¿Cómo se manejan los errores en Java?

Imaginemos que tenemos un método en una clase que se encarga de hacer LOGIN en el sistema. Durante la ejecución del método, podemos tener diferentes errores:

- La contraseña es incorrecta
- El usuario no existe
- No ha conexión con la BBDD en ese momento
- Cualquier otro error inesperado

¿Cómo gestionamos y manejamos estos errores? Se hace utilizando unos objetos que son excepciones.

¿Qué es una Excepción?

Es un objeto de una clase. Igual que todos los objetos en java. Pero es de alguna clase que extiende obligatoriamente de la clase Throwable o Exception (Exception es hija de Throwable)

Los objetos excepciones tienen una peculiaridad: se pueden "lanzar" y "capturar" desde el código. ¿Y qué es eso de "lanzar" y "capturar"?

- Lanzar → Por ejemplo, cuando intento dividir un número entre cero, esto genera un error. Entonces, se crea una excepción y se lanza (se lanza dicho error) para que la ejecución se pare.
- Capturar → Ese error, la excepción que se ha lanzado en algún sitio, se puede capturar desde donde se ha invocado la división para dar un tratamiento específico al error. Por ejemplo, mostrar un mensaje o poner como resultado de la división un valor constante.

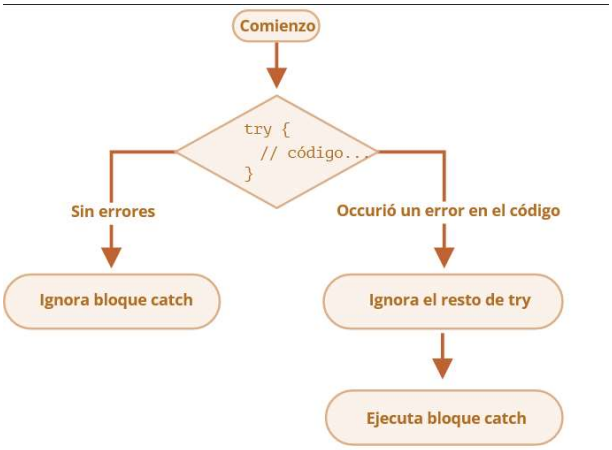
¿Cómo se captura una excepción?

Para capturar una excepción utilizamos la estructura de código try / catch / finally. Se escribe así:

```
try {
    // aquí estaría nuestro código
    // que puede tener tantas líneas
    // como queramos. Si salta una excepción
    // dejaría de ejecutarse

} catch (TipoDeExcepcion nombreVariable) {
    // aquí se ejecutaría el código si se lanza una
    // excepción del tipo "TipoDeExcepcion" en el
    // bloque try
} finally {
    // esto se ejecutaría SIEMPRE, haya o no
    // excepciones que se lancen, se capturen o no
}
```

El siguiente esquema representa cómo funciona:



Si existiera el bloque finally, se ejecutaría siempre, independientemente de si hay o no errores.

De una manera más completa se ve en este otro diagrama:

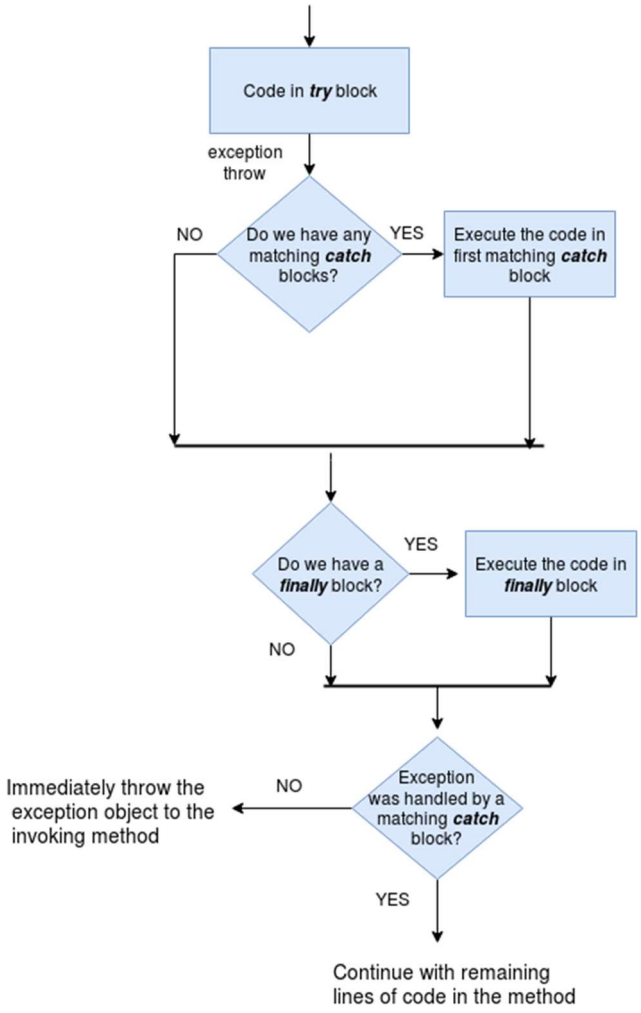


Fig : Flow Diagram of try-catch and finally in Java

Hay que tener en cuenta lo siguiente:

- No es obligatorio tener siempre un finally → Puedo utilizar la estructura try/catch sin incluir finally porque no hay ningún código que quiera ejecutar siempre al final independientemente de si hay errores
- No es obligatorio tener siempre un catch → Puedo utilizar la estructura try/finally para asegurarme de ejecutar algo al final, aunque haya algún error, pero no quiero capturar dichos errores ni hacer nada con ellos.

¿Cómo se captura más de una excepción para el mismo código?

Puede que en el mismo bloque try/catch/finally quiera capturar varias excepciones. Esto lo puedo hacer de diferentes maneras.

- Si queremos ejecutar código diferente según el tipo de excepción que se lance, haríamos lo siguiente:

```
try {

} catch (TipoDeExcepcion1 nombreVariable) {
    // aquí se ejecutaría el código si se lanza una
    // excepción del tipo "TipoDeExcepcion1"
} catch (TipoDeExcepcion2 nombreVariable) {
    // aquí se ejecutaría el código si se lanza una
    // excepción del tipo "TipoDeExcepcion2"
} finally {
    // esto se ejecutaría SIEMPRE
}
```

- Si queremos ejecutar el mismo código independientemente del tipo de excepción que se lance, haríamos lo siguiente:

```
try {

} catch (TipoDeExcepcion1 | TipoDeExcepcion2 nombreVariable) {
    // aquí se ejecutaría el código si se lanza una
    // excepción del tipo "TipoDeExcepcion1" o "TipoDeExcepcion2"
}  finally {
    // esto se ejecutaría SIEMPRE
}
```

- Si queremos capturar todas las excepciones, podemos capturar Exception o Throwable, que son las clases de las que heredan todas las excepciones con las que trabajamos:

```
try {

} catch (Throwable e) {
    // aquí iría el código que se ejecutará si
    // se lanza cualquier excepción (sea del tipo
    // que sea)
}  finally {
    // esto se ejecutaría SIEMPRE
}
```

¿Podemos anidar los bloque try/catch/finally?

Sí. Al igual que los bloque if/else o los bucles, siempre podemos meter un try/catch/finally dentro de otro.

Ejemplo 1:

```
try {
    // código dentro del primer try...
    try {
        // código dentro del segundo try...
    } catch(TipoDeExcepcion1 nombreVariable1) {
        // captura de excepción del código del segundo try...
    } finally {
        // esto se ejecutaría SIEMPRE que se
        // haya empezado ya a ejecutar el segundo try
    }
    // más código dentro del primer try...

} catch (TipoDeExcepcion2 nombreVariable2) {
    // captura de excepción del código del primer try...
} finally {
    // esto se ejecutaría SIEMPRE
}
```

Ejemplo 2:

```
try {
    // código dentro del primer try...
} catch (TipoDeExcepcion1 nombreVariable1) {
    // código por si se lanza TipoDeExcepcion1...
    try {
        // código por si se lanza TipoDeExcepcion1,
        // pero dentro de un segundo try
    } catch(TipoDeExcepcion2 nombreVariable2) {
        // captura de excepción del código del segundo try...
    } finally {
        // esto se ejecutaría SIEMPRE que se
        // haya empezado ya a ejecutar el segundo try
    }
    // más código por si se lanza TipoDeExcepcion1...

} finally {
    // esto se ejecutaría SIEMPRE
}
```

Ejemplo 3:

```
try {
    // código dentro del primer try...
} catch (TipoDeExcepcion1 nombreVariable1) {
    // código por si se lanza TipoDeExcepcion1...
} finally {
    // esto se ejecutaría SIEMPRE
    try {
        // código que se ejecutaría SIEMPRE,
        // pero dentro de un segundo try
    } catch(TipoDeExcepcion2 nombreVariable2) {
        // captura de excepción del código del segundo try...
    } finally {
        // esto se ejecutaría SIEMPRE que se
        // haya empezado ya a ejecutar el segundo try
    }
}
```

¿Cómo se lanza una excepción?

Hemos visto antes como capturar una excepción que se ha lanzado desde otro método para tratar ese error. También podemos nosotros mismos, desde uno de nuestros métodos, lanzar una excepción para generar el "error".

Para esto, utilizamos la palabra clave throw. Por ejemplo:

```
public Integer dividir(Integer a, Integer b) {
    if (a == 0) {
        ArithmeticException e = new ArithmeticException("El denominador no puede ser 0");
        throw e;
    }
    return a / b;
}
```

Normalmente se escribe todo en una única línea:

```
public Integer dividir(Integer a, Integer b) {
    if (a == 0) {
        throw new ArithmeticException("El denominador no puede ser 0");
    }
    return a / b;
}
```

IMPORTANTE: Cuando lanzamos una excepción, el código ya no continúa su ejecución. Es como si hiciéramos un return. Salvo que estemos dentro de un bloque try/catch/finally. En este caso:

- El bloque finally siempre se ejecutará → Lo que hay en el finally se ejecuta siempre, aunque hagamos un throw o un return desde el bloque try.
- El bloque catch, si estoy capturando la misma excepción que estoy lanzando, también se ejecutará.

Ejemplo 1:

```java
public Integer dividir(Integer a, Integer b) {
    try {
        if (a == 0) {
            throw new ArithmeticException("El denominador no puede ser 0");
        }
    }
    catch(ArithmeticException e) {
        System.out.println("Esto se ejecuta porque estoy capturando lo que he lanzado");
        // La excepción que he lanzado más arriba queda aquí capturada y no se lanza hacia arriba
    }
    return a / b;
}
```

Ejemplo 2:

```java
public Integer dividir(Integer a, Integer b) {
    try {
        if (a == 0) {
            throw new ArithmeticException("El denominador no puede ser 0");
        }
        return a / b;
    }
    finally {
        System.out.println("Esto se ejecuta siempre");
    }
}
```

¿Hay que capturar todas las excepciones?

Digamos que hay dos tipos de excepciones:

- Las que no es obligatorio capturar mientras programamos. Estas se supone que son errores de la máquina virtual de java (Java Runtime) y que, en principio, no son errores que el usuario pueda hacer nada para corregirlos. Por ejemplo, NullPointerException o ArithmeticException.
- Las que sí es obligatorio capturar mientras programamos. Estas son errores que se deben a algún problema más probable que ocurra, debido a circunstancias del contexto de ejecución (un fichero que no existe, una conexión que no se ha podido establecer, un password incorrecto, etc.). Por ejemplo, IOException.
  En este caso, al programar, Java nos obliga a "hacer algo" con estas excepciones. Tenemos dos opciones:
  - O bien las capturamos con un try / catch
  - O bien indicamos en nuestro método que puede que lancemos ese tipo de excepciones. ¿Cómo se hace esto? → Con la palabra reservada throws:

```java
public void leerFichero(String nombreFichero) throws IOException{
    if (nombreFichero.isEmpty()) {
        throw new IOException("Nombre de fichero incorrecto");
    }
    // ... resto del código ...
    // ...

}
```

¿Cómo podemos crear nuestras propias excepciones?

Para crear nuestras propias excepciones basta con crearnos una clase que herede de alguna otra excepción. Normalmente, se extiende de alguna de estas clases:

- Throwable o Exception → Para crear excepciones genéricas que tendrán que ser capturadas por el programador. Son las más comunes.
- RuntimeException → Parar crear excepciones genéricas que NO tendrán que ser capturadas por el programador.
- Otras excepciones comunes para ser más específico (IOException, ArithmeticException, etc.)

Cuando creamos la clase, se suelen añadir todos los constructores llamando a los constructores de la clase padre. Esto nos lo hace Eclipse automáticamente.

Los nombres de las excepciones que creemos deben guardar las mismas restricciones que los nombres de cualquier otra clase. Además, por convención, deben terminar en la palabra "Exception". Por ejemplo: LoginIncorrectoException, AccesoDenagadoException, etc.

También es recomendable generar automáticamente un serialVersionUUID.

Ejemplo de excepción que extiende de Exception:

```
public class AccesoDenegadoException extends Exception {

    private static final long serialVersionUID = -6058429112934198190L;

    public AccesoDenegadoException() {
    }
    public AccesoDenegadoException(String message) {
        super(message);
    }
    public AccesoDenegadoException(Throwable cause) {
        super(cause);
    }
    public AccesoDenegadoException(String message, Throwable cause) {
        super(message, cause);
    }
    public AccesoDenegadoException(String message, Throwable cause, boolean enableSuppression,
            boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```