Sumario

Excepciones

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

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: (Extienden directamente de **Exception**) 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: (Extienden de **RuntimeException**)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.

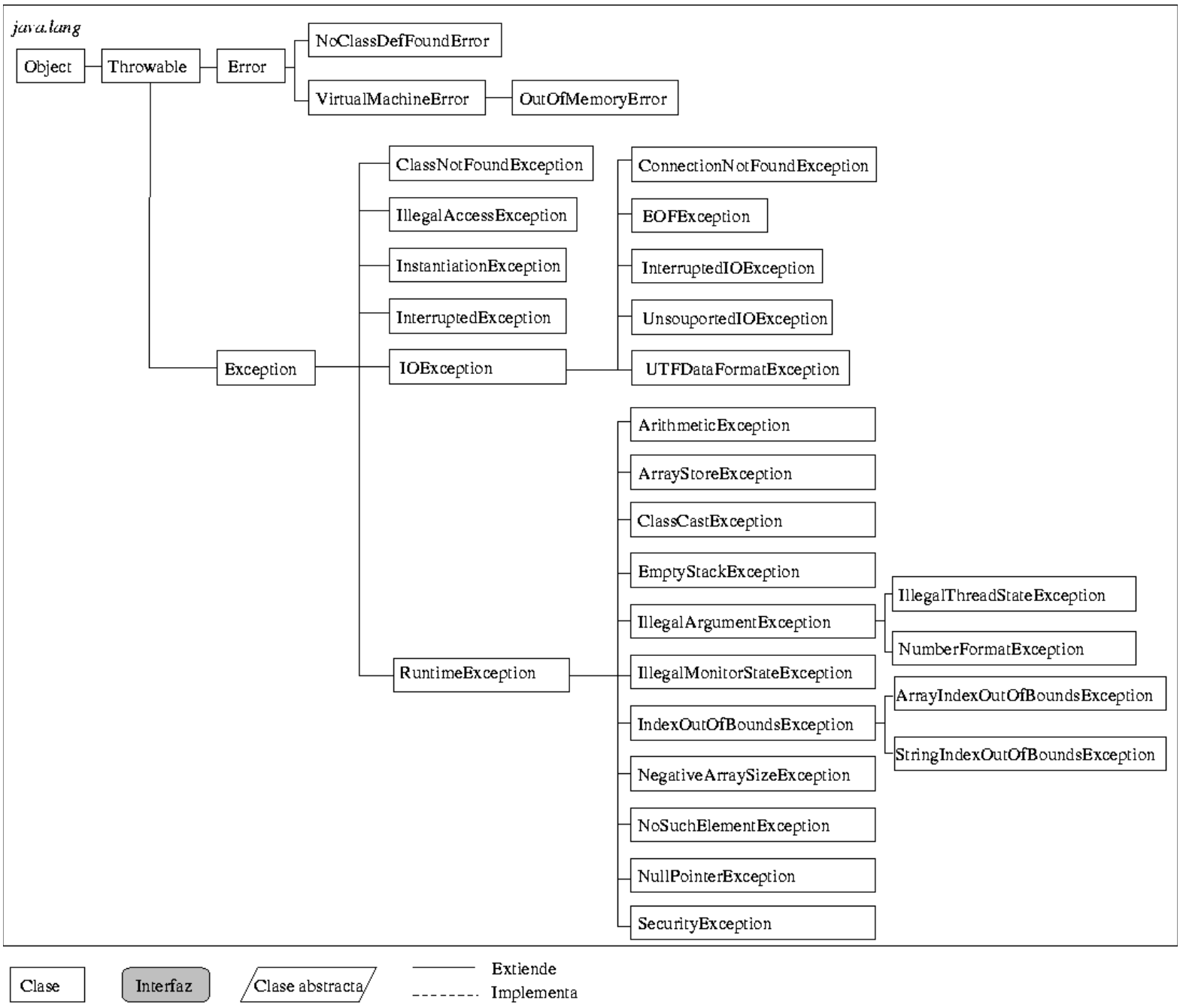
```
System.out.println(10/0);     //arroja una Exception no verificada que
                              //termina con el programa.

int[] vector=new int[10];
vector[20]=2;                 //arroja una Exception no verificada que
                              //tambien termina con la ejecución.
```

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:



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.

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

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

```
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.
Sin 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 cómo 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 sí mismos sin importar lo que sucede dentro del bloque try . Se utiliza el bloque finally para cerrar archivos o liberar otros recursos del sistema.

```
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.

```
    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.

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.

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 sí mismas y tienen habilitado todo el manejo que se puede realizar con una clase. De esta manera, se puede sacar ventaja del hecho creando excepciones propias tan sólo con definir 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.

```
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 y sea atrapado para manejar la anomalía, mediante el método getMessage() .
Si se quiere utilizar la excepción que se creó anteriormente, se puede lanzar en un punto del código y recuperarla atrapándola en el método que realice la invocación al servicio que la define en su prototipo. El Código 6-12 muestra esta situación donde el método conectar lanza la excepción ante un error (notar que para hacerlo, crea un nuevo objeto del tipo de la excepción, para luego lanzarlo con la cláusula throw). Se deberá tener cuidado al realizar las declaraciones porque con throw se lanza una excepción desde un método, mientras que con throws se define en el prototipo que ese método puede lanzar una excepción de la que declara.

Uso de Exceptions para validar Reglas de negocio

En este ejemplo, se pretende controla la sobre venta de pasajes arrojando una exception personalizada.

```
//Exception de sobreventa de pasajes
package exceptions;
public class NoHayMasPasajesException extends Exception{
    private String vuelo;
    private int cantidad;
    public NoHayMasPasajesException(String vuelo, int cantidad) {
        this.vuelo = vuelo;
        this.cantidad = cantidad;
    }
    @Override
    public String toString() {
        return "El vuelo " + vuelo + ", no tiene " + cantidad + " pasajes.";
    }
    public String getVuelo() { return vuelo;    }
    public int getCantidad() { return cantidad; }
}

//La clase vuelo en el método vender pasajes lanza una NoHayMasPasajesException
//en caso de sobreventa de pasajes, el método esta sincronizado para poder
//usarlo en concurrencia.
package exceptions;
public class Vuelo {
    private String nombre;
    private int pasajes;
    public Vuelo(String nombre, int pasajes) {
        this.nombre = nombre;
        this.pasajes = pasajes;
    }
    public synchronized  void venderPasajes(int cantidad) throws
NoHayMasPasajesException{
        if(cantidad>pasajes) throw new
NoHayMasPasajesException(nombre,cantidad);
        pasajes-=cantidad;
```

```
    }
    @Override
    public String toString() {
        return "Vuelo{" + "nombre=" + nombre + ", pasajes=" + pasajes + '}';
    }
    public String getNombre()   { return nombre; }
    public int getPasajes()     { return pasajes; }
}

//Ejemplo de uso y test de la clase vuelo, el método venderPasajes debe
//controlarse la Exception que puede lanzar por sobreventa.
package exceptions;
public class TestVuelos {
    public static void main(String[] args) {
        Vuelo v1=new Vuelo("aer1234",100);
        try {
            v1.venderPasajes(10);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Captura de varias Exceptions en un mismo catch

Por herencia

```
        try {
            FileReader in=new FileReader("texto.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
        //En este ejemplo IOException captura IOException y cualquier Exception
        //que extienda de IOException por ejemplo FileNotFoundException.
```

Con Operador |

```
      //En este ejemplo se declaran dos Exceptions para un mismo bloque catch
      catch (IOException|SQLException ex) {
            logger.log(ex);
            throw ex;
      }

      catch(Exception1 e | Exception2 e | Exception3 e)    // DOES NOT COMPILE
      catch(Exception1 e1 | Exception2 e2 | Exception3 e3) // DOES NOT COMPILE
      catch(Exception1 | Exception2 | Exception3 e)        // Si compilación
```

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 un objeto de conexión JDBC, o una clase de gestión de archivo en el paquete java.io . Con cada tecnología externa se debe tomar los recaudos de manejo de excepciones que cada una exija.

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 (**.close()**), independientemente de si se produce una excepción durante la ejecución en curso.

```
Implementación de la interfaz AutoCloseable
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.

```
      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.

Una cosa que podría llamar la atención es el hecho que se está utilizando la palabra clave try sin incluir ni un catch o un finally , que no sería válido con las versiones anteriores del JDK.

De esta manera la instancia declarada en la sentencia *try-with-resource* será cerrada automáticamente cuando el bloque de sentencias del try sean ejecutadas completamente o cuando se produzca una excepción.
Se pueden declarar tantos recursos como sea necesario, cerrándose de manera inversa a como han sido declarados.

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.

Diferencias entre Closable y AutoCloseable

La interfaz AutoCloseable se introdujo en Java 7. Antes de eso, otra interfaz existió llamada Closeable que se puede cerrar. Era similar a lo que querían los diseñadores del lenguaje, con las siguientes excepciones:

- Closeable restringe el tipo de excepción lanzada a IOException.

Los diseñadores de lenguaje enfatizan la compatibilidad hacia atrás. Como no era deseable cambiar la interfaz existente, crearon una nueva llamada AutoCloseable. Esta nueva interfaz es menos estricta que Closeable. Como Closeable cumple con los requisitos para AutoCloseable, comenzó a implementar AutoCloseable cuando se introdujo este último.

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.

```
public static void main(String[] args) throws Exception{
      try (CierreAutomatico recurso1 = new CierreAutomatico();
            SegundoCierreAutomatico recurso2 = new
            SegundoCierreAutomatico()) {

      recurso1.gestionarRecurso();
      recurso2.manipulateResource();
      }
}
```

Cuando dos o más 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?

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.
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.

```
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);
      }
}
```

Detalles del manejo de Exceptions

- Para terminar un ejemplo de manejo de excepciones sin manejo de recursos.

```
public void oldApproach(Path path1, Path path2) throws IOException {
      BufferedReader in = null;
      BufferedWriter out = null;
      try {
            in = Files.newBufferedReader(path1);
            out = Files.newBufferedWriter(path2);
            out.write(in.readLine());
      } finally {
            if (in != null) in.close();
            if (out != null) out.close();
      }
}
```

y un ejemplo en donde se usa el manejo de recursos. (try with resources)

```
public void newApproach(Path path1, Path path2) throws IOException {
      try (BufferedReader in = Files.newBufferedReader(path1);
            BufferedWriter out = Files.newBufferedWriter(path2)
      ){
            out.write(in.readLine());
      }
}
```

- Los objetos creados en un bloque son destruidos al perder alcance (scope)

```
try (Scanner s = new Scanner(System.in)) {
            s.nextLine();
} catch(Exception e) {
```

```
      s.nextInt(); // DOES NOT COMPILE
} finally{
      s.nextInt(); // DOES NOT COMPILE
}
```

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 porque 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 dé 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.

Las aseveraciones están pensadas para la comprobación de invariantes (condiciones que se cumplen siempre en un determinado lugar del código), por lo que tienen más 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, porque 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>]
```

```
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());
      }
}

java simple.Notas       // no genera Assertion
java -ea simple.notas   // genera Assertion
```

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.

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.