o finally explícito. Recuerda que un bloque try puede asociarse con un bloque catch o un bloque finally o ambos. Una sentencia try-with-resources se expande internamente en un bloque try-finally, por lo que puedes declarar un try-with-resources sin un bloque catch o finally. Observa el siguiente código que hace uso de esta característica.

[TryWithResources2.java](src/java/oca2/cap11/TryWithResources2.java)

Aunque esto es posible (try sin catch ni finally) no significa que puedas hacerlo. Por ejemplo, ya que este código no tiene bloque catch, si escribes alguna entrada inválida, el programa se estrellará.

Entonces el beneficio del try-with-resources simplifica la programación porque no tienes que proveer un finally. Sin embargo, necesitas proveer los bloques catch necesarios para manejar las excepciones que puedan ocurrir.

Nota que para un recurso usable en el try-with-resources, la clase del recurso debe implementar la interface java.lang.AutoCloseable. Esta interface declara un único método llamado close(). Ya sabes que el try-with-resources fue añadido en Java 7. La interface AutoCloseable también fue presentada en Java 7, y la interface está hecha como interface base de la interface Closeable. Esto es para asegurar que las clases existentes anteriores funcionen con try-with-resources. En otras palabras, puedes usar viejas clases stream con esta característica porque todas implementan la interface AutoCloseable.

## Cerrando múltiples recursos

Puedes usar más de un recurso en un try-with-resources. Aquí hay un trozo de código que crea un archivo zip de un texto dado, que hace uso de un try-with-resources.

[ZipSnnipet.txt](src/java/oca2/cap11/ZipSnippet.txt)

En este código, el buffer es un array de bytes. Este array es un almacén temporal útil para copiar datos raw de un stream a otro stream. En la sentencia try-with-resources. Abres dos streams: ZipOutputStream para escribir en el archivo zip y FileInputStream para leer el archivo de texto. (Nota que la API soporta zip y jar en la librería java.util.zip). Quieres leer el archivo de texto de entrada, comprimirla y colocar esta entrada en un archivo zip. Para colocar un archivo/directorio en un archivo zip, la clase ZipOutputStream provee un método llamado putNextEntry(), que toma un objeto ZipEntry como argumento. La sentencia zipFile.putNextEntry(new ZipEntry(fileName)); coloca un archivo de entrada llamado fileName en zipFile.

Para leer los contenidos del archivo de texto, usas el método read() en la clase FileInputStream. El método read() toma un buffer como argumento. La cantidad de datos a leer por iteración es dado por el tamaño del array del buffer; aquí son 1024 bytes en este ejemplo. El método read() retorna el número de bytes que leyó, y si no hay más datos que leer retorna -1. El while checa si la lectura es exitosa usando la condición > 0, antes de escribir en el archivo zip.

Para escribir los datos al archivo zip, usas el método write() en la clase ZipOutputStream. Este método toma tres argumentos: el primer argumento es el buffer de datos (byte[] buffer); el segundo es el comienzo compensado en el buffer (que es 0 porque siempre leerás desde el principio del buffer); y el tercero es el número de bytes a escribirse (lenRead) (que se calculan en la sentencia dentro del condicional del while: lenRead = fileIn.read(buffer)).

Ahora vamos a las discusiones principales. Nota cómo abres los dos recursos en el bloque try y cómo están separados por punto y coma “;”. No tienes que escribir un bloque finally para liberar los recursos porque el compilador automáticamente insertará las llamadas respectivas al método close() para estos dos streams en un bloque finally que el propio compilador crea.

El siguiente código es el programa completo para usar el código ilustrado anteriormente, donde se usa la característica try-with-resources para liberar automáticamente los recursos.

[ZipTextFile.java](src/java/oca2/cap11/ZipTextFile.java)

Ya hemos discutido del try-with-resources. Lo que no hemos discutido son las suppressed exceptions. En un bloque try-with-resources puede haber más de una excepción lanzada; por ejemplo, una en el bloque try, una en el bloque catch y otra en el bloque finally. Sin embargo, solo puedes atrapar una, por lo que las otras excepciones serán listadas como suprimidas. Para obtener esta lista de excepciones suprimidas puedes usar el método getSuppressed().

## Puntos a recordar

* No puedes asignar a las variables de recursos declarados en el try-with-resources dentro del bloque del try. Esto es para asegurar que estos recursos sean cerrados en el bloque finally.
* Es un error común cerrar un recurso explícitamente dentro del try-with-resources. Recuerda que esta característica se expande en tiempo de compilación para llamar al método close() en el bloque finally, por lo tanto el código expandido tiene una doble llamada al método close. Considera el siguiente [código](src/java/oca2/cap11/DoubleClose.txt).
* La documentación del método close() in la clase Scanner dice que si el objeto Scanner ya está cerrado, entonces llamar al método close() otra vez no tendrá efecto. Entonces, está segura esta operación. Sin embargo, en general, no puedes esperar que todos los recursos que tienen implementado el método sea seguro llamarlo dos veces. Por lo tanto esta es una mala práctica.

# Tipos de Excepciones

Hasta ahora, nos hemos enfocado en explicar los constructos con relación a excepciones del lenguaje. Algunos conceptos no están claros para ti todavía. Por ejemplo, aprendiste la cláusula *throws* que declaras en un método que puede lanzar alguna excepción. Viste algunos tipos de excepciones que no necesitas declarar en la cláusula *throws*, pero para otros tipos de excepciones sí. ¿Por qué? ¿Cuál es la diferencia entre estos tipos de excepciones? Para responder estas preguntas y darte una mejor idea acerca del manejo de excepciones soportada en las librerías de Java, discutiremos los tipos de excepciones en esta sección.

En java no puedes lanzar excepciones de tipo primitivo. Esto no significa que puedas lanzar excepciones de cualquier objeto como excepción. El objeto lanzado debe ser una instancia de la clase *Throwable* o alguna de sus subclases. Los constructos para manejo de excepciones, como la sentencia *throw* o la cláusula *throws*; *catch* solo tratan con tipo *Throwable* y sus subclases. Hay tres importantes subclases que necesitas aprender en detalle: *Error, Exception* y *RuntimeException* y la figura siguiente te muestra una vista de estas clases.



**Throwable**: Clase base de la jerarquía de excepciones; todas las clases de excepciones extienden a esta clase.

**Exception**: Clase base de las excepciones checadas como **ParseException** o **IOException**, estas excepciones estas obligado a manejarlas.

**Error**: Los errores de la JVM como **StackOverflowError**, no debes de manejarlos.

**RuntimeException**: Clase base para representar errores de programación como **IndexOutOfBoundsException**.

## La clase Exception

Excepciones de este tipo son conocidas como excepciones checadas. Si tu código puede lanzar una excepción, debes manejarla usando un bloque catch o declarar que el método puede lanzar una excepción, forzando al que lo llama a manejar la excepción. Observa el siguiente código.

[CheckedExceptionExample1.java](src/java/oca2/cap11/CheckedExceptionExample1.java)

El constructor de FileInputStream declara que puede lanzar una excepción FileNotFoundException. Esta excepción es derivada de la clase Exception, por lo tanto es una excepción checada. Con excepciones checadas, Java te obliga a pensar en las condiciones de falla y cómo manejarlas. Ahora, piensa en el siguiente código donde estás intentando abrir el archivo dado por args[0], que puede no existir. Si no quieres atrapar y manejar la excepción, debes declarar que el método main() puede lanzar esta excepción con la cláusula throwss.

[CheckedExceptionExaple2.java](src/java/oca2/cap11/CheckedExceptionExample2.java)

Este código compilará. Sin embargo, este código sigue siendo no-satisfactorio: si pasas el nombre de un archivo que no existe, en la línea de comandos, el programa fallará después de lanzar la excepción.

Una mejor forma de manejar esta situación es lanzar un mensaje de error para el usuario e informarle que debe pasar el nombre de archivo correcto.

[CheckedExceptionExample3.java](src/java/oca2/cap11/CheckedExceptionExample3.java)

NOTA: Si tienes algún código que pueda lanzar una excepción checada de un método, puedes escoger de entre dos alternativas. Puedes manejar la excepción con un bloque catch o declarar que el método lanza la excepción con la cláusula throws. Si no escoges ninguna alternativa, no compilará tu programa.

La siguiente tabla resume importantes subclases de la clase Exception

|  |  |
| --- | --- |
| Clase | Descripción corta |
| CloneNotSupportedException | Se lanza cuando el método clone() es invocado en un objeto que su clase no implementa la interface Cloneable |
| IOException | Se lanza cuando una operación I/O falla |
| EOFException | Se lanza cuando el final de un archivo es alcanzado de manera inesperada; es subclase de IOException |
| FileNotFoundException | Se lanza cuando el programa no puede localizar el archivo dado; es subclase de IOException |
| ReflectiveOperationException | Se lanza cuando una operación de reflexión falla; esta clase es la superclase de excepciones relacionadas con operaciones de reflexión: NoSuchMethodException y InvocationTargetException |
| RuntimeException | Superclase de todas las excepciones no-checadas; se discutirá en la siguiente sección |
| SQLException | Se lanza cuando el acceso a una base de datos o alguna operación relacionada falla; es superclase de las excepciones relacionadas con operaciones en bases de datos. |
| ParseException | Se lanza cuando el análisis falla. Por ejemplo, mientras se procesa información local o fechas y tiempos en la clase Format |

## La clase RuntimeException

Esta clase es derivada de la clase Exception. Las excepciones derivadas de esta clase son conocidas como excepciones no-checadas. Vamos a analizar un ejemplo.

[UnCheckedExceptionExample1.java](src/java/oca2/cap11/UnCheckedExceptionExample1.java)

Si ejecutas este programa sin pasar argumento, este fallará después de lanzar una excepción de tipo ArrayIndexOutOfBoundsException.

En este programa, sin checar el tamaño del array (args.length), intentas leer el contenido del mismo con la expresión args[0]. En otras palabras, asumes que el usuario siempre proveerá el nombre del archivo a abrir, y cuando no es provisto este argumento, viene una condición de excepción. Por lo tanto es un error de programación. Nota que a través de la expresión args[0] puede lanzar esta exceptión, y no tienes que atraparla ni declararla en el método. Esto es porque ArrayIndexOutOfBoundsException es una excepción derivada de RuntimeException.

NOTA: Es opcional manejar excepciones no checadas. Si un segmento de código en un método puede lanzar una excepción no-checada, no es obligatorio lanzarla ni declararla en el método.

Cómo puedes arreglar el problema en el código anterior. Observa el siguiente código.

[UnCheckedExceptionExample2.java](src/java/oca2/cap11/UnCheckedExceptionExample2.java)

Cuando corres este programa sin argumentos, el bloque catch atrapa la excepción e imprime el mensaje que le será útil al usuario para saber en qué falló. Sin embargo, esta aproximación de atrapar excepciones RuntimeException es una mala práctica, ya que estas excepciones indican que hay errores de programación y debes de corregirlos en lugar de atrapar la excepción y manejarla. Para arreglar el problema primero debes de checar el tamaño de args antes de intentar acceder a un miembro del arreglo. Observa el siguiente código.

[UnCheckedExceptionExample3.java](src/java/oca2/cap11/UnCheckedExceptionExample3.java)

La salida es la misma si no se pasa ningún argumento al programa, pero el código es mejor: este checa si es posible indexar el array antes, y eso es un programa defensivo.

NOTA: Es mejor ejecutar código defensivo en lugar de alcanzar RuntimeException innecesarias.

La siguiente tabla resume importantes subclases de RuntimeException

|  |  |
| --- | --- |
| Clase | Descripción corta |
| ArithmeticException | Se lanza cuando ocurren errores aritméticos, como intentar dividir entre cero |
| BufferOverflowException  BufferUnderflowException | Se lanza cuando intentas escribir más allá de los límites de un buffer |
| ClassCastException | Se lanza por un intento de hacer un cast entre tipos incompatibles |
| NegativeArraySizeException | Se lanza cuando hay un intento de crear un array con un índice negativo |
| NoSuchElementException | Se lanza por un intento de invocar el método nextElement() de un tipo Enumeration, cuando ya no hay más elementos. |
| NullPointerException | Cuando hay un intento de referenciar una referencia null |
| UnsupportedOperationException | Se lanza cuando hay un intento de hacer una operación que no está soportada o no existe (Ejemplo, escribir en un archivo de solo lectura resultará en ReadOnlyFileSystemException, que es derivada de esta excepción) |
| IllegalArgumentException | Se lanza cuando se pasa un argumento incorrecto o inapropiado a un método |
| IndexOutOfBoundsException | Se lanza cuando se hace un intento de acceder a una estructura de datos usando un índice fuera del rango; Es la clase base de ArrayIndexOutOfBoundsException |

## La clase Error

Cuando la JVM detecta una condición anormal seria in el programa, alcanza una excepción de tipo Error. Cuando obtienes este tipo de excepciones o cualquiera de sus subtipos, la excepción no está hecha para que la manejes. El mejor curso de la acción es dejar al programa fallar. Vamos a discutir porqué, con un ejemplo trivial para entender esto.

Asume que intentas ejecutar un programa que no existe. Por ejemplo, considera el UnCheckedException3 que viste anteriormente; si tienes un error en el nombre de la clase e intentas invocarla, obtendrás un NoClassDefFoundError. El stack trace indica que has introducido un nombre erróneo y que no existe la clase. Por lo tanto la JVM responde con esta excepción.

Considera el siguiente programa para entender cómo un error puede ocurrir. Asume que estás escribiendo un método recursivo para calgular el factorial de un número y olvidas colocar la condición de término.

[NoTerminatingRecursion.java](src/java/oca2/cap11/NoTerminatingRecursion.java)

Cuando corres el programa fallará lanzando una excepción java.lang.StackOverflowError e imprimirá la llamada al método un gran número de veces. Por cada llamada al método, la JVM crea su pila de llamadas y después de algún tiempo donde se agota el área del stack. En este punto, la JVM no puede continuar la ejecución, por lo que lanza StackOverflowError. Cuando obtienes este error puedes estar seguro de que es un error de programación lo que causa este error. Necesitas arreglarlo en el programa para evitar alcanzar este tipo de excepciones.

NOTA: Excepciones de tipo Error indican una condición anormal en el programa. No tiene sentido atrapar este tipo de excepciones e intentar continuar la ejecución, pretendiendo que nada ha pasado. Esta es una muy mala práctica.

La siguiente tabla lista importantes subclases de la clase Error.

|  |  |
| --- | --- |
| Clase | Descripción corta |
| AssertionError | Se lanza cuando un assertion falla (se verá en breve) |
| IOError | Se lanza cuando un error serio de I/O ocurre |
| VirtualMachineError | Se lanza cuando la JVM entra en un estado de error (algún bug) o cuando la JVM se queda sin recursos (como memoria) |
| OutOfMemoryError | Se lanza cuando la JVM no puede colocar memoria; es derivada de la clase anterior |
| LinkageError | Se lanza cuando la ejecución de enlazado por la JVM falla (por ejemplo, en una jerarquía circular de clases en donde se lanzará un error de tipo ClassCircularityError que es derivado de este LinkageError) |
| NoClassDefFoundError | Se lanza cuando se intenta cargar la definición de una clase que el ClassLoader no puede encontrar |
| StackOverflowError | Se lanza cuando la aplicación tiene una llamada recursiva que no tiene fin, o cuando la aplicación hace muchas llamadas y la JVM no puede manejarlas; es una clase derivada de VirtualMachineError |

## La cláusula throws

Un método puede lanzar excepciones checadas; esta cláusula en la signatura especifica que el método puede lanzar esta excepción. Ya has tenido un breve vistaso a esta palabra clave throws al principio de este capítulo. En la cláusula throws, puedes listar las excepciones checadas que el método puede lanzar, por lo que entender estas es un prerrequisito para entender la cláusula. Ya que hemos cubierto estas excepciones, cubriremos la cláusula throw ahora.

Vamos a intentar leer un entero almacenado en un archivo llamado integer.txt en el directorio actual. Aquí hay un constructor sobrecargado para la clase Scanner que toma un archivo como entrada, vamos a utilizarlo en el siguiente código.

[ThrowsClause1.java](src/java/oca2/cap11/ThrowsClause1.java)

El compilador fallará: unreported exception FileNotFoundException; must be caught or declared to be thrown. Si observas la declaración de este método scaner:

public Scanner(File source) throws FileNotFoundException {...

Entonces cualquier método que invoque este constructor debe, por lo tanto, manejar esta excepción o añadir la cláusula throws para declarar que el método lanza esta excepción. En el siguiente ejemplo añadiremos al código anterior la cláusula throws con la excepción.

[ThrowsClause2.java](src/java/oca2/cap11/ThrowsClause2.java)

En el siguiente ejemplo, vamos a extraer el código dentro del método main() a un nuevo método llamado readIntFromFile(). Tienes que definir como método de instancia, por lo que tienes que crear un objeto ThrowsClause3 para invocar este método desde el método main(). Ya que el código dentro de readIntFromFile() puede lanzar FileNotFoundException, tienes dos alternativas: manejar la excepción con un bloque catch o declararla en el método con la cláusula throws.

[ThrowsClause3.java](src/java/oca2/cap11/ThrowsClause3.java)

## Sobre-escritura de métodos y la cláusula throws

Cuando un método sobre-escribible tiene la cláusula throws, hay muchas cosas que considerar para sobre-escribirlo. Considera el siguiente programa que implementa una interface llamada IntReade. Esta interface declara un único método llamado readIntFromFile() y en la cláusula throws lista una FileNotFoundException.

[ThrowsClause4.java](src/java/oca2/cap11/ThrowsClause4.java)

En este código, puedes observar unas importantes cosas. Primero, puedes declarar la cláusula throws en métodos de interfaces; de hecho, puedes proveer la cláusula throws para métodos abstractos declarados en clases abstractas. Segundo, el método declarado en la interface IntReader declara que lanza una IOException, que es más general que FileNotFoundException. Cuando implementas un método, es aceptable proveer la misma excepción o una excepción más específica del mismo tipo. En este caso, el método sobre-escrito readIntFromFile() lista una excepción más específica en su implementación que la de su método base.

¿Qué pasa si intentas cambiar la cláusula throws? Hay muchas formas de cambiar la cláusula throws en el método sobre-escrito

1. No proveer cláusula throws
2. Proveer excepciones checadas más generales
3. Proveer más excepciones que las declaradas en el método base

Si intentas cualquiera de estos tres casos obtendrás un error de compilación. Por ejemplo, intenta no proveer la cláusula throws in el método readIntFromFile() y obtendrás un error de compilación.

Para resumir, la cláusula throws en el método de la clase base es un contrato que provee al llamador de este método: que tiene que manejar las excepciones listadas o declararlas en la cláusula throws. Cuando sobre-escribes un método base, el método derivado debe adherirse a este contrato. El llamador del método base está preparado para manejar solo las excepciones listadas en el método base, por lo que sobre-escribir el método no puede lanzar excepciones más generales que las listadas.

Sin embargo, nota que esta discusión de la cláusula en el método derivado debe seguir el contrato del método base solo para excepciones checadas. Excepciones no checadas pueden ser adheridas o removidas del contrato. Por ejemplo, al método readIntFromFile se puede agregar la excepción no checada NoSuchElementException.

Este es una cláusula aceptable ya que NoSuchElementException puede ser lanzada por el método Scanner.nextInt() cuando pasas un argumento no válido para este método.

## Puntos a recordar

* Si un método no tiene una cláusula throws, no significa que no lance ninguna excepción; esto significa que no lanza ninguna excepción checada.
* Es una buena práctica usar la anotación @throws en el JavaDoc para documentar las situaciones específicas o casos que pueden lanzar una excepción del método; tanto checadas o no-checadas.
* Es mala práctica usar la cláusula throws para listar excepciones no checadas que un método puede lanzar. ¿Por qué? Ya que el compilador no fuerza a los que llaman al método a manejar excepciones no-checadas, no tiene sentido listarlas en la cláusula throws. Por el contrario, si un método puede lanzar una excepción no checada, es buena práctica mencionarlo en la anotación @throws en el javadoc.
* Los bloques de inicialización estáticos no pueden lanzar ninguna excepción checada. ¿Por qué? Recuerda que estos bloques son invocados cuando la clase se carga, entonces no hay manera de manejar excepciones. Por lo tanto no hay forma de declarar excepciones checadas en una cláusula throws.
* Inicialización no-estática puede lanzar excepciones checadas; sin embargo, todos los constructores deben declarar estas excepciones en su cláusula throws. ¿Por qué? Tel compilador junta el código de la inicialización no-estática y los constructores durante la fase de generación de código, por lo tanto, la cláusula throws del constructor puede ser usada para declarar excepciones checadas que estos bloques puedan lanzar.
* Un método sobre-escrito no puede declarar más excepciones en la cláusula throws que las listadas en la cláusula throws del método base. ¿Por qué? Los que llaman al método base ven solo la lista de excepciones dadas in la cláusula throws de este método y deberán ser declaradas o checadas estas excepciones en el código.
* Un método sobre-escrito puede declarar excepciones más específicas que las de su método base; en otras palabras puedes declarar excepciones derivadas en la cláusula throws del método sobre-escrito.
* Si un método es declarado en dos o más interfaces, y si este método declara que lanza diferentes excepciones en su cláusula throws, la implementación debe listar el conjunto de todas las excepciones.

# Excepciones personalizadas

En muchas situaciones, será suficiente lanzar excepciones que ya están provistas por la API de Java. Por ejemplo, si estás validando los argumentos pasados a una función pública, y encuentras que pueden ser nulos o fuera del rango específico, puedes lanzar una IllegalArgumentException. Sin embargo, para aplicaciones menos triviales, será necesario que desarrolles tus propias clases de excepciones (excepciones personalizadas) para indicar condiciones excepcionales.

¿Cómo puedes definir una excepción personalizada? Hay dos opciones: puedes extender tanto a Exception o RuntimeException, dependiendo de tus necesidades. Si quieres forzar a los usuarios a manejar tu excepción personalizada, entonces deberás extender a Exception, lo que hace a tu excepción una excepción checada. Si quieres dar flexibilidad a los usuarios para tu excepción y dejar que ellos decidan si quieren manejarla o no, entonces deberás extender a RuntimeException. ¿Qué pasa cuando extiendes a Throwable o a Error para excepciones personalizadas? La clase Throwable es muy genérica para hacerla base de tu excepción, por lo que no es muy recomendable. La clase Error es reservada para excepciones fatales que la JVM puede lanzar, por lo que no es recomendable hacer de esta, la clase base de tu excepción.

Para extender de una clase base, necesitas ver qué métodos provee. En este caso, quieres crear una excepción personalizada a partir de la clase Exception o RuntimeException. Ya que la clase Exception es la clase base de RuntimeException, es suficiente conocer los miembros de la clase Exception. La siguiente tabla muestra los miembros más importantes.

|  |  |
| --- | --- |
| Miembro | Descripción corta |
| Exception() | Constructor por default de la clase Exception |
| Exception(String) | Constructor que toma información detallada como String como argumento |
| Exception(String, Throwable) | En adición al anterior, toma la causa de la excepción (que puede ser otra excepción) |
| Exception(Throwable) | Toma la causa de la excepción como un solo argumento |
| String getMessage() | Retorna el mensaje detallado |
| Throwable getCause() | Retorna la causa de la excepción, o null |
| Throwable getSuppressed() | Retorna la lista de excepciones suprimidas (típicamente en try-with-resources) |
| void printStackTrace() | Imprime el stack trace (las llamadas de los métodos con las líneas de número lelevantes) a la consola (la salida de error estandar). Si la causa de una excepción (que es otra excepción) está disponible en la excepción, entonces la información también será impresa. |

Para ilustrar cómo crear tu propia excepción, asume que quieres crear una excepción custom llamada InvalidInputException. Cuando intentas leer una entrada (Leer un entero en este caso), y falla, quieres lanzar esta excepción. El siguiente código define esta excepción extendiendo a la clase RuntimeException.

[InvalidInputException.java](src/java/oca2/cap11/InvalidInputException.java)

En esta excepción no introduces ningún nuevo campo, pero puedes agregar campos si es necesario. Esta es una simple excepción personalizada que en su construcción simplemente llama a los constructores de la clase base. El programa que sigue muestra cómo usar esta excepción personalizada.

[CustomExceptionTest.java](src/java/oca2/cap11/CustomExceptionTest.java)

En este código usas InvalidInputException como cualquier otra excepción definida en java. Atrapas esta excepción lanzada por el método readIntFromConsole(). Al imprimir la excepción invocas al método toString(), que está implementado por default en la clase RunntimeException. Este método toString() imprime el nombre de la excepción que fue lanzada (InvalidInputException) e incluye la información detallada del String (Invalid integer input typed in console) que pasas cuando creaste la excepción con el operador new. La última sentencia en el método main() es obtener la causa de la excepción. Ya que la causa de la excepción es InputMismatchException se imprime el nombre cualificado de la misma, java.util.InputMismatchException. Puedes pensar que InputMismatchException causa InvalidInputException; estas dos excepciones son excepciones encadenadas.

# Assertions

Cuando creas aplicaciones asumes muchas cosas. Sin embargo, las cosas asumidas no prevén condiciones erróneas. La sentencia assert es usada para checar o probar tus asunciones acerca del programa.

La palabra clave assert provee soporte para assertions en java. Cada assertion contiene una expresión Boolean. Si el resultado de la expresión es true, significa que la asunción es verdadera, nada pasa. Sin embargo, si el resultado es falso, entonces la asunción que tienes sobre el programa falla y es lanzado un AssertionError. Recuerda que la clase Error y sus derivadas indican un error serio en tiempo de ejecución y no deben de ser manejados. Del mismo modo, si este error es lanzado, la mejor acción es no atrapar la excepción y dejar que el programa termine. Después tendrás que revisar tu asunción y corregir tu código para que de true.

Hay muchas razones para usar assertions en los programas. Una razón es que te ayuda a encontrar problemas temprano; cuando checas las asunciones en el programa, y cuando alguna falla, sabes inmediatamente dónde buscar el problema y arreglarlo. También, cuando otros programadores lean tu código con assertions, entenderán mejor el código porque tus asunciones son explícitas con los assertions.

## Sentencia assert

Hay dos formas de esta sentencia:

* assert booleanExpression;
* assert booleanExpression : “String con mensaje detallado”;

Es un error de compilación si esta sentencia no contiene una expresión Boolean. El siguiente código es el primer ejemplo de uso de asserion.

[AssertionExample.java](src/java/oca2/cap11/AssertionExample.java)

En este programa estás checando primero si el valor es negativo, entonces lo haces positivo. Ya que la condición if(i < 0) el valor i no puede ser NUNCA negativo, o esta es tu asunción. La asunción puede ser probada con un con una sentencia assert.

El programa trabaja bien si la expresión booleana (i >= 0) evalúa a true. Sin embargo, si evalúa a false, el programa fallará lanzando un AssertionError. Vamos a checar este comportamiento, para lo cual deberás ejecutar el programa con la bandera –ea para habilitar assertions en tiempo de ejecución; después discutiremos esta bandera.

El programa se ejecuta exitosamente sin lanzar excepciones.

¿Hay algún valor para el cual la condición puede fallar? Sí. Si el valor es el mínimo posible de un entero, no puede ser convertido en un valor positivo. ¿Por qué? Recuerda que el valor de los int se encuentra en el intervalo desde -2147483648 a 2147483647. En otras palabras, el valor positivo 2147483648 no está en el rango de los enteros. Entonces si el valor -2147483648 pasa a la expresión –i sobrepasará el rango y se volverá otra vez negativo, entonces la asunción ya no es verdadera. Haz un cambio en el programa y verifica los resultados: int i = Integer.MIN\_VALUE;.

En la salida verás que la assertion falla. La aplicación termina porque el programa lanzó un error AssertionError, no hay quién lo maneje y el programa finaliza.

Has visto que las assertions están deshabilitadas en tiempo de ejecución; para habilitarlas tienes que usar un switch –ea (o su versión larga -enableasserts). Para deshabilitar assertions en el tiempo de ejecución, usa el switch –da. Si las assertions están deshabilitadas por default, ¿Qué pasa si uso el switch –da? Hay muchos usos. Por ejemplo, si quieres habilitar assertions para todas las clases de un paquete y deshabilitarlas para un paquete específico, entonces –da es muy útil. La siguiente tabla lista los argumentos de línea de comandos más importantes y su significado. Nota que necesitas no recompilar tus programas para habilitar o deshabilitar assetions; solo usar los argumentos cuando invocas a la JVM.

|  |  |
| --- | --- |
| Argumento | Descripción corta |
| -ea | Habilita las assertions por default con excepción de las clases de sistema. |
| -ea:<class name> | Habilita las assertions para la clase dada |
| -ea:<package name> | Habilita las assertions para el paquete dado |
| -ea:... | Habilita las assertions en el paquete dado sin nombrarlo |
| -esa | Habilita las assertions en las clases de sistema. Esta opción es raro usarla |
| -da | Deshabilita las assertions por default con excepción de las clases de sistema |
| -da:<class name> | Deshabilita las assertions para la clase dada |
| -da:<package name> | Deshabilita las assertions para el paquete dado |
| -da:... | Deshabilita las assertions para el paquete sin nombre dado |
| -dsa | Deshabilita las assertions para las clases de sistema; es raro usarlo |

## Cómo no usar asserts

La clave para entender assertions es que estas son útiles para análisis, pruebas y corrección de aplicaciones, y las assertions se deshabilitan cuando la aplicación es desplegada para el usuario final.

* No uses Assetions para validar valores de entrada o para validar argumentos para métodos públicos. Para señalar fallas en runtime usa mejor excepciones.
* No uses Assertions para checar condiciones que son requeridas para el correcto funcionamiento de la aplicación. Ya que las assertions están deshabilitadas por default, la aplicación puede no funcionar correctamente cuando las asserts no están presentes en el código.
* La expresión booleana dada dentro de la sentencia assert no debe tener efectos: modificar variables, imprimir valores a la consola, etc. En otras palabras, la funcionalidad de la aplicación debe de ser la misma, sin importar que las assertions estén habilitadas o no.

# Resumen

Introducción al manejo de excepciones

* Cuando una excepción es lanzada de un bloque try, la JVM busca por el catch de la lista de catchs en el método call-chain. Si no se encuentra un catch, esta excepción terminará con la aplicación.
* Cuando provees múltiples manejadores de excepciones (catch apilados), deberán ser provistos primero los que manejan excepciones específicas y después excepciones más generales. Proveer excepciones base antes de sus derivados resultan en error de compilación.
* Puedes programáticamente acceder al stack trace usando métodos como printStackTrace() o getStackTrace(), que pueden ser llamados desde cualquier tipo de excepción.
* un bloque try puede tener múltiples catch. Si la causa de dos o más excepciones es similar, o el código que las maneja, puedes considerar combinar los manejadores en un multi-catch.
* El código dentro de un bloque finally será ejecutado independientemente de que el bloque try lance o no una excepción. Esto hace al bloque try el más correcto para liberar recursos, como manejadores de archivos, bases de datos, sockets, etc.

try-with-resources

* Olvidar liberar recursos con el método close() es un error común. Puedes usar try-with-resources para simplificar tu código y auto-cerrar los recursos. Para poder usar un recurso dentro de un try-with-resources, la clase de este recurso debe implementar a la interface java.lang.AutoCloseable y definir el método close().
* Puedes auto-cerrar múltiples recursos con una sentencia try-with-resources. Estos recursos deben separarse por punto y coma “;” en el header del try-with-resources.
* Porque puedes usar múltiples recursos en un try-with-resources, muchas excepciones pueden ser lanzadas del bloque try y el bloque finally es alto. Si un bloque try lanza una excepción, y un bloque finally también lanza excepciones, entonces las excepciones lanzadas en el bloque finally serán adheridas a las excepciones suprimidas de la excepción lanzada del try.

Tipos de excepciones

* La clase Throwable es la clase raíz de la jerarquía de excepciones. Solo Throwable y sus clases derivadas pueden ser usadas para el manejo de excepciones en java.
* La clase Exception (con excepción de RuntimeException y su subjerarquía) y sus clases derivadas, son conocidas como excepciones checadas. Estas excepciones representan condiciones excepcionales que pueden esperarse razonablemente que ocurran cuando el programa se ejecuta, por lo tanto es obligatorio manejarlas. Un método que contiene algún código que pueda lanzar una excepción checada debe también proveer un manejador catch o declarar la excepción en la cláusula throws del método.
* Las clases RuntimeException y Error y sus clases derivadas son conocidas como excepciones no-checadas. Estas pueden lanzarse desde cualquier parte del programa, sin que el código las declare o las maneje.
* La clase RuntimeException y sus clases derivadas representan errores de programación (errores lógicos) y no son generalmente esperados para manejarlos. Sin embargo, en algunos casos, es útil manejar esas excepciones en un bloque catch.
* La clase Error y sus clases derivadas representas excepciones de la JVM; cuando la JVM detecta una condición anormal seria o corre fuera de sus recursos. Cuando un Error ocurre, el mejor curso del programa es que termine.
* Un bloque catch debe manejar la excepción o relanzarla. Para esconder o tragar una excepción después de atraparla y hacer nada es una muy mala práctica.

Cláusula throws

* La cláusula throws para un método está diseñada para listar las excepciones checadas que el cuerpo de ese método puede lanzar.
* Los bloques de inicialización estáticos no pueden lanzar ninguna excepción checada. Bloques de inicialización no-estáticos pueden lanzar excepciones checadas; sin embargo, todos los constructores deben declararlas en su cláusula throws.
* La cláusula throws de un método es parte del contrato para sus métodos sobre-escritos en las clases derivadas. Un método sobre-escrito puede proveer la misma cláusula throws que su método base o excepciones más específicas (subclases de excepciones de la original). El método sobre-escrito no puede proveer una cláusula más general o declarar excepciones checadas adicionales, que las que tiene su método base.

Excepciones personalizadas

* Puedes definir tus propias clases de excepciones (conocidas como excepciones personalizadas) en tus programas.
* Es recomendado que derives excepciones personalizadas de las clases Exception o RuntimeException. Crear excepciones personalizadas extendiendo a la clase Throwable (muy genérica) o Error no es recomendado.
* Puedes envolver una excepción y lanzarla como otra excepción. Estas dos excepciones serán encadenadas. Para la excepción lanzada, puedes obtener la causa original de esa excepción con el método getCause().

Assertions

* Assertions son condiciones checadas en el programa y existen para ser usadas para checar explícitamente asunciones que se hagan en el flujo de un programa.
* La sentencia assert puede ser de dos maneras: una toma un argumento Boolean y la otra, además toma un argumento String.
* Si las condiciones booleanas en la sentencia assert fallan (si evalúan a falso), el programa terminará después de lanzar un AssertionError. No es recomendable atrapar y manejar este error.
* Por default, assertions están deshabilitados en tiempo de ejecución. Puedes usar argumentos de línea de comandos –ea para habilitar, -da para deshabilitar assertions (y sus variantes).