

NOTAS – REPASO – EDITS:   
[UT1, UT3 Generación de números aleatorios] status: updated  
Falta: [UT4 Programación Orientada a Objetos, final]

# UT8 Gestión de Excepciones

## Excepciones. Categorías.

Las excepciones son el mecanismo por el cual pueden controlarse en un programa Java las condiciones de error que se producen. Estas condiciones de error pueden ser errores en la lógica del programa como un índice de un array fuera de su rango, una división por cero o errores disparados por los propios objetos que denuncian algún tipo de estado no previsto, o condición que no pueden manejar.

La idea general es que cuando un objeto encuentra una condición que no sabe manejar crea y dispara una excepción que deberá ser capturada por el que le llamó o por alguien más arriba en la pila de llamadas. Las excepciones son objetos que contienen información del error que se ha producido y que heredan de la clase Throwable o de la clase Exception. Si nadie captura la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar quién produjo la excepción.

Existen dos categorías de excepciones:

* Excepciones verificadas: El compilador obliga a verificarlas. Son todas las que son lanzadas explícitamente por objetos de usuario.
* Excepciones no verificadas: El compilador no obliga a su verificación. Son excepciones como divisiones por cero, excepciones de puntero nulo, o índices fuera de rango.

## Generación de excepciones

Supongamos que tenemos una clase Empresa que tiene un array de objetos Empleado (clase vista en capítulos anteriores). En esta clase podríamos tener métodos para contratar un Empleado (añadir un nuevo objeto al array), despedirlo (quilarlo del array) u obtener el nombre a partir del número de empleado. La clase podría ser algo así como lo siguiente:

**public class** Empresa {  
    String nombre;  
    Empleado [] listaEmpleados;  
    **int** totalEmpleados = 0;  
    . . .  
    Empresa(String n, **int** maxEmp) {  
        nombre = n;  
        listaEmpleados = **new** Empleado [maxEmp];  
    }  
    . . .  
    **void** nuevoEmpleado(String nombre, **int** sueldo) {  
       if (totalEmpleados < listaEmpleados.length ) {  
            listaEmpleados[totalEmpleados++] = **new** Empleado(nombre,sueldo);  
       }  
    }  
}

Obsérvese en el método nuevoEmpleado que se comprueba que hay sitio en el array para almacenar la referencia al nuevo empleado. Si lo hay se crea el objeto. Pero si no lo hay el método no hace nada más. No da ninguna indicación de si la operación ha tenido éxito o no. Se podría hacer una modificación para que, por ejemplo el método devolviera un valor booleano true si la operación se ha completado con éxito y false si ha habido algún problema.

Otra posibilidad es generar una excepción verificada (Una excepción no verificada se produciría si no se comprobara si el nuevo empleado va a caber o no en el array). Vamos a ver como se haría esto.

Las excepciones son clases, que heredan de la clase genérica Exception. Es necesario por tanto asignar un nombre a nuestra excepción. Se suelen asignar nombres que den alguna idea del tipo de error que controlan. En nuestro ejemplo le vamos a llamar CapacidadEmpresaExcedida.

Para que un método lance una excepción:

* Debe declarar el tipo de excepción que lanza con la cláusula throws, en su declaración.
* Debe lanzar la excepción, en el punto del código adecuado con la sentencia throw.

En nuestro ejemplo:

**void** nuevoEmpleado(String nombre, **int** sueldo) **throws** CapacidadEmpresaExcedida {  
       if (totalEmpleados < listaEmpleados.length) {  
            listaEmpleados[totalEmpleados++] = **new** Empleado(nombre,sueldo);  
       }  
       else **throw** **new** CapacidadEmpresaExcedida(nombre);  
    }

Además, necesitamos escribir la clase CapacidadEmpresaExcedida. Sería algo así:

**public** **class** CapacidadEmpresaExcedida **extends** Exception {  
     CapacidadEmpresaExcedida(String nombre) {  
        **super**("No es posible añadir el empleado " + nombre);  
    }  
    . . .  
}

La sentencia throw crea un objeto de la clase CapacidadEmpresaExcedida. El constructor tiene un argumento (el nombre del empleado). El constructor simplemente llama al constructor de la superclase pasándole como argumento un texto explicativo del error (y el nombre del empleado que no se ha podido añadir).

La clase de la excepción puede declarar otros métodos o guardar datos de depuración que se consideren oportunos. El único requisito es que extienda la clase Exception. Consultar la documentación del API para ver una descripción completa de la clase Exception.

*<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Throwable.html>*

De esta forma se pueden construir métodos que generen excepciones.

## Captura de excepciones

Con la primera versión del método nuevoEmpleado (sin excepción) se invocaría este método de la siguiente forma:

Empresa em = **new** Empresa("La Mundial");  
em.nuevoEmpleado("Adán Primero",500);

Si se utilizara este formato en el segundo caso (con excepción) el compilador produciría un error indicando que no se ha capturado la excepción verificada lanzada por el método nuevoEmpleado. Para capturar la excepción es utiliza la construcción try / catch, de la siguiente forma:

Empresa em = **new** Empresa("La Mundial");  
**try** {  
    em.nuevoEmpleado("Adán Primero",500);  
} **catch** (CapacidadEmpresaExcedida exc) {  
    System.out.println(exc.toString());  
    System.exit(1);  
}

* Se encierra el código que puede lanzar la excepción en un bloque try / catch.
* A continuación del catch se indica que tipo de excepción se va a capturar.
* Después del catch se escribe el código que se ejecutará si se lanza la excepción.
* Si no se lanza la excepción el bloque catch no se ejecuta.

El formato general del bloque try / catch es:

**try** {   
    . . .  
} **catch** (*Clase\_Excepcion* *nombre*) { . . .}  
  **catch** (*Clase\_Excepcion* *nombre*) { . . .}  
   . . .

Obsérvese que se puede capturar más de un tipo de excepción declarando más de una sentencia catch. También se puede capturar una excepción genérica (clase Exception) que engloba a todas las demás.

En ocasiones el código que llama a un método que dispara una excepción tampoco puede (o sabe) manejar esa excepción. Si no sabe que hacer con ella puede de nuevo lanzarla hacia arriba en la pila de llamada para que la gestione quien le llamo (que a su vez puede capturarla o reenviarla). Cuando un método no tiene intención de capturar la excepción debe declararla mediante la cláusula throws, tal como hemos visto en el método que genera la excepción.

Supongamos que, en nuestro ejemplo es el método main de una clase el que invoca el método nuevoEmpleado. Si no quiere capturar la excepción debe hacer lo siguiente:

**public** **static** **void** main(String [] args) **throws**   
//nunca se pone el throws en el main lolCapacidadEmpresaExcedida {  
    Empresa em = **new** Empresa("La Mundial");  
    em.nuevoEmpleado("Adán Primero",500);  
}

## Cláusula finally

La cláusula finally forma parte del bloque **try** / **catch** y sirve para especificar un bloque de código que se ejecutará tanto si se lanza la excepción como si no. Puede servir para limpieza del estado interno de los objetos afectados o para liberar recursos externos (descriptores de fichero, por ejemplo). La sintaxis global del bloque **try** / **catch** / **finally** es:

**try** {   
    . . .  
} **catch** (*Clase\_Excepcion* *nombre*) { . . .}  
  **catch** (*Clase\_Excepcion* *nombre*) { . . .}  
   . . .   
  **finally** { . . .}

# UT8 Matches

El método matches de String nos permite comprobar si un String cumple una expresión regular pasada como parámetro. Si es cierta devuelve true, sino false.

Una expresión regular es una expresión textual que utiliza símbolos especiales para hacer búsquedas avanzadas.

Las expresiones pueden contener:

* **Caracteres**.
* **Caracteres de control,** por ejemplo, \s, \d, etc. Recuerda que añadir un \ más al introducirlo en una cadena en java. Los más usados son:
  + \d, dígito, es igual que [0-9]
  + \D, no dígito, es igual que [^0-9] *//^ que no esté en este rango*
  + \s, carácter en blanco, es igual que [\t\n\x0B\f\r]
  + \S, no carácter en blanco, es igual que [^\s]
  + \w, carácter alfanumérico, es igual que [a-zA-Z0-9]
  + \W, no carácter alfanumérico, es igual que [^\w], [^a-zA-Z0-9]
* **Opciones de caracteres**, se usa el corchete. Por ejemplo, [afgd] significa que puede contener a, f, g o d.
* **Negación de caracteres**, funciona al revés que el anterior, se usa ^. Por ejemplo, [^afgd]
* **Rangos**, se usa para que incluya un rango de caracteres. Por ejemplo, para que incluya los caracteres entre a y z [a-z]
* **Intersección**: permite unir dos condiciones, es como el operador &&.
* **Cualquier carácter**: se usa un punto.
* **Opcional**: se usa el símbolo ?, indica que un carácter puede o no aparecer.
* **Repetición**: se usa el símbolo \*, indica que un conjunto de caracteres se puede repetir o no.
* **Repetición obligada**: se usa el símbolo +, es como el anterior, pero debe aparecer mínimo una vez.
* **Repetición un número exacto de veces**: después de una expresión abrimos llaves {} con un número dentro, indica el número de veces que debe repetirse un carácter o expresión. Si después del número escribimos una coma, indica que debe repetirse como mínimo el número que indiquemos y como máximo los que queramos. Si después de la coma escribimos un número, indica que debe repetirse entre los números que le indiquemos como si fuera un rango.

Página para probar expresiones regulares:

[regex101: build, test, and debug regex](https://regex101.com/)

Símbolos comunes en expresiones regulares

|  |  |
| --- | --- |
| . | Un punto indica cualquier carácter |
| ^expresión | El símbolo ^ indica el principio del String. En este caso el String debe contener la expresión al principio. |
| expresión$ | El símbolo $ indica el final del String. En este caso el String debe contener la expresión al final. |
| [abc] | Los corchetes representan una definición de conjunto. En este ejemplo el String debe contener las letras a ó b ó c. |
| [abc] [12] | El String debe contener las letras a ó b ó c seguidas de 1 ó 2 |
| [^abc] | El símbolo ^ dentro de los corchetes indica negación. En este caso el String debe contener cualquier carácter excepto a ó b ó c. |
| [-A Z1-9] | Rango. Indica las letras minúsculas desde la a hasta la z (ambas incluidas) y los dígitos desde el 1 hasta el 9 (ambos incluidos)`. Solo 1. |
| A | B | El carácter | es un OR.  A ó B |
| AB | La concatenación. A seguido de B |

Meta caracteres en expresiones regulares

|  |  |
| --- | --- |
| **Expresión** | **Descripción** |
| \ D | Dígito. Equivale a [0-9] |
| \ D | No dígito. Equivale a [^0-9] |
| \ s | Espacio en blanco. Equivale a [ \t\n\x0b\r\f] |
| \ S | No espacio en blanco. Equivale a [^\s] |
| \ W | Una letra mayúscula o minúscula, un dígito o el carácter \_Equivale a  [a-zA-Z0-9\_] |
| \ W | Equivale a [^\w] |
| \ B | Límite de una palabra. |

Cuantificadores en expresiones regulares

|  |  |
| --- | --- |
| **Expresión** | **Descripción** |
| {X} | Indica que lo que va justo antes de las llaves se repite X veces |
| {X, Y} | Indica que lo que va justo antes de las llaves se repite mínimo X veces y máximo Y veces. También podemos poner {X,} indicando que se repite un mínimo de X veces sin límite máximo. |
| \* | Indica 0 ó más veces. Equivale a {0,} |
| + | Indica 1 ó más veces. Equivale a {1,} |
| ? | Indica 0 ó 1 veces. Equivale a {0,1} |

**Cómo usar los símbolos especiales de expresiones regulares en Java**

En Java debemos usar una doble barra invertida \\. Por ejemplo para utilizar \w tendremos que escribir \\w.

Si queremos indicar que la barra invertida es un carácter de la expresión regular tendremos que escribir  \\\\.

**Expresiones Regulares con Matches de Java**

Para usar expresiones regulares en Java se usa el package java.útil.regex

Contiene las clases Pattern y Matcher y la excepción PatternSyntaxException

**Algunos ejemplos en código de Expresiones Regulares con Matches de Java**// Comprobar si el String cadena contiene exactamente el patrón (matches) “abc”

     Patternpat Pattern.compile = ("abc");

      Matchermat = pat.matcher (Cadena);

     if (mat.matches ()) {

          System.out.println ("SI");

     } else {

          System.out.println ("NO");

   }

# UT8 Ejercicios

**EJERCICIOS DE EXCEPCIONES**

**1.-** Realiza un programa que calcule el valor del factorial de un número que se pide al usuario. Si el número es negativo debe saltar una excepción llamada FactorialNegExc creada por el programador que indique que no se puede calcular el factorial de un número negativo. En la cláusula finally escribirá el resultado, siempre que haya sido posible calcularlo.

**2.-** Realiza un programa en Java que permita crear cuentas bancarias pidiendo la cantidad inicial al usuario y el número de cuenta (no se permite repetir número de cuenta), así como realizar las operaciones ingresar y sacar dinero de esas cuentas. En el caso de intentar sacar de la cuenta corriente más dinero del que hay, se mostrará el mensaje asociado a una excepción denominada NoHayDineroExcepcion que crearás.

El programa constará de un menú con las siguientes opciones:

1. Abrir cuenta

2. Ingresar dinero en cuenta

3. Sacar dinero de cuenta

4. Visualizar todas las cuentas con su saldo actual

5. Mostrar una cuenta

6. Borrar una cuenta

7. Borrar todas las cuentas

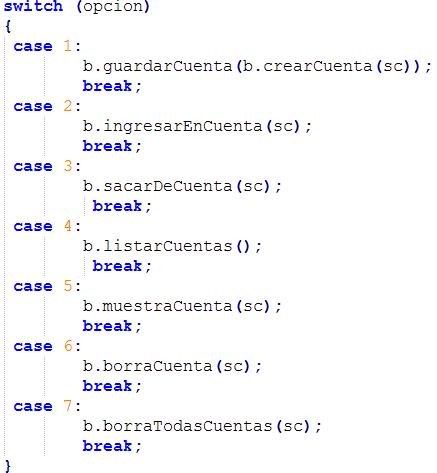
8. Salir

El orden al mostrar todas las cuentas será por saldo de mayor a menor, y si dos cuentas tienen el mismo saldo por orden creciente de cuenta. Utiliza el interfaz **Comparable**.

Para realizar el programa deberás distribuirlo en varios archivos:

**Cuenta.java** para desarrollar métodos y atributos propios de una cuenta corriente

**CuentaApp.java** que contendrá el método main y mostrará el menú con las opciones indicadas, realizando llamadas a los métodos que se muestran a continuación:

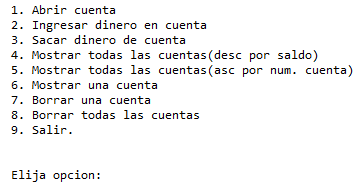


**Banco.java** que contendrá los métodos llamados en el menú, así como el arraylist de cuentas corrientes.

Con la clase **Lector.java**, que se te proporciona con el enunciado crearás un paquete externo que incorporarás en el proyecto.

Añadirás los archivos/clases necesarios para llevar a cabo los requerimientos pedidos.

Crea una **segunda versión** en la que las cuentas se puedan ver de dos formas distintas, ascendentemente por número de cuenta y descendentemente por saldo:



**Ejercicio X. CambiosDGTExcepciones**

**7.-** Definir los atributos de las clases de objetos necesarias para almacenar la información relativa a unas muestras de alcoholemia recogidas en un análisis preventivo de la Dirección General de Tráfico, de forma que:

* 1. Para cada conductor interesa tener su nombre, DNI. y una colección de referencias a las muestras que se le han tomado durante el período de estudio.
  2. Cada muestra tendrá los siguientes datos: Día y hora de la muestra, código del puesto de control preventivo (dos letras y dos números), matrícula del vehículo y la tasa de alcohol espirado en aire (entre 0 y 2.5 mg/l).

Hacer una simulación del funcionamiento de estas clases usando el siguiente menú:

1. Introducir un nuevo conductor.  
2. Introducir una muestra.  
3. Mostrar los datos de un conductor.  
4. Mostrar las muestras de un conductor.  
5. Mostrar los datos de todos los conductores.  
6. Salir.

Elige una opción:

Hacerlo con Arrays y ArrayList.

Al introducir una muestra, se elige de qué conductor por su dni.

Los datos del conductor se muestran buscando por dni.

Cambios pedidos:

1. Mostrar los conductores con su número de orden para identificar de qué conductor se quieren introducir muestras.
2. Hacer el control de formato de código y de alcohol con dos nuevas excepciones que crearás. (se vuelve a pedir en caso de saltar la excepción) La fecha y hora de las muestras será obtenida por programa como las actuales, no se piden al usuario ya.

**Ejercicio Vehículos**

1. Se quiere implementar un simulador de vehículos. Existen dos clases de Vehiculos: Cochey Camion.

1. De todos los Vehiculos queremos almacenar la matrícula y la velocidad. En el momento  
   de crearlos, la matricula se recibe como parámetro y la velocidad se pone a 0. En su  
   correspondiente método toString() se devolverá la matrícula y la velocidad. Además se  
   puede acelerar pasando como parámetro la cantidad en km/h que se tiene que acelerar.
2. Los coches tienen además un atributo para el número de puertas que se recibe también  
   como parámetro a la hora de crearlo. Y contiene un método que devuelve el número de  
   puertas.
3. Los camiones tienen un atributo del tipo de Remolque que se inicializa a null en el  
   momento de crear el camión para indicar que aún no tiene remolque. Definimos en la  
   clase dos métodos, ponRemolque que recibe un Remolque por parámetro y otro  
   quitaRemolque que pone el atributo a null. Cuando se muestre la información que lleve  
   remolque, además de matrícula y velocidad debe aparecer información del remolque.
4. En la clase Camion hay que sobrescribir el método de acelerar, de forma que si el  
   camión lleva remolque y la velocidad supera los 100 km/h debe levantarse una  
   excepción del tipo DemasiadoRapidoExcepcion.
5. Hay que implementar la clase Remolque que tiene un atributo de tipo entero que es el  
   peso del remolque y su valor se le asigna a la hora de crear el Remolque. Tendrá su  
   método toString() como todas las demás clases.
6. Se implementará la clase DemasiadoRapidoExcepcion.
7. Crearemos la clase PruebaVehiculo con un método main donde se tenga un menú con  
   las siguientes operaciones (utiliza un ArrayList para guardar los vehículos que se creen  
   desde el menú):

1.- Añadir coche2.- Añadir camión3.- Añadir un remolque (de peso en kg el que indique el usuario) a todoslos camiones4.- Muestre la información de todos los vehículos por pantalla.5.- Borre todos los camiones que tengan un remolque de más de 1000 kg6.- Salir

Para practicar ordenación con interfaces de Java, prueba a mostrar los vehículos  
ordenados por todos los criterios que se te ocurran.