

Excepciones - Tratamiento de problemas en tiempo de ejecución

El objetivo de este tema es aprender a prever los errores que se pueden producir durante la ejecución de una aplicación y la forma de enfrentarlos desde el código.

Tipo de Errores

En programación, los errores indican la presencia de problemas en la interpretación del programa. Existen tres categorías principales de errores:

1. **Errores de Sintaxis:** ocurren durante la etapa de compilación y están relacionados con la estructura y el formato del código. Algunos ejemplos incluyen:
 - Falta de punto y coma
 - Errores gramaticales, como nombres incorrectos de variables o métodos
 - Asignación de un valor incompatible a una variable

Con frecuencia se deben realizar varias iteraciones de corrección de errores de sintaxis antes de lograr una compilación exitosa. Por ejemplo:

```
// Error de sintaxis: Usar "=" en lugar de "=="
if (x = y) {
    // ...
} //end if

// Corrección: Usar "==" para comparar valores
if (x == y) {
    // ...
} //end if
```

Este tipo de errores está muy relacionado al lenguaje de programación y sus sintaxis.

2. **Errores de Lógica:** son causados por un razonamiento incorrecto del programador. Estos errores no resultan en fallos de compilación ni de tiempo de ejecución, pero pueden llevar a resultados inesperados. Ejemplos de errores de lógica incluyen:
 - Colocar un punto y coma después de una condición "if" o en la inicialización de un bucle, lo que hace que el intérprete trate el código posterior como fuera del bucle.
 - Usar el operador "==" para comparar objetos o cadenas de texto, en lugar del método "equals()" para comparar sus contenidos.

Un ejemplo de error de lógica sería:

```
// Error de lógica: Punto y coma fuera de lugar
for (int i = 0; i < 5; i++);
System.out.println(i); // Esta sentencia se ejecutará una vez, ¿por qué?

// Corrección: Eliminar el punto y coma
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

3. **Errores durante la ejecución - Excepciones:** En ocasiones, surgen situaciones inesperadas durante la ejecución de un programa. Por ejemplo, un elemento necesario para que el programa funcione puede faltar o la entrada de datos puede no cumplir con las expectativas. En otras situaciones, los métodos diseñados por el programador pueden no completarse correctamente, y es necesario informar a la sección que los llamó que la operación no se ha realizado con éxito. Estos errores deben ser manejados por el programador dentro del código que está desarrollando. Una excepción es un error que ocurre durante la ejecución de un programa, interrumpiendo el flujo normal de ejecución. Sin embargo, es posible manejar estas excepciones en el programa y tomar las medidas correctivas necesarias para que el programa pueda continuar su ejecución, lo cual se conoce como manejo de excepciones.

*En síntesis: Los **errores de sintaxis** impedirán la compilación. Los **errores de lógica** se compilarán y ejecutarán, pero el resultado puede no ser el esperado. Los programas con **errores de ejecución** se compilarán y comenzarán a ejecutarse, pero fallarán, produciendo una excepción. Si se maneja la excepción, se ejecutarán hasta finalizar.*

Comprobación de estados

La comprobación de estados hace referencia a las condiciones que deben cumplir los estados de ciertos objetos antes o después de realizar una operación.

Ejemplo:

```
resultado = a / b;
```

Es evidente que la condición previa que se debe cumplir es que **b** sea distinto de cero. Sin embargo, no siempre las precondiciones son tan fáciles de evaluar.

En otros casos, menos habituales, se deben chequear postcondiciones, como en la lectura de un dato correspondiente a una fecha (condición de tipo de dato).

Con la comprobación de estados se busca evitar o minimizar la posibilidad de que un objeto quede en estado inválido, por no haber terminado correctamente una tarea que afecta al objeto.

Este objetivo puede ser enfocado de dos maneras: en forma conservadora, o en forma optimista.

1. Enfoque conservador

Al enfoque conservador de comprobación de estados se lo puede caracterizar como: *comprobar primero, actuar después*. La idea de este enfoque es comprobar que se cumple la precondición y realizar la tarea solamente en ese caso. En el ejemplo de la división, se hubiera seguido un enfoque conservador si se hubiese chequeado que **b** no fuera cero antes de hacer la operación.

Es una forma muy común de encarar las dependencias de estados, y permite tomar acciones correctivas, pero sólo se puede aplicar a la verificación de precondiciones, no así de postcondiciones.

Hay tres políticas alternativas a seguir cuando una precondición no se cumple:

- Rechazo
- Suspensión protegida
- Plazos temporales

En un caso de **rechazo**, si no se cumplen las condiciones no se realiza la operación, enviando algún tipo de aviso al usuario o al módulo invocante. Es una solución extrema y poco recomendable, dado que no permite resolver el problema.

Ejemplo:

```
if (b != 0){
    resultado = a / b;
}
else{
    System.out.println ("Valor de divisor inválido");
}
```

Una implementación un poco más elegante de la misma política, y que permite resolver el problema en otro ámbito (desde donde es invocado el método), es:

```
public boolean unMetodo(){
    boolean unError = true;
    if (b != 0) {
        resultado = a / b;
    }
    else{
        unError = false;
    }
    return unError;
}

public void metodoInvocante(){
    if(unMetodo())
        // continua ejecución
    else{
        System.out.print("Divisor inválido");
    }
}
```

Utilizando **suspensión protegida** se espera a que la precondición sea verdadera para realizar la acción. Generalmente

en estos casos se da algún tipo de participación al usuario, como en el ejemplo que sigue:

```
while (b == 0) {  
    System.out.println("Valor de b inválido, ingréselo nuevamente");  
    leer(b);  
}  
resultado = a / b;
```

Este es un enfoque ingenuo, dado que presupone que el usuario sabe qué valor debe darle a una variable interna del programa.

Si se tratara de un programa concurrente, la precondition puede variar como resultado de otro hilo o proceso que modifique el estado de alguno de los objetos que participan de la misma. En estos casos, se implementaría así:

```
do {}  
while (b == 0); // espera a que el estado cambie  
resultado = a / b;
```

Los **plazos temporales** corresponden a una suspensión protegida, pero limitada a un intervalo de tiempo (en contextos concurrentes) o a un número de intentos, pasando luego al rechazo.

No obstante, el enfoque conservador no es la única ni necesariamente la mejor forma de resolver estos problemas.

2. Enfoque optimista

Este enfoque consiste en *intentar primero, analizar después*. Se realiza la operación independientemente de que se pueda hacer o no. Y si durante la ejecución surge un problema o al final de la misma no se cumple una postcondición necesaria, se intenta resolver el inconveniente.

Parece una forma rara de resolver problemas. Sin embargo, a menudo es la forma más razonable. Esto puede ocurrir, por ejemplo, si la precondition es muy costosa de evaluar y simultáneamente es poco probable que no se cumpla. Si en un caso así se evalúa siempre la precondition se estaría introduciendo un factor de ineficiencia a la aplicación, para tratar casos que habitualmente no se presentan. Por otra parte, es la única forma de comprobar postcondiciones. La forma más común de manejar la dependencia de estados con un enfoque optimista es mediante el uso de excepciones.

Excepciones

La razón de trabajar con excepciones es que habitualmente en el punto en que surge un error no siempre se sabe qué decisión tomar. Por ejemplo, en el caso de una función de biblioteca comprada, que deba obtener el promedio de una serie de valores contenidos en un vector. ¿Qué debería hacer la función si el vector que se le pasa está vacío? El problema surge porque en el contexto de la función no hay suficiente información para resolver la anomalía. Por lo tanto, se debe salir de la función hacia el módulo que la invocó, informando esta situación, pero sin provocar la finalización del programa ni enviar mensajes al usuario, que tampoco tiene información suficiente sobre el problema.

Por lo tanto, **las excepciones** son errores que se producen durante la ejecución de un programa, y el problema no se puede resolver en el contexto en el que se produce. Si se pudiera resolver sería un inconveniente normal, no una excepción. Es decir que cuando surge una excepción no hay forma de continuar y se debe elevar la misma a un contexto de nivel superior para que resuelva el inconveniente.

Por lo tanto, una excepción (o condición excepcional) es una situación inesperada o inusual que interrumpe el normal funcionamiento del programa, y que no puede ser resuelto en el contexto en el que se produce.

Una excepción se provoca cuando no se cumple una precondition o una postcondición.

Las operaciones de manejo de archivos, de reserva de memoria, los conflictos de hardware o del sistema operativo, etc., pueden generar excepciones que deban controlarse.

Una solución ingenua es pedirle al usuario final que resuelva el problema. Se dice que es ingenua debido a que no considera que hay muchos errores que un usuario final no es capaz de entender ni de resolver, además de que puede hablar otro idioma.

En la programación tradicional, el modo habitual de tratar condiciones de excepción fue devolviendo un valor especial, poniendo un valor en una variable global o mediante un parámetro definido a tal efecto.

Esto tenía más de un inconveniente:

- a menudo los programadores clientes no chequean estas variables o valores especiales. Esto puede parecer un comportamiento equivocado, pero pasa a ser bastante lógico cuando para chequear cada valor de retorno se convierte al código en ilegible. Un caso clásico son los programas concurrentes, en los cuales las posibilidades de excepciones pueden provocar más líneas que el código principal.

- en un sentido similar, el chequear las condiciones de error deja el código normal o básico muy acoplado con aquél que se ha escrito para manejar los errores, haciendo que se pierda la claridad del programa (alto acoplamiento).

El propósito de trabajar con excepciones es aislar el código que se usa para tratar las mismas, del código básico que resuelve el problema original. De esta manera el código básico queda más legible. Además, brinda una forma unificada de reportar errores. Se busca crear software más robusto de una forma sencilla. Para que un producto sea robusto, cada componente debe serlo. Cabe recordar que la robustez es uno de los factores que contribuye a la calidad del software.

El primer lenguaje conocido que soportó excepciones fue Basic. Otros lenguajes implementaron alguna función que se debía llamar en caso de errores. Desde sus primeras versiones el lenguaje Ada llevó el concepto de excepción a los lenguajes estructurados y lo perfeccionó. Luego fue implementado en los lenguajes de POO.

Importancia del Manejo de Excepciones

El manejo de excepciones proporciona un enfoque sencillo y transparente para tratar los errores. Las excepciones son el mecanismo mediante el cual se pueden controlar las condiciones de error en un programa. Estas condiciones de error pueden surgir debido a errores en la lógica del programa, como acceder a un índice de un array fuera de rango o realizar una división por cero. También pueden ser generadas por los propios objetos, indicando un estado inesperado o una condición que no pueden manejar.

Cuando se utilizan de manera adecuada, las excepciones mejoran significativamente la robustez de las aplicaciones, permitiendo que los programas se recuperen de situaciones inesperadas y continúen su ejecución de manera controlada. El manejo de excepciones es una parte esencial de la programación orientada a objetos, que permite lidiar con errores y situaciones imprevistas de manera efectiva, garantizando la integridad y la fiabilidad de las aplicaciones.

Cuando ocurre una excepción durante la ejecución de un programa, suceden dos cosas cruciales:

- Finalización de la Ejecución del Programa: En primer lugar, la ejecución del programa se interrumpe. Esto significa que el programa se detiene en el punto donde ocurrió la excepción.
- Impresión de un Rastreo de Pila: A continuación, se genera un rastreo de pila que contiene detalles precisos sobre la excepción. Este rastreo de pila muestra la secuencia de métodos y funciones que condujeron a la excepción. Esta información es extremadamente valiosa para depurar el programa.

El Rastreo de Pila

Un rastreo de pila es una representación visual de la cadena de llamadas a métodos o funciones que llevaron a la excepción. Proporciona un registro detallado de cómo se llegó al punto de error. Esto es esencial para los programadores, ya que les permite identificar la causa raíz de la excepción y corregir el problema.

```
1
2 public class DivisionEntreCeros{
3
4     public static void main(String args[]) {
5         int d = 0;
6         int a = 10 / d;
7         System.out.print(a);
8     }
9 }
10
```

```
java.lang.ArithmeticException: / by zero
    at DivisionEntreCeros.main(DivisionEntreCeros.java:6)
```

Excepciones y continuación del Programa

Es importante destacar que, en Java y en otros lenguajes de programación, la reacción ante una excepción puede variar. En algunos casos, el programa puede continuar ejecutándose después de que se haya producido una excepción y se haya generado un rastreo de pila. Sin embargo, esta no es la norma.

En situaciones donde el programa continúa después de una excepción, puede generar resultados inesperados o comportarse de manera incoherente. Esto es especialmente crítico en aplicaciones de interfaz gráfica de usuario (GUI), donde la continuidad de la ejecución a menudo no es deseable. En tales casos, el manejo adecuado de excepciones es esencial para garantizar un comportamiento controlado y predecible.

Formas de tratar excepciones

Hay varias formas de tratar excepciones:

- **Finalización súbita:** es la respuesta extrema, aplicable solamente si se está seguro de que el problema hace fallar a toda la actividad completa y los objetos implicados no van a ser usados nunca más. También es conveniente aplicarla si la pretensión de seguir trabajando puede empeorar las cosas. La forma de implementarla es, o bien dejar que el programa falle irreversiblemente, o provocar explícitamente una excepción que finalice la aplicación completa.
- **Continuación ignorando las fallas:** este tipo de respuesta, aparentemente irresponsable, es válida cuando no tiene consecuencias. Por ejemplo, cuando algo falla al intentar cerrar un archivo sobre el que sólo se realizaron consultas, o cuando se pierde un cuadro o parte del mismo durante una animación.
- **Vuelta atrás:** en este modelo, cuando se da la falla, el objeto vuelve al estado anterior. Este tratamiento suele ser más complejo si no viene soportado en el lenguaje: en general hay que guardar el estado del objeto (o la parte que se modifica del mismo) de alguna manera antes de la excepción, lo que implica que toda operación usada en el bloque problemático debe tener una especie de método inverso para garantizar la vuelta atrás, como un crédito que se anula con un débito. Por lo tanto, esta metodología no es aplicable cuando la acción problemática cambia irrevocablemente objetos, o el medio externo, como por ejemplo con acciones de entrada y salida.
- **Avance y recuperación:** esta respuesta consiste en seguir adelante, pero tratando de restablecer un estado legal y coherente en los objetos implicados. En general se aplica cuando la vuelta atrás no es factible.
- **Nuevo intento:** es un caso ampliado de la vuelta atrás, volviendo a probar luego que se restablecieron los estados de los objetos. Es una solución interesante en ámbitos de concurrencia. Por ejemplo, en un programa concurrente o distribuido, se puede esperar que otro hilo o proceso cambie los estados de los objetos mientras el hilo o proceso en cuestión sigue intentando. Se aplica también a los casos de fallos en máquinas remotas o en la conexión con las mismas, que se espera que puedan ser transitorios o reversibles. Para evitar quedar bloqueado consumiendo ciclos de procesador en exceso, se puede usar la técnica del retardo exponencial, que consiste en que cada retardo es proporcionalmente más largo que el anterior (como hacen los protocolos de Ethernet), o abandonar luego de un número de intentos o tiempo transcurrido. Una variante sería la de intentar de nuevo resolver el mismo problema, pero con un algoritmo distinto u otra versión del mismo, esperando que el problema no surja de nuevo.

Lanzamiento de excepciones

El objetivo de esta técnica es que el código que encuentra un problema y no lo pueda tratar *eleva, lance o arroje (throw)* una excepción, que será *atrapada o capturada (catch)* por el módulo que invocó al método que se está ejecutando. Dicho módulo podrá manejar el problema de acuerdo a cómo esté preparado para ello.

Por ejemplo, una función podría ser:

En lenguaje Ada 83	En lenguaje Java
<pre>function Dividir (X, Y : FLOAT) return FLOAT is begin if y = 0 then raise DivisionPorCero; - eleva una excepción else return X/Y; endif; end Dividir;</pre>	<pre>public float dividir (float x, float y){ if (y == 0){ throw new DivisionPorCero; } else{ return x/y; } }</pre>
donde <i>DivisionPorCero</i> es una excepción definida en un módulo más externo como:	
DivisionPorCero : exception;	class DivisionPorCero extend Exception { }

Se espera que el módulo que invocó a la función tenga un *manejador* que le permita ejecutar un código especial que trate el problema. En este ejemplo, la sección de código que utilice (invoca) a la función/método dividir, debe atrapar (catch) la excepción, y actuar en consecuencia (manejarla).

Cuando se dispara una excepción se corta el hilo de ejecución normal y se eleva la excepción hacia el contexto superior que invocó el método. De alguna manera, elevar una excepción es reconocer un fracaso en el módulo en cuestión, y como consecuencia se provoca una excepción en el módulo invocante, que es quien deberá manejarla. Quien toma el hilo de ejecución es ahora el manejador de excepciones del módulo invocante, y éste se ocupa de resolver el problema. Si no existiera un manejador de excepciones en el mismo, se dispara la misma excepción al nivel superior, y así sucesivamente. Si la excepción no fuera capturada en ningún nivel, llegaría a salir del programa, abortándolo.

Los lenguajes orientados a objetos prefirieron definir clases de excepciones. Por ejemplo, en Java:

```
if (p == null){
    throw new NullPointerException();
}
```

lo que hace es lanzar una excepción de la clase *NullPointerException* (nótese que se la está construyendo en el momento de lanzarla, con el operador new).

Por supuesto, en algún lugar debe estar definida la clase *NullPointerException*. En este caso es una excepción predefinida, del paquete estándar *java.lang*, pero en otras oportunidades será necesario definirla.

Los constructores de excepciones son siempre dos por defecto: uno sin parámetros (como el mostrado en el ejemplo anterior), y otro que recibe una cadena de caracteres que se usa para enviar información pertinente, como muestra el ejemplo que sigue:

```
if (p == null){
    throw new NullPointerException("Puntero nulo");
}
```

La modalidad de trabajo es disparar una excepción de una clase diferente por cada tipo de anomalía. En general es la propia clase de excepción la que se usa para determinar de qué error se trata por parte de quien la recibe. Por eso es especialmente importante elegir un nombre representativo para la clase de excepción.

En Java, en la declaración del método se indica qué excepciones arroja, de modo que el cliente de la clase sepa qué excepciones debe manejar. Para ello se usa la cláusula *throws*.

Ejemplo:

```
public static double potenciaReal(double base, double exponente) throws
                                ProblemaConPotencia {
    if (base < 0){
        throw new ProblemaConPotencia();
    }else{
        return (exponente * Math.exp(Math.log(base)));
    }
}
```

donde *ProblemaConPotencia* es una excepción definida en un módulo más externo como:

```
class ProblemaConPotencia extend Exception{}
```

Captura de excepciones y manejadores

El *manejador* es una porción del código que va a decidir qué hacer con la excepción que recibe. Por ejemplo, un fragmento de código Ada 83 podría usar la función *Dividir* que definimos más arriba y tratar la excepción *DivisionPorCero*:

```
begin
    resultado := Dividir(Numerador,Denominador);
exception
    when DivisionPorCero
        PUT("Error: el denominador es nulo");
end;
```

Los lenguajes más nuevos, como Java, manejan el concepto de **región segura**, la cual consiste en un bloque especial enmarcado por la palabra *try*. Los manejadores, entonces, se ocupan de tratar las excepciones lanzadas dentro del bloque *try*. La estructura del bloque *try* es como sigue:

```
try {
    // código que puede provocar excepciones
}
catch (Clase1 exc1) {
    // código que maneja las excepciones cuya clase es Clase1
}
catch (Clase2 exc2) {
    // código que maneja las excepciones cuya clase es Clase2
}
```

try

Es el bloque de código donde se prevé que se genere una excepción. Es como decir "intenta estas sentencias y mira a ver si se produce una excepción". El bloque try tiene que ir seguido, al menos, por una cláusula catch o una cláusula finally

catch

Es el código que se ejecuta cuando se produce la excepción. Es como decir "controlo cualquier excepción que coincida con mi argumento". Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula, como esta:

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

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas, y no permite tomar la decisión adecuada.

En Java todas las clases de excepciones derivan de un ancestro por omisión que se llama *Exception*. De hecho, una excepción es tal por el hecho de declararla como descendiente de *Exception*. Como cualquier excepción puede ser capturada por un manejador diseñado para una excepción de una clase ancestro, poner un manejador de excepciones *Exception* es una buena idea para capturar las excepciones no mencionadas explícitamente.

Ejemplo:

```
try {
    z = potenciaReal(x,y);
}
catch (ProblemaConPotencia e) {
    System.err.println("Hay un problema con la base de la potencia");
    z = 0;
}
catch (Exception e) { // captura cualquier excepción salvo ProblemaConPotencia
    System.err.println("Problema desconocido");
}
```

Hay una clase de excepción predefinida llamada *RuntimeException*, de la cual descienden todas las clases de excepciones provocadas por errores de programación como *NullPointerException* y *IndexOutOfBoundsException*. Son excepciones que Java lanza siempre, y si no se tratan terminan en un error de programa que provoca un aborto.

Cada vez que se provoca una excepción, el programa compara la excepción generada con cada manejador hasta encontrar el primero que se ajusta al tipo de error. Una vez que la excepción fue manejada, nadie más la puede manejar y el problema se considera terminado.

Existe una cláusula *finally* que contiene código que se ejecuta siempre, al final de la región segura, independientemente de que se hayan disparado o no excepciones. Esta porción de código, que se ejecuta luego de todos los manejadores y antes de que se lancen excepciones al contexto de nivel superior, suele ser muy útil para liberar recursos, como cerrar un archivo o una conexión de red y, en los lenguajes que no tienen recolección automática de basura, para liberar memoria.

Se debe tener especial cuidado con las excepciones lanzadas o capturadas por constructores, pues un objeto no se crea realmente hasta que no finaliza correctamente su constructor. Esto hace que un constructor pueda haber comprometido recursos que luego no se utilicen por causa de la excepción.

Una excepción en Java no puede ser ignorada como si fuera un argumento booleano. Una excepción que se recibe de la cláusula *throws* de un método invocado debe ser capturada o vuelta a lanzar.

Por lo tanto, esta versión de método **no es válida**:

```
// el código que sigue no compila:
public double suma() {
    double s = 0;
    for (int i = 0; i < 10; i++){
        s += potenciaReal(x[i],y[i]);
    }
    return s;
}
```

Este código no es válido debido a que no se captura la excepción. Hay dos soluciones posibles. Una es capturar la excepción recibida de *potenciaReal*, como sigue:

```
public double suma() {
    double s = 0;
    try {
        for (int i = 0; i < 10; i++){
            s += potenciaReal(x[i],y[i]);
        }
    }
    catch (ProblemaConPotencia e ){
        System.err.println("Hay un problema con la base de la potencia ");
    }
    return s;
}
```

La otra posibilidad es declarar que se lanza nuevamente la excepción, lo cual obliga al módulo de nivel superior a capturarla:

```
public double suma() throws ProblemaConPotencia {
    double s = 0;
    for (int i = 0; i < 10; i++){
        s += potenciaReal(x[i],y[i]);
    }
    return s;
}
```

Esta segunda opción se utiliza cuando no se sabe qué hacer con la excepción. No se debe olvidar que el objetivo de lanzar una excepción es enviarla a un ámbito en el cual se sepa cómo tratarla. Por lo tanto, si en un método no se sabe cómo tratar una excepción, lo más coherente es enviarla un nivel más arriba.

Hay programadores que evitan estas normas declarando simplemente que el método arroja la excepción raíz *Exception*, o bien capturan siempre dicha excepción. El problema con este enfoque es que, al ser tan general, se pierden las ventajas de documentación y control que da una mayor granularidad en el tratamiento de problemas.

La obligación de tratar las excepciones de los métodos invocados tiene dos ventajas:

- Hace a los programas más robustos
- Es el compilador el que avisa que no se está capturando la excepción, por lo que no es posible olvidarse, y el problema nunca llega al tiempo de ejecución.

Gestión de Recursos y la Cláusula Finally en Excepciones

En la programación, es importante garantizar que los recursos adquiridos se liberen adecuadamente para evitar fugas de recursos. En lenguajes como C y C++, la fuga de memoria es un problema común. Java aborda este problema mediante la recolección automática de basura, que elimina la memoria no utilizada. Sin embargo, en Java, aún es posible experimentar fugas de recursos relacionadas con archivos, conexiones de bases de datos y redes que no se cierran adecuadamente cuando ya no son necesarios, lo que podría afectar la disponibilidad de recursos para otros programas. El bloque finally, opcional y a veces llamado cláusula finally, es una parte esencial de la gestión de excepciones. Se ubica después de los bloques catch o directamente después del bloque try si no hay bloques catch asociados.

El bloque finally se ejecutará sin importar si se lanza o no una excepción en el bloque try correspondiente. También se ejecutará si el control abandona el bloque try debido a una instrucción return, break, continue, o simplemente al llegar al final del bloque try.

Sin embargo, hay una excepción: si la aplicación sale prematuramente del bloque try utilizando System.exit, el bloque finally no se ejecutará.

Atributos y métodos en excepciones creadas por el programador

El objetivo de que el programador cree sus propias clases de excepciones es indicar con precisión el tipo particular de error que un método puede elevar.

Una nueva clase de excepción debe extender a una clase de excepción existente, para poder asegurar que la clase pueda utilizarse con el mecanismo de manejo de excepciones. Una clase de excepción es igual que cualquier otra clase; sin embargo, una nueva clase común de excepción contiene sólo cuatro constructores:

- uno que no toma argumentos y pasa un objeto String como mensaje de error predeterminado al constructor de la superclase
- uno que recibe un mensaje de error personalizado como un objeto String y lo pasa al constructor de la superclase
- uno que recibe un mensaje de error personalizado como un objeto String y un objeto Throwable (para encadenar excepciones), y pasa ambos objetos al constructor de la superclase
- uno que recibe un objeto Throwable (para encadenar excepciones) y pasa sólo este objeto al constructor de la superclase.

En Java, cuando se lanza una excepción se crea un objeto de su clase, que es capturado por el manejador. El propósito de esta práctica es poder almacenar datos en los atributos del objeto, que puedan ser utilizados por el contexto de nivel superior como información adicional. Por ejemplo:

```
public class ProblemaConPotencia extends Exception {
    private double base;
    private double exponente;
    // constructor vacío:
    public ProblemaConPotencia() { }
    // constructor con un parámetro:
    public ProblemaConPotencia (String mensaje) {
        super(mensaje); // llama al constructor de la superclase (Exception), que tiene un string
    }
    // constructor con tres parámetros
    public ProblemaConPotencia (String mensaje, double p_base, double
                                p_exponente) {
        super(mensaje);
        this.setBase(p_base);
        this.setExponente(p_exponente);
    }
    public double getBase() {
        return this.base;
    }
    public double getExponente () {
        return this.exponente;
    }
}
```

A continuación, se muestra una función que utiliza esa excepción:

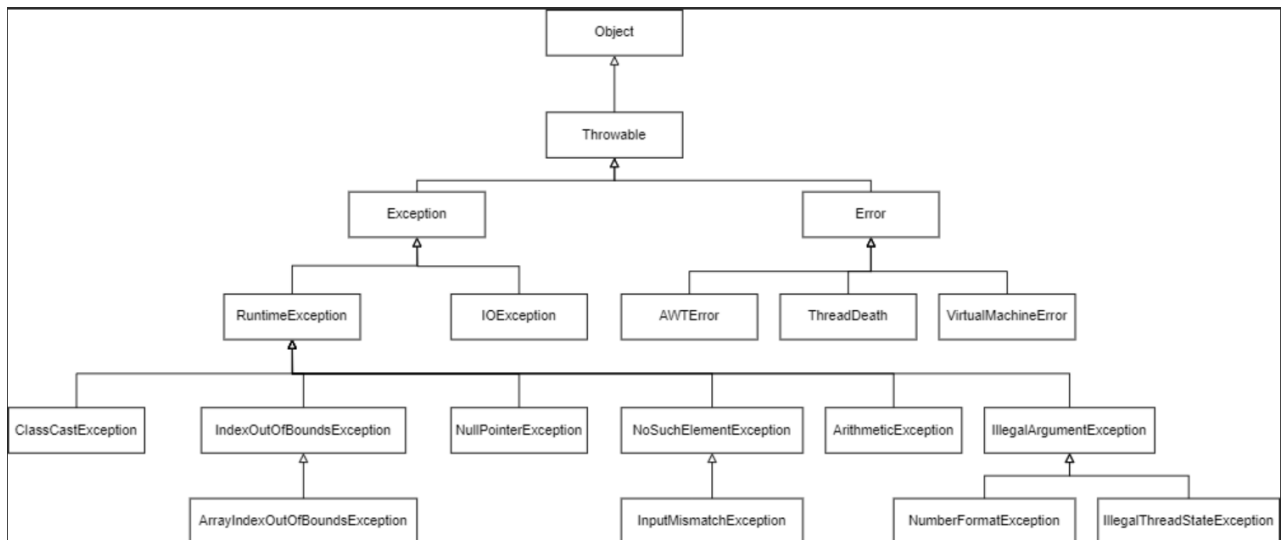
```
public class Math2 {
    // ... varios métodos previos
    public static double potenciaReal(double base, double exponente) throws
                                    ProblemaConPotencia {
        if (base < 0){
            throw new ProblemaConPotencia("Base negativa", base, exponente);
        }else{
            return (exponente*Math.exp(Math.log(base)));
        }
    }
}
```

Más adelante, quien llame al método *potenciaReal* podrá manejar la excepción de esta manera:

```
public class Prueba{
    ...
    try {
        double w = Math2.potenciaReal(u,v);
    }catch (ProblemaConPotencia e) {
        System.err.println("problema con base: " +e.getBase()+
                           " y exponente: "      + e.getExponente());
    }
}
```

Jerarquías de excepciones en Java

Java define una serie de excepciones y errores en forma estándar. Todos ellos derivan de la clase *Throwable*, según el siguiente diagrama, que muestra la jerarquía de clases de excepciones predefinidas que son las generadas con mayor frecuencia:



Las excepciones de clases descendientes de *Error* no se deberían capturar pues indican problemas muy serios e irreversibles. Tampoco se pueden capturar al usar la cláusula *catch* para una *Exception* genérica, pues no derivan de esa clase.

Algunas de las excepciones predefinidas más frecuentes son las siguientes:

ArithmeticException

Las excepciones aritméticas son típicamente el resultado de una división por 0:

NullPointerException

Se produce cuando se intenta acceder a una variable o método antes de ser definido:

```

class Hola extends Applet {
    Image img;

    paint( Graphics g ) {
        g.drawImage( img, 25, 25, this );
    }
}
  
```

NoClassDefFoundException

Se referenció una clase que el sistema es incapaz de encontrar.

ArrayIndexOutOfBoundsException

Es la excepción que más frecuentemente se produce. Se genera al intentar acceder a un elemento de un array más allá de los límites definidos inicialmente para ese array.

Categorías de Excepciones en Java

Java clasifica las excepciones en dos categorías principales: excepciones verificadas (o comprobadas) y excepciones no verificadas (o no comprobadas). Esta distinción es crucial, ya que el compilador de Java impone requisitos especiales para las excepciones verificadas. El tipo de excepción determina su clasificación.

1. Excepciones No Verificadas

Las excepciones *RuntimeException* son consideradas excepciones no verificadas. Estas generalmente se originan debido a defectos en el código de los programas. Algunos ejemplos de excepciones no verificadas incluyen:

ArrayIndexOutOfBoundsException: Este error ocurre cuando se intenta acceder a un índice fuera del rango válido de un arreglo. Para prevenirlo, es necesario asegurar que los índices estén dentro de los límites del arreglo.

ArithmeticException: Esta excepción surge al intentar dividir entre cero. Para evitarla, se debe verificar que el denominador no sea cero antes de realizar el cálculo.

Las excepciones que heredan directa o indirectamente de la clase `Error` también se consideran no verificadas. Estos errores son tan graves que un programa no debería intentar manejarlos.

2. Excepciones Verificadas

Las excepciones verificadas son todas aquellas clases que heredan de la clase `Exception` pero no lo hacen directa o indirectamente de la clase `RuntimeException`. Estas excepciones generalmente se relacionan con condiciones que están más allá del control del programa. Por ejemplo, al procesar archivos, un programa no puede abrir un archivo si este no existe.

La diferencia clave entre excepciones verificadas y no verificadas radica en cómo se gestionan en el código. Las excepciones verificadas deben ser declaradas o capturadas explícitamente, mientras que las excepciones no verificadas no tienen este requisito.

Uso de Aserciones

Las aserciones son declaraciones que se utilizan para establecer condiciones que deben ser verdaderas en puntos específicos de un método. Las precondiciones y poscondiciones son tipos de aserciones.

En Java, se pueden utilizar aserciones para implementar precondiciones y poscondiciones programáticamente o para verificar estados intermedios del programa **durante el desarrollo y la depuración**. Las aserciones pueden ser habilitadas o deshabilitadas en tiempo de ejecución y son útiles para atrapar errores lógicos en la aplicación.

Ejemplo: se utiliza una aserción para verificar si un número es positivo. Si la aserción falla, se lanzará una excepción `AssertionError`.

```
public class EjemploAsercion {
    public static void main(String[] args) {
        int numero = -5;
        assert numero > 0 : "El número debe ser positivo";

        System.out.println("El número es positivo.");
    }
}
```

Las aserciones se consideran **una herramienta de desarrollo y depuración**, no una característica que deba estar habilitada en un entorno de producción. Es importante que los desarrolladores utilicen aserciones con sensatez y las empleen principalmente para validar suposiciones internas **durante el desarrollo y las pruebas de una aplicación**, con la comprensión de que se pueden deshabilitar cuando la aplicación se despliega en producción para garantizar un rendimiento óptimo y la seguridad del sistema. Algunas de sus características son:

- **Rendimiento:** Las aserciones agregan una sobrecarga al rendimiento de la aplicación. Incluso si una aserción es verdadera, se incurre en un costo en términos de tiempo de CPU para evaluar la expresión booleana de la aserción. En situaciones de producción donde el rendimiento es crítico, esta sobrecarga puede ser indeseable.
- **Seguridad:** En un entorno de producción, es fundamental garantizar que la aplicación sea segura y no presente vulnerabilidades. Si las aserciones estuvieran habilitadas en producción, un atacante podría explotar posibles debilidades en el código al proporcionar entradas maliciosas que causen fallas en las aserciones y, en consecuencia, comportamientos inesperados o incluso errores graves.
- **Interrupción del Flujo de la Aplicación:** Las aserciones interrumpen el flujo normal de la aplicación cuando fallan. Esto significa que, si una aserción falla en un entorno de producción, la aplicación puede detenerse de manera inesperada, lo que resulta en una experiencia de usuario negativa y una posible pérdida de datos o trabajo no guardado.
- **Depuración:** En la fase de desarrollo y pruebas, las aserciones son muy útiles para identificar errores y problemas en el código.