Programación en Java

1. Introducción al lenguaje Java
2. Los operadores
3. Las estructuras de control
4. Las estructuras de bucle
5. Los arrays
6. Las cadenas de caracteres
7. Orientación a Objetos
8. Enumeraciones
9. Fecha y hora
10. **Excepciones y aserciones**
11. Las colecciones
12. Genéricos
13. Interfaces funciones y expresiones lambda
14. Streams
15. Concurrencia
16. Ficheros
17. Bases de Datos
18. Localización

Enlaces

- Descargas
- Software necesario
- Referencias

ajustes:excepciones

Excepciones y aserciones

Excepciones

Las excepciones en Java son errores que se producen en tiempo de ejecución. Cuando realizamos el proceso de compilación pueden aparecer una serie de errores que deben ser corregidos para que el programa pueda compilarse y generarse. Más tarde, durante la ejecución del mismo, puedne producirse una serie de errores, los cuales son imposibles de detectar durante la fase de compilación. Son este tipo de errores los que, en Java, acaban lanzando una excepción si llegan a producirse:

- Estamos escribiendo en disco y éste se llena (\verb IOException)
- Estamos descargando un fichero y cae la conexión a Internet (\verb ConnectionException)
- Estamos recorriendo un array en un bucle y accedemos a una posición que no existe (\verb IndexOutOfBoundException)
- Accedemos atributos o métodos de un objeto cuyo valor es nulo (\verb NullPointerException)
- Realizamos una operación matemática no válida (división por cero) (\verb ArithmeticException)

En Java se propone el control de excepciones para evitar que se produzcan estos errores en lugar de tener que escribir continuamente estructuras \verb if que controlen que no ocurre nada anómalo que implique la ejecución de un cierto código. Así, lo que haremos será colocar dentro un bloque controlado todo el código (y el que dependa de éste) que sea susceptible de producir una excepción, sin interrumpir el flujo de nuestro programa (al contrario de lo que ocurre añadiendo sentencias \verb if).

Se recomienda excepciones como \verb NullPointerException y \verb IndexOfOutBoundException no se traten como tal ya que, por lo general, corresponderán a errores de codificación, por lo que realmente, se debería evitar que se produjeran corrigiendo el programa y no esperando capturarlas cuando se produzcan. Son dos excepciones que derivan de \verb RuntimeException . Son lo que se conoce como \emph{unchecked exceptions}. Echa un vistazo a \verb|{{https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html}}| para más información al respecto.

\subsection{Bloque try-catch}

El bloque \verb try-catch es el bloque Java que permite delimitar el código susceptible de generar excepción (parte \verb try) y ejecutar otro bloque para corregir o notificar el problema (parte \verb catch).

```
. . .
try {
    // código
} catch (Exception e1) {
} catch (Exception e2) {
} . . .

. . .
```

En este caso, el código que hay asociado al bloque \verb try se ejecutará de forma que, si dentro de él se produjera una excepción, el flujo de ejecución pasaría al bloque \verb catch de la excepción que corresponda. A continuación, seguirá ejecutando justo al final de todo el grupo \verb try-catch .

```
. . .
File fichero = null;
FileReader lectorFichero = null;
BufferedReader buffer = null;
try {
    buffer = new BufferedReader(new FileReader(new File("archivo.txt")));
    String linea = null;
    while ((linea = buffer.readLine()) != null) {
        System.out.println(linea);
    }
    buffer.close();
} catch (FileNotFoundException fnfe) {
    System.out.println("El fichero especificado no existe");
    fnfe.printStackTrace();
} catch (IOException ioe) {
    System.out.println("Se ha producido un error al leer de disco");
    ioe.printStackTrace();
}
. . .
```

Instrucción throws

También podemos, en lugar de capturar la excepción, lanzarla haciendo que lenga que ser controlada en un nivel superior. Por ejemplo, si estoy desarrollando una clase que debe controlar una determinada excepción, quizás me pueda resultar interesante hacerla "subir" de nivel para que sea controlada, por ejemplo, en el controlador de la aplicación o bien cerca de la interfaz con el usuario. Así es probable que podamos ajustar mejor el comportamiento en caso de error.

```
public class Util {
    public static Date parseFecha(String fecha) throws ParseException {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        return sdf.parse(fecha);
    }
}
```

El ejemplo de arriba es un caso en el que tenemos un método estático que utilizamos para parsear fechas en nuestra aplicación. Cuando el texto que se le pase no se corresponda con un formato válido de fecha, lanzará una excepción. Pero justamente en el método no tenemos muy claro que hacer puesto que dicho método lo utilizarán desde diferentes localizaciones del código en la aplicación. Por eso es mejor que lancemos fuera la excepción para que sea quién utilice este método quién deba decidir qué hacer en caso de fallo. Por ejemplo

```
public class Ventana {
    . . .
    private JTextField tfFecha;
    . . .
    private void comprobarInput() {
        . . .

        try {
            . . .
            Date fecha = Util.parseFecha(tfFecha.getText());
            . . .
        } catch (ParseException pe) {
            JOptionPane.showMessageDialog(. . .);
        }
    }
}
```

Bloque try-catch-finally

Se puede añadir también un tercer bloque de código asociado a la parte \verb try que permite añadir código que se ejecutará siempre, de forma que incluso lo hará cuando se produzca una excepción no controlada. Normalmente se utiliza para la zona donde liberar recursos que hayan sido ocupados durante el bloque \verb try . De esta forma nos aseguramos que siempre se liberen correctamente, incluso aunque el flujo de ejecución del código se vea interrumpido por una excepción.

```
. . .
BufferedReader buffer = null;
try {
    buffer = new BufferedReader(new FileReader(new File("archivo.txt")));
    String linea = null;
    while ((linea = buffer.readLine()) != null) {
        System.out.println(linea);
    }
} catch (FileNotFoundException fnfe) {
    System.out.println("El fichero especificado no existe");
    fnfe.printStackTrace();
} catch (IOException ioe) {
    System.out.println("Se ha producido un error al leer de disco");
    ioe.printStackTrace();
} finally {
    if (buffer != null) {
        try {
            buffer.close();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
. . .
```

Cómo crear tus propias excepciones

Como ya hemos visto, Java tiene una serie de \emph{unchecked} y \emph{checked exceptions} que son lanzadas en determinadas situaciones. Vamos a ver ahora cómo también podemos crear nuestras propias excepciones y lanzarlas cuando se produzcan situaciones de "más alto nivel" en nuestra lógica de la aplicación.

Por ejemplo, en una aplicación para la gestión de inventarios, almacenes o pedidos, podemos definir una excepción como la del siguiente ejemplo, para que sea lanzada en el caso de que se termine el stock de algún producto de nuestro almacén:

```
public class FueraDeStockException extends Exception {

    public FueraDeStockException() {
        super();
    }

    public FueraDeStockException(String mensaje) {
        super(mensaje);
    }
}
```

Así, cuando nos encontramos con esa situación, podemos directamente lanzar nuestra excepción:

```
. . .
private void anadirProductoAlCarrito(Producto producto) {
    if (producto.getStock() <= 0) {
        throw new FueraDeStockException();
    }
}
. . .
```

Y de esta manera, por ejemplo, en el controlador o interaz de la aplicación, podríamos hacer algo así:

```
public class Ventana {
    . . .
    try {
        . . .
        anadirProductoAlCarrito(producto);
        . . .
    } catch (FueraDeStockException fdse) {
        JOptionPane.showMessageDialog(. . .);
        fdse.printStackTrace();
    }
    . . .
}
```

Bloques multi-catch

También es posible capturar más de un tipo de excepción bajo un mismo bloque \verb catch , aunque no es una práctica recomendada ya que no permite afinar el caso de error en algunos casos, pero bien utilizado permite que agrupemos en el caso de que varias excepciones tengan asociado el mismo bloque de código.

```
public class Ventana {
    . . .
    try {
        . . .
        anadirProductoAlCarrito(producto);
        . . .
    } catch (FueraDeStockException | ProductoDescatalogadoException e) {
        JOptionPane.showMessageDialog(. . .);
        fdse.printStackTrace();
    }
    . . .
}
```

Uso de recursos en bloques try-catch

Como hemos visto en el primer ejemplo, accedíamos a un fichero en la excepción, para luego acabar liberando los recursos al finalizar el bloque \verb try-catch . A veces es habitual que dichos recursos se inicialicen al comenzar la excepción (incluso dentro de éste) y se liberen al terminar. Así, ese ejemplo podría escribirse de la siguiente forma:

```
. . .
try (BufferedReader buffer =
        new BufferedReader(new FileReader(new File("archivo.txt")))) {
    String linea = null;
    while ((linea = buffer.readLine()) != null) {
        System.out.println(linea);

} catch (FileNotFoundException fnfe) {
    System.out.println("El fichero especificado no existe");
    fnfe.printStackTrace();
} catch (IOException ioe) {
    System.out.println("Se ha producido un error al leer de disco");
    ioe.printStackTrace();
}
. . .
```

Para este caso, hay que tener en cuenta que, para que Java pueda liberar automáticamente los recursos de una clase Java, debemos implementar uno de los interfaces \verb java.lang.AutoCloseable o \verb java.io.Closeable , y declarar un método \verb close()

```
public class MiRecurso implements AutoCloseable {
    . . .
    . . .
    public void close() throws Exception {
        // Aquí hay que liberar los recursos
        // Se ejecutará automáticamente
        . . .
        . . .
    }
}
. . .
```

Aserciones

Las aserciones son sentencias utilizadas para comprobar si una condición es cierta (o no) y controlar así los errores en el código. Básicamente equivale a una sentencia \verb if que evalúa una condición y genera una Excepción \verb AssertException si ésta se cumple, mostrando el mensaje asociado a la misma. De forma similar a cómo hacen las excepciones, permite hacerlo de forma cómoda y sin romper el flujo del código. Además, estas condiciones en forma de aserción sólo tienen efecto si se pasa la opción \verb -ea o \verb -enableassertions a la máquina virtual cuando se ejecuta la aplicación. De esta forma sólo se activan cuando es necesario o interesa utilizarlas.

Veamos un ejemplo:

```
. . .
String cadena = "Esto es un texto";
. . .
. . .
assertion cadena == null : "La cadena es nula";
. . .
// También se puede especificar una condición sin mensaje
assertion cadena == null;
. . .
```

Si ejecutamos el código anterior con la opción \verb -ea activada como opción de la JVM, se nos mostrará el siguiente mensaje de error:

```
run:
Exception in thread "main" java.lang.AssertionError: La cadena es nula
        at com.xfaci.prueba.JavaApplication1.main(JavaApplication1.java:29)
/Users/Santi/Library/Caches/NetBeans/8.2/executor-snippets/run.xml:111: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

© 2019-2021 Santiago Faci

Except where otherwise noted, content on this wiki is licensed under the following license: Public Domain

Print · Slideshow · Chapter · Back · Up · Edit · Discuss