Módulo profesional Programación

UD 10 – Excepciones

Clara-mente tu programa puede que funcione, pero duele.

Contenido

- Concepto excepción
- Gestión de excepciones
  - Tratarlas
  - Lanzarlas
- Tipo de excepciones
- Clases de excepciones
  - Excepciones verificadas
  - Excepciones no verificadas
- Pila de llamadas a métodos
- Crear nuestras propias excepciones

Ejemplo de excepción en tiempo real

Esta interrupción no es más que una excepción.

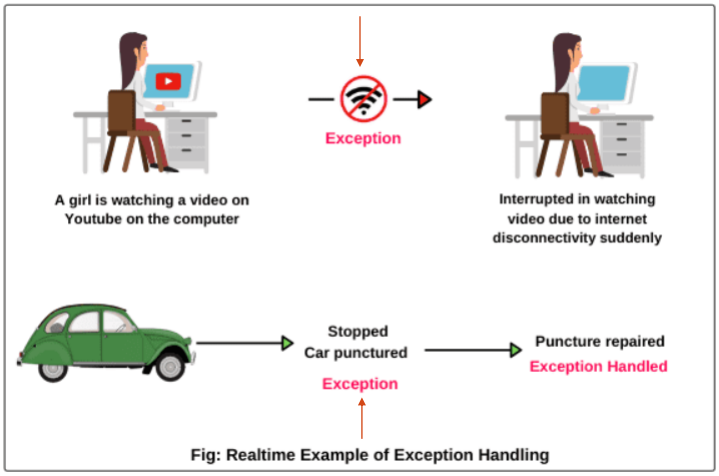


Fig: Realtime Example of Exception Handling

Este acontecimiento inesperado o no deseado no es más que una excepción.

Ejemplo de excepción en tiempo real

El propietario del coche siempre guarda un neumático extra como alternativa en un viaje de larga distancia.

Cambia el neumático pinchado por uno nuevo. Después de cambiar el neumático, continúa el resto del viaje.

Este modo alternativo se denomina manejo de excepciones.

Ejemplo de excepción en tiempo real

Del mismo modo, cuando creamos un programa java y se compila con éxito, incluso pueden producirse excepciones en tiempo de ejecución debido a errores en la lógica del programa. Esta excepción debe ser manejada para mantener el flujo de ejecución normal del programa.

Al manejar la ocurrencia de la excepción, podemos proporcionar un mensaje significativo al usuario sobre el error en lugar de un mensaje de error generado por el sistema que no es fácil de entender para un usuario.

Concepto de excepción

Una **excepción** es una condición anormal que se produce en una porción de código durante su ejecución y que puede abortar la ejecución del programa.

La **anomalía** ocurre cuando tu programa se está ejecutando.

Concepto de excepción

Vamos a entender la excepción de Java con un simple programa de ejemplo y ver cómo se crea y lanza un objeto de excepción.

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter two integer numbers");

        // Read two integer numbers.
        int num1 = sc.nextInt();
        int num2 = sc.nextInt();
        System.out.println(num1 + "/" + num2 + " = " + (num1/num2));
    }
}
```

En el programa anterior, cuando introducimos 0 para el segundo número, se produce un error de ejecución porque no podemos dividir un entero por 0.

```
Output 1:
    Enter two integer numbers
    4
    2
    4/2 = 2
Output 2:
    Enter two integer numbers
    2
    0
    Exception in thread "main" java.lang.ArithmeticException: / by zero
```

¿Por qué se produce una excepción en el programa?

¿Por qué se produce una excepción en el programa?

Puede haber muchas razones que puedan generar una excepción en un programa Java.

1. Abrir/Leer un archivo inexistente en tu programa.
2. Escribir datos en un disco pero el disco está lleno o sin formatear.
3. Cuando el programa pide la entrada del usuario y éste introduce datos no válidos.
4. Cuando un usuario intenta dividir un valor entero por cero, se produce una excepción.
5. Cuando un flujo de datos tiene un formato no válido.

Lo que realmente importa es "¿qué sucede después de que ocurrió una anomalía?"

Concepto de excepción

En otras palabras, **"cómo se manejan las situaciones anormales en tu programa".**

Si estas excepciones no se manejan adecuadamente, el **programa termina abruptamente** y puede causar graves consecuencias.

Concepto de excepción

¿qué **consecuencias**?

Por ejemplo, las conexiones de red, las conexiones de base de datos y los archivos pueden permanecer abiertos; La base de datos y los registros de archivos pueden quedar en un estado incoherente.

Concepto de excepción

Java tiene un mecanismo incorporado para manejar los errores de tiempo de ejecución, conocido como **manejo de excepciones**.

Esto nos permite garantizar que se puedas escribir programas robustos para aplicaciones de misión crítica, por ejemplo.

Concepto de excepción

Los lenguajes de programación más antiguos como C tienen algunos inconvenientes en el manejo de excepciones. Por ejemplo, supongamos que un programador desea abrir un archivo para procesarlo. Podemos encontrarnos con:

1. Los programadores no son conscientes de las condiciones excepcionales. Por ejemplo, el archivo que se quiere abrir puede no existir. Por lo tanto, el programador no escribe código para probar si el archivo existe antes de abrirlo.
2. Supongamos que el programador es consciente de las condiciones excepcionales, pero decide terminar primero la lógica principal y escribir el código de manejo de excepciones más tarde; esto "más tarde", desafortunadamente, generalmente nunca sucede.
3. Supongamos que el programador decide escribir código de manejo de condiciones excepcionales, el código de manejo de excepciones se entrelaza con la lógica principal en muchas sentencias if-else. Esto hace que la lógica principal sea difícil de seguir y que todo el programa sea difícil de leer.

Concepto de excepción

```
if (file exists) {
   open file;
   while (there is more records to be processed) {
      if (no IO errors) {
         process the file record
      } else {
         handle the errors
      }
   }
   if (file is opened) close the file;
} else {
   report the file does not exist;
}
```

Lógica principal

Concepto de excepción

Java supera estos inconvenientes al integrar el manejo de excepciones en el lenguaje en lugar de dejarlo a la discreción de los programadores:

- Se te informará de las condiciones excepcionales que pueden surgir al llamar a un método: las excepciones se declaran en la cabecera del método.
- Te verás obligado a manejar las excepciones mientras escribes la lógica principal y no puede dejarlas como una idea de último momento: tu programa no puede compilarse sin los códigos de manejo de excepciones.
- Los códigos de manejo de excepciones están separados de la lógica principal: a través de la construcción **try-catch-finally.**

Tratamiento de excepciones (verificadas)

Podemos gestionar las excepciones de dos modos:

- **Tratarlas** donde se producen (try-catch)
- **Lanzarlas** para que se traten en otro método superior (throws).

Tipos de excpciones

Tipos de excepciones:

- IlegalArgumentException: Error en la llamada de un método por un parámetro erróneo.
- IndexOutOfBounds: Error de posición incorrecta en un array o String.
- NullPointerException: Error al acceder a un objeto null (no inicializado).
- NumberFormatException: Error al convertir una cadena a número. `Integer.parseInt("abc");`
- ArithmeticException: Error matemático (Ejemplo: dividir por 0).
- IOException: Error genérico de E/S
- EOFException: Error por final inesperado de archivo.
- FileNotFoundException: Error por archivo no encontrado.

Lanzar excepciones

Cuando decidimos lanzar una excepción para que sea tratada en métodos superiores, es necesario indicarlo en la cabecera del método. Supondamos que la siguiente es la definición de methodD

```
public void methodD() throws XxxException, YyyException {
   // method body throw XxxException and YyyException
}
```

La cabecera del método indica que la ejecución de methodD() en su cuerpo puede encontrar dos excepciones comprobadas: XxxException y YyyException. En otras palabras, algunas de las condiciones anormales en methodD () pueden desencadenar XxxException o YyyException.

```
public void methodD() throws XxxException, YyyException {
   // method body throw XxxException and YyyException
}
```

Lanzar excepciones

Supongamos que el methodC () que llama a methodD () no desea manejar las excepciones (a través de un try-catch), puede declarar que estas excepciones se en su cabecera de la siguiente forma:

```
public void methodC() throws XxxException, YyyException {   // for next higher-level method to handle
   ...
   // uses methodD() which declares "throws XxxException, YyyException"
   methodD();   // no need for try-catch
   ...
}
```

Tratar excepciones

El tratamiento de excepciones en el lugar se realiza mediante los bloques **try, catch, finally.**

```
public void methodC() {  // no exception declared
   ......
   try {
      ......
      // uses methodD() which declares XxxException & YyyException
      methodD();
      ......
   } catch (XxxException ex) {
      // Exception handler for XxxException
      ......
   } catch (YyyException ex} {
      // Exception handler for YyyException
      ......
   } finally {   // optional
      // These codes always run, used for cleaning up
      ......
   }
   ......
}
```

Tratar excepciones

- En el bloque **try** se encapsula todo el código susceptible de producir alguna excepción sea la que sea, incluso si puede generar varias excepciones de tipos diferentes.
- Se define un bloque **catch** por cada excepción diferente que se pretende tratar, empezando por la más específica y acabando por la más genérica.
- Si un mismo error se puede tratar en más de un catch se tratará con el primero de ellos. Si no se produce ningún error no se ejecutará ningún catch (Poner ejemplo)
- El bloque **finally** es opcional. En el caso de introducirse, el código que encapsula se ejecutará siempre, se produzca una excepción del tipo que sea o no se produzca ninguna excepción, incluso si hay un **return**.

```
public Scanner(File source) throws FileNotFoundException;
```

Tratar excepciones. Ejemplo

Ejemplo

Si un método declara una excepción en su cabecera, no puedes usar este método sin manejar la excepción, el **programa no compila**.

```
import java.util.Scanner;
import java.io.File;
public class ScannerFromFile {
   public static void main(String[] args) {
      Scanner in = new Scanner(new File("test.in"));
      // do something ...
   }
}
```

```
ScannerFromFile.java:5: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown
     Scanner in = new Scanner(new File("test.in"));
                  ^
```

Tratar excepciones. Ejemplo

Para usar un método que declare una excepción en su cabecera, DEBES:

- proporcionar código de manejo de excepciones en un bloque "try-catch" o "try-catch-finally", o
- no manejar la excepción en el método actual, y lanzarla (throws) para que el siguiente método de nivel superior la maneje.

Tratar excepciones. Ejemplo

Tratar la excepción con un bloque try-catch

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class ScannerFromFileWithCatch {
   public static void main(String[] args) {
      try {
         Scanner in = new Scanner(new File("test.in"));
         // do something if no exception ...
         // you main logic here in the try-block
      } catch (FileNotFoundException ex) { // error handling separated from the main logic
         ex.printStackTrace();             // print the stack trace
      }
   }
}
```

Si no se puede encontrar el archivo, la excepción queda atrapada en el bloque catch. En este ejemplo, el controlador de errores simplemente imprime el seguimiento de la pila, que proporciona información útil para la depuración. En algunas situaciones, es posible que deba realizar algunas operaciones de limpieza o, en su lugar, abrir otro archivo. Ten en cuenta que la lógica principal en el bloque try está separada de los códigos de manejo de errores en el bloque catch.

Tratar excepciones. Ejemplo

Has decidido no manejar la excepción en el método actual, en cambio lanzarla para que el siguiente método de nivel superior la maneje:

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class ScannerFromFileWithThrow {
   public static void main(String[] args) throws FileNotFoundException {
                              // to be handled by next higher-level method
      Scanner in = new Scanner(new File("test.in"));
                           // this method may throw FileNotFoundException
      // main logic here ...
   }
}
```

- En este ejemplo, decidiste no manejar la excepción FileNotFoundException lanzada por el método de Scanner (File) (con try-catch).
- En cambio, el método que llama Scanner(File) -el método main()- declara en su cabecera "throws FileNotFoundException", lo que significa que esta excepción se lanzará para que el próximo método de nivel superior sea manejado. En este caso, el siguiente método de mayor nivel de main() es la JVM, que simplemente finaliza el programa e imprime la excepción

Tratar excepciones. Ejemplo

Como se muestra en el ejemplo, la lógica principal está contenida en el bloque de try , mientras que los códigos de manejo de excepciones se mantienen en el (los) bloque (es) catch separados de la lógica principal. Esto mejora enormemente la legibilidad del programa. Por ejemplo, un programa Java para el procesamiento de archivos podría ser el siguiente:

```
try {
   // Main logic here
   open file;
   process file;
   ......
} catch (FileNotFoundException ex) {   // Exception handlers below
   // Exception handler for "file not found"
} catch (IOException ex) {
   // Exception handler for "IO errors"
} finally {
  close file;      // always try to close the file
}
```

Tratar excepciones. Ejemplo

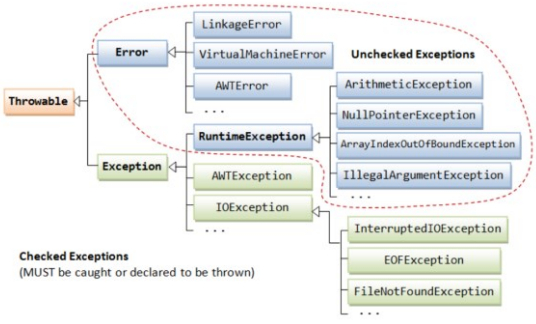
Clases y métodos a tener en cuenta

- **Exception**: Esta excepción genérica engloba a todas las excepciones, incluidas las definidas por el propio programador.
- **void printStackTrace():** Este método se puede utilizar dentro de cualquier bloque catch y muestra por pantalla la pila (stack) de llamadas incluyendo los números de línea y ficheros donde se ha producido la excepción. Es decir, la información completa de la excepción.
- **String getMessage():** Este método se puede utilizar dentro de cualquier bloque catch y devuelve una cadena de caracteres con la descripción de la excepción.
- **System.err.println("mensaje"):** Muestra un mensaje por la salida de error en lugar de la salida estándar. Será el usado por defecto en los catch
- **System.exit(1):** Esta instrucción aborta la ejecución actual. Se utilizará en el catch cuando no sea posible continuar la ejecución normal por la excepción ocurrida.

Clases de excpecioneS: Throwable, Error, Exception & RuntimeException

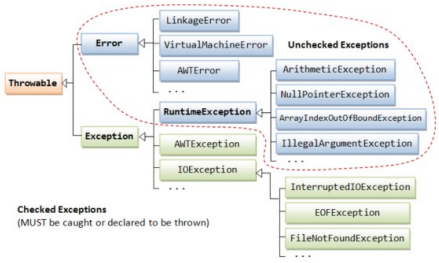
- La clase **Error** describe errores internos del sistema (por ejemplo, VirtualMachineError, LinkageError) que rara vez ocurren. Si se produce un error de este tipo, es poco lo que puede hacer y el programa de ejecución de Java terminará el programa.
- La clase de **Exception** describe el error causado por tu programa (por ejemplo, FileNotFoundException, IOException).

El programa puede detectar y manejar estos errores (por ejemplo, realizar una acción alternativa o hacer una salida correcta cerrando todos los archivos, la red y las conexiones de la base de datos).



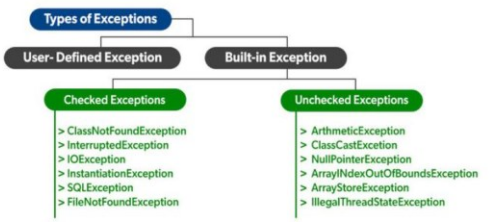
Excepciones verificadas vs. no verificadas

- Las subclases de **Error** y **RuntimeException** se conocen como **excepciones no verificadas** (unckecked). El compilador no comprueba estas excepciones y, por lo tanto, no es necesario que sean capturadas o lanzadas en tu programa.
- Todas las demás excepciones se denominan **excepciones verificadas (checked).** Son revisadas por el compilador y deben ser capturadas o lanzadas en tu programa.



Excepciones verificadas vs. no verificadas

- Una excepción de tipo **verificada** representa un error del cual técnicamente podemos recuperarnos. Por ejemplo, una operación de lectura/escritura en disco puede fallar porque el fichero no exista.
- Todas las excepciones de tipo **verificadas** deben se capturadas o relanzadas.
- Una excepción de tipo **no verificada** representa u error de programación. Uno de los ejemplos más típico es el de intentar leer en un array de n elementos u elemento que se encuentra en una posición mayor qu n.
- El aspecto más destacado de las excepciones de tipo **no verificadas** es que no deben ser forzosamente declaradas ni capturadas (en otras palabras, no son verificadas). Por ello no son necesarios bloques try-catch ni declarar formalmente en la cabecera del método el lanzamiento de excepciones de este tipo.



Actividad. Con catch y con throws

1. Copia estos dos ejemplos y ejecútalos para que encuentre el archivo y no lo encuentre

```java
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class ScannerFromFileWithCatch {
   public static void main(String[] args) {
      try {
         Scanner in = new Scanner(new File("test.in"));
         // do something if no exception ...
         // you main logic here in the try-block
      } catch (FileNotFoundException ex) { // error handling separated from the main logic
         ex.printStackTrace();             // print the stack trace
      }
   }
}
```

```java
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class ScannerFromFileWithThrow {
   public static void main(String[] args) throws FileNotFoundException {
                                    // to be handled by next higher-level method
      Scanner in = new Scanner(new File("test.in"));
                                   // this method may throw FileNotFoundException
      // main logic here ...
   }
}
```

Actividad. Try-catch

RecorreCadena

Crea una clase **RecorreCadena** con un método main que:

- Crea una cadena con una frase
- Recorre la cadena mediante un bucle infinito mostrando cada carácter de la misma
- Cuando se alcance el final de la cadena se producirá una excepción que detendrá la aplicación.

A continuación:

- Lee en la consola el tipo de excepción que se ha producido.
- Añade a la aplicación el código necesario para que, cuando se produzca la excepción, muestre el mensaje "Fin de la cadena".

Actividad. Try-catch

ProcesaEnteros

- Crea una clase **ProcesaEnteros** que pida números enteros al usuario en una línea, los lea de a uno por vez, los sume y muestre la suma.
- Por ejemplo, para la entrada 10 20 30 40 debería producir la salida “La suma de los enteros en la línea es: 100”

Actividad. Try-catch

ProcesaEnteros

- Prueba con la línea “Tenemos 2 perros y 1 gato”: Deberías obtener una excepción NumberFormatException cuando intentes llamar a Integer.parseInt con “Tenemos", que no es un entero.
- Una forma de evitar esto es poner el bucle que lee dentro de un try y capturar la NumberFormatException pero no hacer nada con ella. De esta manera, si no es un entero, no causa un error; va al manejador de excepciones, que no hace nada.
- Modifica el programa para añadir una sentencia try que abarque todo el bucle while. El try y la apertura { deben ir antes del while, y el catch después del cuerpo del bucle. Atrapa una NumberFormatException y vacía la sentencia catch.

Actividad. Try-catch

ProcesaEnteros

- Verás que termina el programa, por lo que la línea anterior producirá una suma de 0, y la línea "1 pez 2 peces" producirá una suma de 1. Esto se debe a que todo el bucle está dentro de try, por lo que cuando se lanza una excepción el bucle termina.
- Para hacer que continúe, mueve el try y el catch dentro del bucle. Ahora cuando se lanza una excepción, la siguiente sentencia es la siguiente iteración del bucle, por lo que se procesa toda la línea. La entrada perros y gatos debería dar una suma de 3.

Try-catch-finally

La sintaxis de try-catch-finally es:

- Si no se produce ninguna excepción durante la ejecución del bloque try , todos los bloques de catch se omiten y el bloque finally se ejecutará después del bloque de try.
- Si una de las sentencias en el bloque try provoca una excepción, Java ignora el resto de las sentencias en el bloque try y comienza a buscar un controlador de excepciones coincidente. Compara el tipo de excepción con cada uno de los bloques catch secuencialmente.
- Si un bloque catch captura esa clase de excepción o captura una superclase de esa excepción, se ejecutará la instrucción en ese bloque catch.

```
try {
   // main logic, uses methods that may throw Exceptions
   ......
} catch (Exception1 ex) {
   // error handler for Exception1
   ......
} catch (Exception2 ex) {
   // error handler for Exception1
   ......
} finally {   // finally is optional
   // clean up codes, always executed regardless of exceptions
   ......
}
```

Try-catch-finally

- Las instrucciones en el bloque finally se ejecutan después de ese bloque catch. El programa continúa en la siguiente sentencia después del try-catch-finally.

- Si ninguno de los bloques catch coincide, la excepción pasará a la pila de llamadas. El método actual ejecuta la cláusula finally (si existe) y es quitado la pila de llamadas. El método que llama sigue los mismos procedimientos

```
try {
   // main logic, uses methods that may throw Exceptions
   ......
} catch (Exception1 ex) {
   // error handler for Exception1
   ......
} catch (Exception2 ex) {
   // error handler for Exception1
   ......
} finally {   // finally is optional
   // clean up codes, always executed regardless of exceptions
   ......
}
```

Try-catch-finally

Esta es la salida cuándo ocurre una excepción

```
Start of the main logic
Try opening a file ...
File Not Found caught ...
finally-block runs regardless of the state of exception
After try-catch-finally, life goes on...
```

Esta es la salida cuándo no ocurre una excepción

```
Start of the main logic
Try opening a file ...
File Found, processing the file ...
End of the main logic
finally-block runs regardless of the state of exception
After try-catch-finally, life goes on...
```

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class TryCatchFinally {
   public static void main(String[] args) {
      try {         // main logic
         System.out.println("Start of the main logic");
         System.out.println("Try opening a file ...");
         Scanner in = new Scanner(new File("test.in"));
         System.out.println("File Found, processing the file ...");
         System.out.println("End of the main logic");
      } catch (FileNotFoundException ex) {    // error handling separated from the main logic
         System.out.println("File Not Found caught ...");
      } finally {   // always run regardless of exception status
         System.out.println("finally-block runs regardless of the state of exception");
      }
      // after the try-catch-finally
      System.out.println("After try-catch-finally, life goes on...");
   }
}
```

Actividad. Try-catch-finally

4. Copia y ejecuta el siguiente código- Primero para que encuentre el archivo y luego para que no lo encuentre.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class TryCatchFinally {
   public static void main(String[] args) {
      try {         // main logic
         System.out.println("Start of the main logic");
         System.out.println("Try opening a file ...");
         Scanner in = new Scanner(new File("test.in"));
         System.out.println("File Found, processing the file ...");
         System.out.println("End of the main logic");
      } catch (FileNotFoundException ex) {    // error handling separated from the main logic
         System.out.println("File Not Found caught ...");
      } finally {   // always run regardless of exception status
         System.out.println("finally-block runs regardless of the state of exception");
      }
      // after the try-catch-finally
      System.out.println("After try-catch-finally, life goes on...");
   }
}
```

Actividad. Throws

PruebaFactorial

- El programa **PruebaFactorial** contiene código que llama al método factorial de la clase MathUtils para calcular los factoriales de enteros introducidos por el usuario.
- Descarga estos archivos en tu directorio y estudia el código en ambos, luego compila y ejecuta PruebaFactorial para ver cómo funciona.

```
public class MathUtils{

    public static int factorial(int n) {
        int fac = 1;
        for (int i=n; i>0; i--)
            fac *= i;
        return fac;
    }
}
```

```
public class PruebaFactorial {

    public static void main(String[] args){
        String keepGoing = "y";
        Scanner scan = new Scanner(System.in);
        while (keepGoing.equals("y") || keepGoing.equals("Y")) {
            System.out.print("Enter an integer: ");
            int val = scan.nextInt();
            System.out.println("Factorial(" + val + ") = " + MathUtils.factorial(val));
            System.out.print("Another factorial? (y/n) ");
            keepGoing = scan.next();
        }
    }
}
```

Actividad. Throws

PruebaFactorial

- Prueba con varios números enteros positivos, luego prueba con un número negativo. Encontrarás que funciona para enteros positivos pequeños (valores < 17), pero que devuelve un valor negativo grande para enteros más grandes y que siempre devuelve 1 para enteros negativos.
- Devolver 1 como factorial de cualquier entero negativo no es correcto-matemáticamente, la función factorial no está definida para enteros negativos. Para corregir esto, podría modificar su método factorial para comprobar si el argumento es negativo, pero ¿entonces qué? El método debe devolver un valor, cualquiera que sea el valor devuelto podría ser mal interpretado.

Actividad. Throws

PruebaFactorial

- En su lugar, deberías lanzar una excepción indicando que algo ha ido mal y que no ha podido completar su cálculo. Podrías definir tu propia clase de excepción, pero ya existe una excepción apropiada para esta situación-IllegalArgumentException, que extiende RuntimeException.
- Modifica la cabecera del método factorial para indicar que factorial puede lanzar una IllegalArgumentException. Modifica el cuerpo de factorial para comprobar el valor del argumento y, si es negativo, lanzar unaIllegalArgumentException.
- Ten en cuenta que lo que pasas a lanzar es en realidad una instancia del método IllegalArgumentException, y que el constructor toma un parámetro String. Utilice este parámetro para ser específico sobre cuál es el problema.

Actividad. Throws

PruebaFactorial

- Compila y ejecuta tu clase PruebaFactorial después de hacer estos cambios. Ahora, cuando introduzcas un número negativo se lanzará una excepción, terminando el programa. El programa termina porque la excepción no es capturada, por lo que es lanzada por el método principal, causando un error de ejecución.
- Modifica el método main de su clase PruebaFactorial para capturar la excepción lanzada por factorial e imprimir un mensaje apropiado, pero luego continua con el bucle. Piensa cuidadosamente dónde tendrás que poner el try y el catch.

Actividad. Throws

PruebaFactorial

- Devolver un número negativo para valores superiores a 16 tampoco es correcto. El problema es el desbordamiento aritmético. El factorial es más grande de lo que puede ser representado por un int. Esto también puede ser considerado como una IllegalArgumentException-este método factorial solo está definido para argumentos de hasta 16.
- Modifique tu código en factorial para comprobar si hay un argumento mayor de 16 así como un argumento negativo. Deberías lanzar una IllegalArgumentException en cualquier caso, pero pasar diferentes mensajes al constructor para que el problema claro.

IllegalArgumentException

¿Cuándo y por qué suele ocurrir IllegalArgumentException en Java?

IllegalArgumentException es una excepción de Java que indica que un método ha recibido un parámetro que no es válido o es inadecuado para los fines de este método. IllegalArgumentException se usa comúnmente en escenarios donde el tipo de parámetro de un método no es suficiente para restringir adecuadamente sus valores posibles.

IllegalArgumentException

¿Cómo atrapar IllegalArgumentException en Java?

Dado que IllegalArgumentExceptiones una excepción **no verificada**, no tienes que manejarla en tu código: Java le permitirá compilar sin problemas.

En muchos casos, en lugar de intentar capturar IllegalArgumentException, simplemente puedes comprobar que un valor se encuentre en el rango esperado antes de pasarlo a un método.

IllegalArgumentException

Cuándo y cómo lanzar IllegalArgumentException en Java

Normalmente arroja un error IllegalArgumentException al validar los parámetros de entrada pasados a un método Java y debe ser más estricto de lo que permite el sistema de tipos.

Por ejemplo, si tu método acepta un parámetro entero que utiliza para expresar un porcentaje, probablemente necesites asegurarse de que, para que tenga sentido, el valor de ese parámetro esté entre 0 y 100. Si el valor se sale de ese rango , puedes lanzar un IllegalArgumentException:

```
public static int getAbsoluteEstimateFromPercentage(double percentOfTotal) {

    int totalPopulation = 143_680_117;

    if (percentOfTotal < 0 || percentOfTotal > 100) {
        throw new IllegalArgumentException("Percentage of total should be between 0 and 100, " +
                                           "but was %f".formatted(percentOfTotal));
    }

    return (int) Math.round(totalPopulation * (percentOfTotal * 0.01));
}
```

IllegalArgumentException

Cuando lanzas una IllegalArgumentExceptionen tu método, no tienes que agregarlo a la cláusula **throws** del método porque no está verificada. Sin embargo, muchos desarrolladores tienden a agregar excepciones **no verificadas** a la cláusula throws de todos modos con fines de documentación:

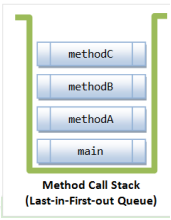
```
public static int getAbsoluteEstimateFromPercentage(double percentOfTotal) throws IllegalArgumentException
```

Incluso si agregas throws IllegalArgumentExceptiona, las personas que llamen a tu método no estarán obligadas a manejarla.

Una forma alternativa de documentar excepciones importantes **no verificadas** es usar la etiqueta @throws en la documentación de Javadoc .

```
/**
* @param percentOfTotal Percentage of total
* @return A rough estimate of the absolute number resulting from taking a percentage of total
* @throws IllegalArgumentException if percentage is outside the range of 0..100
*/
public static int getAbsoluteEstimateFromPercentage(double percentOfTotal) {
```

Pila de llamadas a métodos



Una aplicación típica involucra muchos niveles de llamadas a método que es administrada por una llamada **pila de llamadas a métodos.**

Una pila es una cola de último en entrar, primero en salir.

En el siguiente ejemplo,

- el método main () invoca a methodA();
- methodA() llama a methodB();
- methodB() llama a methodC().

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit methodC()
Exit methodB()
Exit methodA()
Exit main()
```

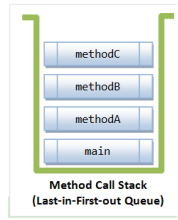
```
public class MethodCallStackDemo {
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit main()");
   }

   public static void methodA() {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit methodA()");
   }

   public static void methodB() {
      System.out.println("Enter methodB()");
      methodC();
      System.out.println("Exit methodB()");
   }

   public static void methodC() {
      System.out.println("Enter methodC()");
      System.out.println("Exit methodC()");
   }
}
```

Pila de llamadas a métodos



Como se ve en la salida, la secuencia de eventos es:

1. JVM invoca el main ().
2. main () insertado en la pila de llamadas, antes de invocar a methodA ().
3. methodA () es insertado en la pila de llamadas, antes de invocar methodB ().
4. methodB () es insertado en la pila de llamadas, antes de invocar methodC ().
5. methodC () se completa.
6. methodB () es quitado de la pila de llamadas y se completa.
7. methodA () es quitado de la pila de llamadas y se completa.
8. main () es quitado de la pila de llamadas y se completa. Programa termina

```
public class MethodCallStackDemo {
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit main()");
   }

   public static void methodA() {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit methodA()");
   }

   public static void methodB() {
      System.out.println("Enter methodB()");
      methodC();
      System.out.println("Exit methodB()");
   }

   public static void methodC() {
      System.out.println("Enter methodC()");
      System.out.println("Exit methodC()");
   }
}
```

Pila de llamadas a métodos

Supongamos que modificamos methodC () para llevar a cabo una operación "dividir por 0", que provoca una excepción ArithmeticException.

```
public static void methodC() {
   System.out.println("Enter methodC()");
   System.out.println(1 / 0);  // divide-by-0 triggers an ArithmeticException
   System.out.println("Exit methodC()");
}
```

El mensaje de excepción muestra claramente el método de seguimiento de la pila de llamadas con los números de línea de sentencias relevantes:

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
        at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

Pila de llamadas a métodos

El methodC() provoca una excepción ArithmeticException. Como no maneja esta excepción, sale de la pila de llamadas inmediatamente. methodB() tampoco maneja esta excepción y sale de la pila de llamadas. Lo mismo ocurre con los métodos methodA() y main(). El método main() pasa de nuevo a JVM, que termina abruptamente el programa e imprime el seguimiento de la pila de llamadas, como se muestra.

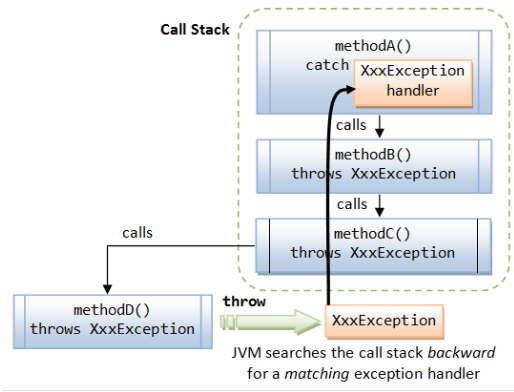
```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
        at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

Excepciones y pila de llamadas

- Cuando se produce una excepción dentro de un método Java, el método crea un objeto Exception y pasa el objeto Exception a la JVM (en el término Java, el método "lanza“ (throw) una Excepción).
- El objeto Exception contiene el tipo de excepción y el estado del programa cuando se produce la excepción. La JVM es responsable de encontrar un controlador de excepciones para procesar el objeto Exception.
- Busca hacia atrás a través de la pila de llamadas hasta que encuentra un controlador de excepciones coincidente para esa clase particular de objeto de Exception (en el término Java, se llama "captura" –catch- de la excepción). Si la JVM no puede encontrar un controlador de excepciones coincidente en todos los métodos de la pila de llamadas, finaliza el programa.

Excepciones y pila de llamadas

- Este proceso se ilustra a continuación. Supongamos que methodD () encuentra una condición anormal y lanza una XxxException a la JVM.
- La JVM busca hacia atrás a través de la pila de llamadas para un controlador de excepciones que coincida.
- Encuentra methodA () que tiene un controlador XxxException y pasa el objeto de excepción al controlador.
- Observa que methodC () y methodB () están obligados a declarar "xxxException" en sus cabeceras de método para compilar el programa.



Actividad. Pila de llamadas

5. Copia el siguiente código y ejecútalo

```
public class MethodCallStackDemo {
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit main()");
   }

   public static void methodA() {
      System.out.println("Enter methodA()");
      try {
         System.out.println(1 / 0);
            // A divide-by-0 triggers an ArithmeticException - an unchecked exception
            // This method does not catch ArithmeticException
            // It runs the "finally" and popped off the call stack
      } finally {
         System.out.println("finally in methodA()");
      }
      System.out.println("Exit methodA()");
   }
}
```

```
Enter main()
Enter methodA()
finally in methodA()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:11)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

Crear nuestras propias excepciones

Deberías intentar reutilizar las clases de excepción proporcionadas en el JDK, por ejemplo, IndexOutOfBoundException, ArithmeticException, IOException y IllegalArgumentException.

Pero siempre puedes **crear sus propias clases de Excepción** extendiéndose desde la clase Exception o una de sus subclases.

```
// Create our own exception class by subclassing Exception. This is a checked exception
public class MyMagicException extends Exception {
   public MyMagicException(String message) {  //constructor
      super(message);
   }
}
```

```
public class MyMagicExceptionTest {
   // This method "throw MyMagicException" in its body.
   // MyMagicException is checked and need to be declared in the method's signature
   public static void magic(int number) throws MyMagicException {
      if (number == 8) {
         throw (new MyMagicException("you hit the magic number"));
      }
      System.out.println("hello");  // skip if exception triggered
   }

   public static void main(String[] args) {
      try {
         magic(9);   // does not trigger exception
         magic(8);   // trigger exception
      } catch (MyMagicException ex) {   // exception handler
         ex.printStackTrace();
      }
   }
}
```

```
hello
MyMagicException: you hit the magic number
        at MyMagicExceptionTest.magic(MyMagicExceptionTest.java:6)
        at MyMagicExceptionTest.main(MyMagicExceptionTest.java:14)
```

Una excepción es un objeto que describe una situación errónea.

Resumen

Un programa puede gestionar una excepción de una de estas tres maneras:

- ignorarla
- tratarla en el momento en que se produce
- tratarla en otro lugar del programa

La forma en que se procesa una excepción es una consideración de diseño importante

Resumen

Una excepción puede ser **verificada** o **no verificada**.

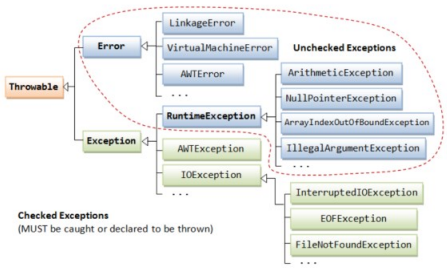
Una excepción **verificada**:

- Debe ser capturada por un método o debe aparecer en la cláusula throws de cualquier método que pueda lanzarla o propagarla
- Se añade una cláusula throws a la cabecera del método
- El compilador emitirá un error si una excepción verificada no es capturada o utilizada en una cláusula throws.

Resumen

Una excepción **no verificada** no requiere un tratamiento explícito, aunque podría procesarse de ese modo

- Las únicas excepciones no verificadas en Java son objetos de tipo RuntimeException o cualquiera de sus descendientes
- Los Error son similares a las RuntimeException y sus descendientes:
  - Los errores no deben capturarse
  - Los errores no requieren una cláusula throws



Recomendaciones de uso

Para convertirnos en ”***maestros”*** de las excepciones, debemos evitar el uso de aquellas malas practicas que se han generalizado a los largo de los años (y por supuesto no inventar las nuestras...). La primera que vamos a ver es la más peligrosa y, a pesar de ello, también la más común:

```
try {
    // Código que declara lanzar excepciónes
} catch(Exception ex) {}
```

El código anterior ignorará cualquier excepción que se lance dentro del bloque try, o mejor dicho, capturará toda excepción lanzada dentro del bloque try pero la silenciará no haciendo nada (frustrando así el principal propósito de la gestión de excepciones verificadas: gestiónala o relánzala). Cualquier error de diseño, de programación o de funcionamiento en esas líneas de código pasará inadvertido tanto para el programador como para el usuario.

Recomendaciones de uso

Lo mínimamente aceptable dentro de un bloque catch es un mensaje de log informando del error:

```
try {
    // Código que declara lanzar excepciónes
} catch(Exception ex) {
    logging.log("Se ha producido el siguiente error: " + ex.getMessage());
    logging.log("Se continua la ejecución");
}
```

Recomendaciones de uso

Programming Tip 11.1

Throw Early, Catch Late

When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix. For example, suppose a method expects to read a number from a file, and the file doesn't contain a number. Simply using a zero value would be a poor choice because it hides the actual problem and perhaps causes a different problem elsewhere.

Throw an exception as soon as a problem is detected. Catch it only when the problem can be handled.

Conversely, a method should only catch an exception if it can really remedy the situation. Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler.

These principles can be summarized with the slogan "throw early, catch late".

```
//La pendiente de una recta vertical (dos puntos con la misma x) no puede ser definida
//por lo que no queda otra que lanzar una excepción
public double pendiente(Punto otroPunto) {
    DecimalFormat decimalFormat = new DecimalFormat("#.####");
    double pendiente = 0;
    if (esVertical(otroPunto))
        throw new IllegalStateException("/ by zero");
    else
        pendiente = Double.parseDouble(decimalFormat.format((y - otroPunto.y) / (x - otroPunto.x)));
    return pendiente;
}
```

You should only catch those exceptions that you can handle.

Programming Tip 11.2

Do Not Squelch Exceptions

When you call a method that throws a checked exception and you haven't specified a handler, the compiler complains. In your eagerness to continue your work, it is an understandable impulse to shut the compiler up by squelching the exception:

```
try
{
   Scanner in = new Scanner(new File(filename));
   // Compiler complained about FileNotFoundException
   . . .
}
catch (FileNotFoundException e) {} // So there!
```

The do-nothing exception handler fools the compiler into thinking that the exception has been handled. In the long run, this is clearly a bad idea. Exceptions were designed to transmit problem reports to a competent handler. Installing an incompetent handler simply hides an error condition that could be serious.

Programming Tip 11.3

Do Throw Specific Exceptions

When throwing an exception, you should choose an exception class that describes the situation as closely as possible. For example, it would be a bad idea to simply throw a `RuntimeException` object when a bank account has insufficient funds. This would make it far too difficult to catch the exception. After all, if you caught all exceptions of type `RuntimeException`, your `catch` clause would also be activated by exceptions of the type `NullPointerException`, `ArrayIndexOutOfBoundsException`, and so on. You would then need to carefully examine the exception object and attempt to deduce whether the exception was caused by insufficient funds.

If the standard library does not have an exception class that describes your particular error situation, simply provide a new exception class.

Recomendaciones de uso

Un buen uso del tratamiento de excepciones es usar **excepciones que ya existen,** en lugar de crear las tuyas propias, siempre que ambas fueran a cumplir el mismo cometido (que es básicamente informar y, en caso de las checked, obligar a gestionar).

Se suelen usar excepciones que ya existen cuando se dispone de un profundo conocimiento del API que se está usando (en otras palabras, experiencia). Si un argumento pasado a uno de tus métodos no es del tipo esperado, o no tiene el formato correcto, lanza una excepción IllegalArgumentException en lugar de crear tu propia excepción.

Esto es bueno porque:

- Uno de los pilares de Java es la reutilización de código (no reinventes la rueda)
- Tu código es más universal (FormatoInvalidoException puede no significar nada para un germanoparlante)

Recomendaciones de uso

Utilizar Excepciones específicas. Cuanto más específica es la excepción que arrojas, es mejor.

Asegúrate de proporcionar la mayor cantidad de información posible. Eso hace que su API sea más fácil de entender. Y como resultado, la persona que llama a su método podrá manejar la excepción mejor o evitarla con un control adicional.

Recomendaciones de uso

Por lo tanto, siempre intente encontrar la clase que mejor se adapte a tu excepción, por ejemplo, lanzar un NumberFormatException en lugar de un IllegalArgumentException. Y evita lanzar una excepción no especificada.

```
public void doNotDoThis() throws Exception { ... }

public void doThis() throws NumberFormatException { ... }
```

Recomendaciones de uso

Debes **documentar** adecuadamente las excepciones que lanza tu código. Para ello, detalla en tus Javadoc todas las excepciones que lanzan tus métodos, informando que condiciones van a provocar el lanzamiento de cada una de ellas.

```
*
* @param input
* @throws MyBusinessException if ... happens
*/
public void doSomething(String input) throws MyBusinessException { ... }
```

Captura la **excepción más específica primero**. Siempre captura primero la clase de excepción más específica y agrega los bloques de captura menos específicos al final de su lista.

```
public void catchMostSpecificExceptionFirst() {
    try {
        doSomething("A message");
    } catch (NumberFormatException e) {
        log.error(e);
    } catch (IllegalArgumentException e) {
        log.error(e)
    }
}
```

Recomendaciones de uso

No captures Throwable. Throwable es la super clase de todas las excepciones y errores. Se puede utilizar en una cláusula catch, pero nunca debes hacerlo. Si usa Throwable en una cláusula cath, no solo capturará todas las excepciones; también detectará todos los errores.

La JVM produce errores para indicar problemas graves que no están destinados a ser manejados por una aplicación. Ejemplos típicos para eso son OutOfMemoryError o StackOverFlowError. Ambos son causados por situaciones que están fuera de control de la aplicación y no se pueden manejar. Por lo tanto, no atrape un Trowable.

```
public void doNotCatchThrowable() {
    try {
        // realice algo
    } catch (Throwable t) {
        // No haga esto.
    }
}
```

Recomendaciones de uso

No trates una excepción para luego lanzarla. Esta es probablemente la recomendación más ignorada en esta lista. Puedes encontrar código e incluso librerías en donde una excepción es capturada, luego lanzada.

```
try {
   new Long("xyz");
} catch (NumberFormatException e) {
   log.error(e);
   throw e;
}
```

Puede ser intuitivo tratar una excepción cuando ocurrió y luego volver a lanzarla para que la persona que la llame pueda manejarla adecuadamente. Pero escribirá múltiples mensajes de error para la misma excepción.

Recomendaciones de uso

```
17:44:28,945 ERROR TestExceptionHandling:65 - java.lang.NumberFormatException: For input string: "xyz
Exception in thread "main" java.lang.NumberFormatException: For input string: "xyz"
      at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
      at java.lang.Long.parseLong(Long.java:589)
      at java.lang.Long.(Long.java:965)
      at com.stackify.example.TestExceptionHandling.logAndThrowException(TestExceptionHandling.java:6
      at com.stackify.example.TestExceptionHandling.main(TestExceptionHandling.java:58)
```

Los mensajes adicionales tampoco agregan ninguna información. El mensaje de excepción debe describir el evento excepcional. Y el seguimiento de la pila le dice en que clase, método y línea se lanzo la excepción.

Recomendaciones de uso

Envuelve la excepción sin consumirla. A veces es mejor **capturar una excepción estándar y envolverla en una personalizada.** Un ejemplo típico para una excepción de este tipo es una excepción comercial específica de la aplicación o del framework. Eso le permite agregar información adicional y también puede implementar un manejo especial para su clase de excepción.

```
public void wrapException(String input) throws MyBusinessException {
   try {
      // do something
   } catch (NumberFormatException e) {
       throw new MyBusinessException("A message that describes the error.", e);
   }
}
```

Recomendaciones de uso


```
public void wrapException(String input) throws MyBusinessException {
   try {
      // do something
   } catch (NumberFormatException e) {
       throw new MyBusinessException("A message that describes the error.", e);
   }
}
```


Cuando lo hagas, asegúrate de establecer la excepción original como la causa. La clase Excepcion proporciona métodos de constructor específicos que aceptan un Throwable como parámetro. De lo contrario, perderá el rastro de la pila y el mensaje de la excepción original que dificultará el análisis del evento excepcional que causó la excepción.

Recursos

- Curso youtube : Java desde 0
- Libro Java 9. Manual imprescindible. F. Javier Moldes Teo. Editorial Anaya
- App SoloLearn: Aprende a Programar. Curso Java