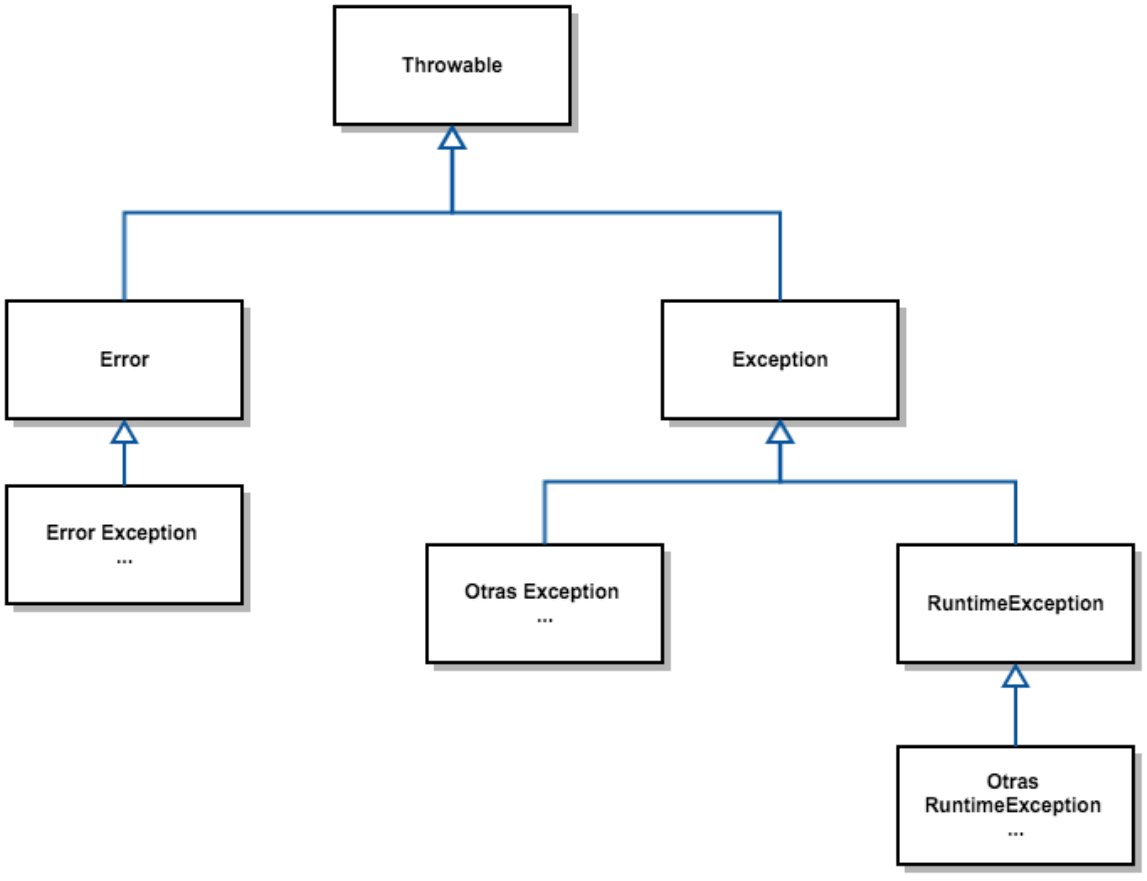
5.2 Throws, throw y finally

1. Tipos de excepciones

En Java, la **herencia** define una relación entre clases en la cual una clase posee características (métodos y propiedades) que proceden de otra. A la clase que posee las características a heredar se la llama **superclase** o **clase padre** y la clase que las hereda se llama **subclase** o **clase hija**. Lo beneficioso de este tipo de relaciones es que los hijos pueden heredar las características de los padres y luego aportar características propias convirtiéndose en una manera de reutilizar código muy eficiente.

Java utiliza muchísimo la herencia en las clases que conforman su API. En cuanto a las excepciones se refiere, existen tres tipos de excepciones que Java maneja a través de la herencia: errores, comprobadas (*checked*) y no comprobadas (*unchecked*). El gráfico que se muestra a continuación muestra el árbol de herencia de las excepciones en Java. El paquete de todas es java.lang:



La clase principal de la cual heredan todas las excepciones Java es Throwable. De ella nacen dos ramas: Error y Exception. Error representa errores de una magnitud tal que una aplicación nunca debería intentar realizar nada con ellos (como errores de la JVM, desbordamientos de buffer, etc). La segunda rama, encabezada por Exception, representa aquellos errores que normalmente sí solemos gestionar, y a los que comúnmente solemos llamar *excepciones*.

De Exception nacen múltiples ramas: ClassNotFoundException, IOException, ParseException, SQLException y otras muchas, todas ellas de tipo *checked*. La única excepción que es de tipo *unchecked* es RuntimeException y encabeza todas las de este tipo.

Excepciones checked

Una excepción de tipo checked representa un error del cual técnicamente podemos recuperarnos. En ciertos momentos, a pesar de la promesa de recuperabilidad, nuestro código no estará preparado para gestionar la situación de error, o simplemente no será su responsabilidad. En estos casos lo más razonable es relanzar la excepción y confiar en que un método superior en la cadena de llamadas sepa gestionarla.

Por tanto, todas las excepciones de tipo checked deben ser capturadas o relanzadas. En el primer caso, utilizamos el bloque try-catch. En caso de querer relanzar la excepción, debemos declarar dicha intención en la firma de la función que contiene las sentencias que lanzan la excepción, y lo hacemos mediante la claúsula throws.

Hay que tener presente que cuando se relanza una excepción estamos forzando al código cliente de nuestro método a capturarla o relanzarla. Una excepción que sea relanzada una y otra vez hacia arriba terminará llegando al método primigenio y, en caso de no ser capturada por éste, producirá la finalización de su hilo de ejecución (thread).

La dos preguntas que debemos hacernos en este momento es: ¿Cuándo capturar una excepción? ¿Cuándo relanzarla? La respuesta es muy simple. Capturamos una excepción cuando:

- Podemos recuperarnos del error y continuar con la ejecución.
- Queremos registrar el error.
- Queremos relanzar el error con un tipo de excepción distinto.

En definitiva, cuando tenemos que realizar algún tratamiento del propio error. Por contra, relanzamos una excepción cuando:

- No es competencia nuestra ningún tratamiento de ningún tipo sobre el error que se ha producido.
- Para centralizar el control de excepciones y facilitar el mantenimiento del código.

Excepciones unchecked

Son excepciones de tipo RuntimeException o de cualquiera de sus subclases. El aspecto más destacado de las excepciones de tipo unchecked es que no deben ser forzosamente declaradas ni capturadas (en otras palabras, no son comprobadas). Por ello no son necesarios bloques try-catch ni declarar formalmente en la firma de la función el lanzamiento de excepciones de este tipo. Ésto, por supuesto, también afecta a funciones y/o clases más hacia arriba en la cadena invocante.

2. Throws

En caso de querer relanzar la excepción, debemos declarar dicha intención en la firma de la función que contiene las sentencias que lanzan la excepción, y lo hacemos mediante la claúsula throws.

¿Tiene obligación de tener un try-catch el que llame a la función? Si la excepción es de tipo RuntimeException o cualquiera de sus hijas, no existe la obligación de poner un try-catch. En cualquier otro caso, sí existe dicha obligación, de hecho si no se hace, Java da un error de compilación.

Ejemplo:

```
package tema5_2_ThrowsThrowFinally;

import java.util.Scanner;

public class Throws {

    @SuppressWarnings("resource")
    public void show() {

        final String FIN = "fin", SIGUIENTE = "siguiente";
        int number;
        String string;
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Introduce un número o siguiente para pasar al siguiente
número: ");
        string = keyboard.nextLine();
        if (!string.toLowerCase().equals(SIGUIENTE)) {
            /*
             * Aquí no estamos obligados a poner un try-catch porque
             * NumberFormatException es hija de RuntimeException:
             */
            number = convertirNumero(string);
            System.out.printf("Valor del número introducido: %d\n", number);
        }

        try {
            System.out.print("Introduce un número o fin para terminar: ");
            string = keyboard.nextLine();
            if (!string.toLowerCase().equals(FIN)) {
                /*
                 * Aquí sí estamos obligados a poner un try-catch
                 * porque Exception no es hija de RuntimeException:
                 */
                number = convertirNumero2(string);
                System.out.printf("Valor del número introducido: %d\n", number);
            }
        } catch (Exception e) {
            System.err.println("Error en el número");
        }

    }
```

```
    public int convertirNumero(String string) throws NumberFormatException {

        return Integer.parseInt(string);

    }

    public int convertirNumero2(String string) throws Exception {

        return Integer.parseInt(string);

    }

    public static void main(String[] args) {

        new Throws().show();

    }

}
```

3. Orden de los catch

Cuando se tienen varios catch para el mismo try, hay que tener la precaución de poner los catch de las subclases antes que los de las superclases porque si no, los catch de las subclases nunca se capturarían. Ejemplo:

```
package tema5_2_ThrowsThrowFinally;

import java.util.InputMismatchException;
import java.util.Scanner;

public class CatchOrder1 {

    @SuppressWarnings("resource")
    public void show() {

        final String FIN = "fin";
        String string;
        byte number = 0;
        Scanner keyboard = new Scanner(System.in);

        try {
            System.out.print("Introduce un número de tipo byte, es decir, entre
-128 y 127: ");
            number = keyboard.nextByte();
            keyboard.nextLine(); //Limpieza del buffer
            System.out.printf("Valor del número introducido: %d\n", number);
            System.out.print("Introduzca otro número de tipo byte o fin para
terminar: ");
            string = keyboard.nextLine();
            if (!string.toLowerCase().equals(FIN)) {
                number = Byte.parseByte(string);
                System.out.printf("Valor del otro número introducido: %d\n",
number);
```

```
            }
        } catch (Exception e) {
            System.err.println("Error: no ha introducido un número entre -128 y
127");
        } catch (InputMismatchException e) {//Error de compilación: inalcanzable
bloque catch
            System.err.println("Error");
        }

    }

    public static void main(String[] args) {

        new CatchOrder1().show();

    }

}
```

En este caso, Java da un error de compilación ya que InputMismatchException es hija de Exception. Si el usuario no introduce un número de tipo byte, se lanza una excepción de tipo InputMismatchException. Dicho objeto también es un objeto de tipo Exception, ya que los objetos de tipo hijo, también son objetos de tipo padre. Por lo tanto, nunca entraría en el catch del InputMismatchException. La solución es cambiar el orden de los catch:

```
package tema5_2_ThrowsThrowFinally;

import java.util.InputMismatchException;
import java.util.Scanner;

public class CatchOrder2 {

    @SuppressWarnings("resource")
    public void show() {

        final String FIN = "fin";
        String string;
        byte number = 0;
        Scanner keyboard = new Scanner(System.in);

        try {
            System.out.print("Introduce un número de tipo byte, es decir, entre
-128 y 127: ");
            number = keyboard.nextByte();
            keyboard.nextLine(); //Limpieza del buffer
            System.out.printf("Valor del número introducido: %d\n", number);
            System.out.print("Introduzca otro número de tipo byte o fin para
terminar: ");
            string = keyboard.nextLine();
            if (!string.toLowerCase().equals(FIN)) {
                number = Byte.parseByte(string);
                System.out.printf("Valor del otro número introducido: %d\n",
number);
            }
        } catch (InputMismatchException e) {
```

```
            System.err.println("Error: no ha introducido un número entre -128 y
127");
        } catch (Exception e) {
            System.err.println("Error");
        }

    }

    public static void main(String[] args) {

        new CatchOrder2().show();

    }

}
```

4. Throw

Esta instrucción le permite al programador lanzar una excepción. El flujo del programa se dirigirá a la instrucción try-catch más cercana. Ejemplos:

- `throw new Exception();`
- `throw new Exception("Error grave");` Construye una excepción con el mensaje indicado. Ese mensaje se puede obtener en el catch con el método *getMessage* del objeto que contiene la excepción. Ejemplo:

```
package tema5_2_ThrowsThrowFinally;

import java.util.Scanner;

public class Throw {

    @SuppressWarnings("resource")
    public void show() {

        double n;
        Scanner keyboard = new Scanner(System.in);

        try {
            System.out.println("Introduce un número: ");
            n = keyboard.nextDouble();
            division(n);
        } catch (ArithmeticException e) {
            System.err.println(e.getMessage());
        }

    }

    public void division(double n) {
        /*
         * float y double admiten Infinity por lo que no se lanza
         * una excepción cuando se divide entre cero.
         * Podemos lanzar la excepción usando throw
         * */
```

```
        if (n == 0) {
            throw new ArithmeticException("Error: no se puede dividir entre
cero");
        }
        System.out.printf("500 entre %.2f: %.2f\n", n, 500 / n);

    }

    public static void main(String[] args) {

        new Throw().show();

    }

}
```

5. Finally

Las instrucciones que se encuentran en el **finally** se ejecutan independientemente de si hubo o no excepción. Es decir, esas sentencias se ejecutan siempre. Si el código del try ha ido bien y no se ha lanzado ninguna excepción, después se ejecutan las instrucciones del *finally* y luego se continúa por el código que le sigue al bloque del try-catch. Pero si se ha lanzado alguna excepción, se ejecutan las instrucciones del catch correspondiente, luego se ejecutan las instrucciones del *finally* y luego se continúa por el código que le sigue al bloque del try-catch. Por lo tanto, se coloca en el bloque *finally* código común para todas las excepciones y también para cuando no hay excepciones. Normalmente se suelen poner instrucciones de limpieza. Ejemplo:

```
package tema5_2_ThrowsThrowFinally;

import java.util.InputMismatchException;
import java.util.Scanner;

public class Finally {

    @SuppressWarnings("resource")
    public void show() {

        int number = 0;
        String string;
        boolean error = false;
        Scanner keyboard = new Scanner(System.in);

        try {
            System.out.print("Introduce un número: ");
            number = keyboard.nextInt();
        } catch (InputMismatchException e) {
            System.err.println("Error");
            error = true;
        } finally {
            keyboard.nextLine();//Limpieza del buffer
        }

        System.out.print("Introduce una cadena: ");
        string = keyboard.nextLine();
```

```
        System.out.printf("El número introducido ha sido: %s\n", error ? "error"
: number);
        System.out.printf("La cadena introducida ha sido: %s", string);

    }

    public static void main(String[] args) {

        new Finally().show();

    }

}
```

En el ejemplo, hay que limpiar el buffer haya o no una excepción, por lo tanto, se limpia en el *finally*.

Pero ¿qué diferencia hay entre poner código en el *finally* o ponerlo después del bloque try-catch? Pues que en el *finally* siempre se va a ejecutar aunque se rompa el flujo de ejecución, es decir, aunque exista por ejemplo un `return` en el *try* o un `throw` en el *catch*. El código que se encuentre después del bloque try-catch, no se ejecutaría en esos casos.