Índice

Introducción

Las excepciones en Java son **eventos anómalos o inesperados que ocurren en el proceso de ejecución de un programa**. Estas excepciones pueden ser causadas por una variedad de factores, incluyendo errores de programación, entrada incorrecta por parte del usuario, problemas en el hardware y problemas en el sistema.

Java maneja las excepciones mediante el uso de la clase `Exception` y sus subclases. Cuando una excepción ocurre, el programa se detiene y se lanza una excepción, la cual puede ser capturada y manejada por el programador. El manejo adecuado de las excepciones es importante para garantizar una ejecución segura y estable de los programas en Java.

Excepciones más comunes

Las excepciones más comunes en Java son:

1. **Entrada/salida incorrecta**: Cuando se intenta leer o escribir en un archivo que no existe o que no tiene permisos adecuados, puede producirse una excepción de entrada/salida.
2. **Dividir por cero**: Una operación matemática que intenta dividir por cero lanzará una excepción de `ArithmeticException`.
3. **Referencia a `null`**: Intentar acceder a un objeto nulo puede provocar una `NullPointerException`.
4. **Tipos incompatibles**: Intentar asignar un objeto a una variable que no es compatible con su tipo puede provocar una `ClassCastException`.
5. **Índices fuera de rango**: Intentar acceder a un elemento en un array o una colección fuera de sus límites puede provocar una `ArrayIndexOutOfBoundsException`. También una referencia a una posición fuera de los límites de una cadena provocaría un `StringIndexOutOfBoundsException`.

6. **Conversión de tipos inválida:** Intentar convertir un tipo de datos a otro que no es compatible puede provocar una `NumberFormatException`.
7. **Excepciones con bases de datos SQL**: Provocan una excepción del tipo `SQLException`. Este tipo de excepción se produce cuando hay un problema con una base de datos, como un error en la conexión o una consulta mal formada.
8. **Error de sintaxis:** Un error de sintaxis en el código fuente, como una falta de paréntesis o una declaración inválida, puede provocar una `SyntaxError` o una `CompilationException`.

Estas son solo algunas de las muchas causas que pueden provocar una excepción en Java. Es importante tener en cuenta que las excepciones son una forma de notificar al programador sobre un problema en tiempo de ejecución, para que pueda solucionarlo antes de que el programa se detenga o produzca resultados incorrectos.

Tipos de excepciones

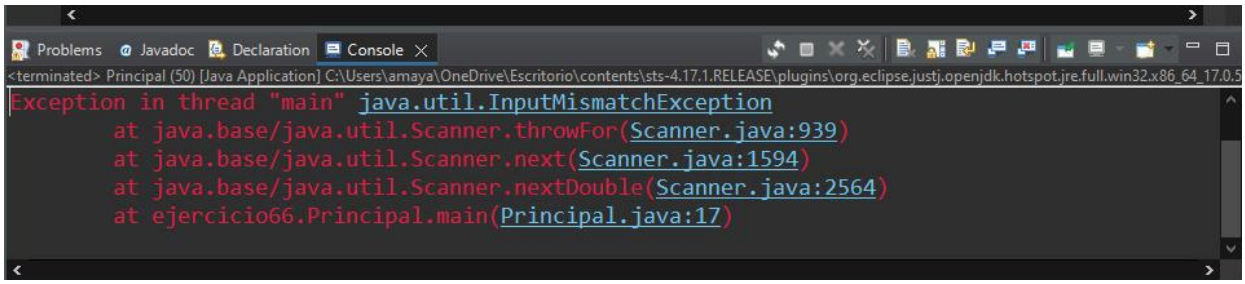
Las excepciones en Java se pueden clasificar en dos tipos: **unchecked** y **checked**.

Unchecked Exception

Las excepciones unchecked son aquellas que **no se deben gestionar explícitamente**, ya que Java las considera errores en tiempo de ejecución. Estas excepciones incluyen `NullPointerException`, `ArrayIndexOutOfBoundsException`, etc. Es importante tener en cuenta que, aunque estas excepciones no sean necesarias de gestionar explícitamente, es recomendable hacerlo para poder proporcionar una respuesta adecuada al usuario o para poder solucionar el problema que ha generado la excepción.

Un ejemplo de excepción unchecked es la que se produce cuando el usuario debe introducir un número con `scanner.nextInt()`, e introduce letras. Esto provocará una excepción de tipo `InputMismatchException` y el programa finalizará. Como Java la excepción depende de los datos que introduzca el usuario, no nos obligarán a capturar la excepción porque puede que no llegue a lanzarse.

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Introduce un número: ");
3 double numero = scanner.nextDouble(); // El usuario introduce el texto "No
  quiero", en lugar de un número
4 System.out.println(numero); // Nunca llegaría a ejecutarse esta línea
```

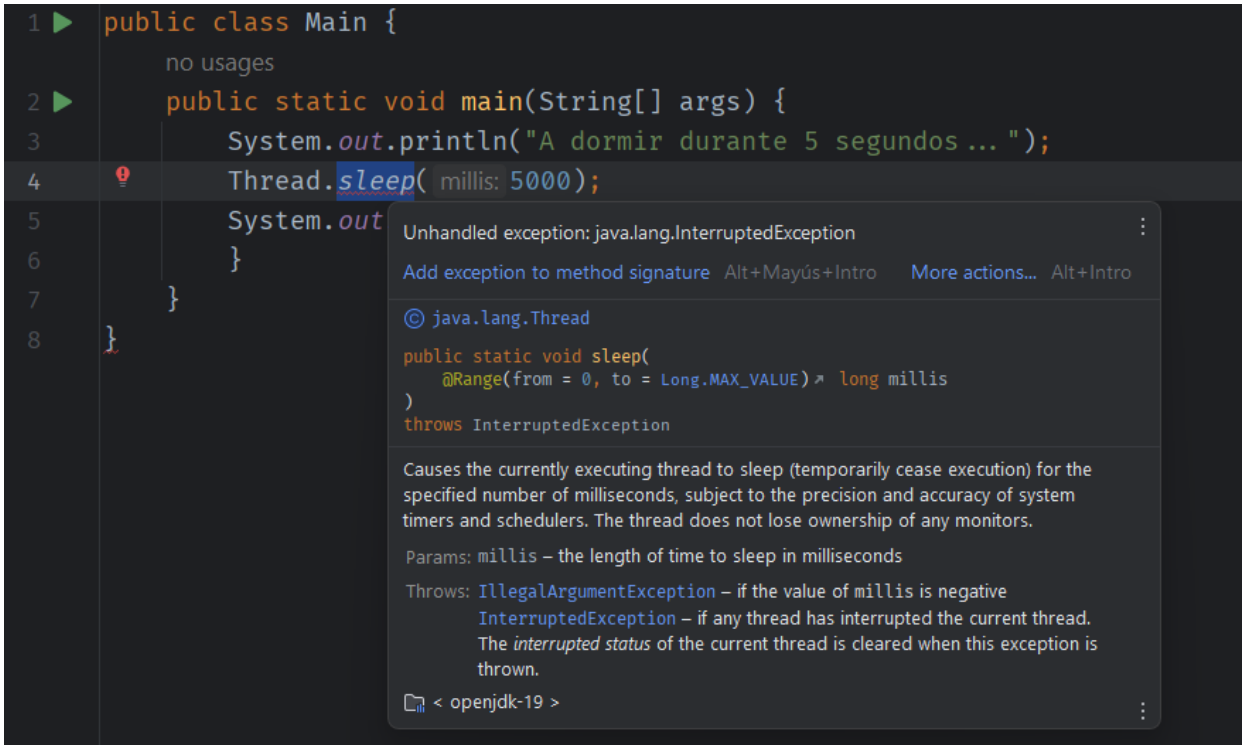


Checked Exception

Las excepciones checked son aquellas que **se deben gestionar explícitamente** por el programador. Estas excepciones están definidas por las clases de la biblioteca de Java, como por ejemplo `IOException`, `SQLException`, etc.

Cuando un método puede generar una excepción checked, el IDE nos "obligará" a que se capture y gestione en el código. Esto se hace mediante un bloque try-catch, en el que se pone el código que puede generar la excepción, y el catch se encarga de gestionarla.

Un ejemplo de excepción checked sería el uso de la función `Thread.sleep()`. La función lo que hace es mandar a "dormir" el hilo de ejecución durante los milisegundos que se le indique. Tan solo con usar la función, los IDES nos avisarán de que esa función puede lanzar una excepción por lo que estamos obligados a tratarla.



Ante una excepción, los IDE (Eclipse, STS, Visual Studio o IntelliJ) nos sugerirá dos posibilidades (que veremos más detenidamente en siguientes apartados):

- 👋💣 Que en el método en el que estamos, también tenemos que definir que puede lanzar ese tipo de excepciones (es como pasar la patata caliente al método que invocó a nuestro método).
- 🤜🛡️ Capturar la excepción con un bloque `try-catch`. Esto es como "hacer una explosión controlada y sin daños". Lo veremos a continuación.

Diferencias

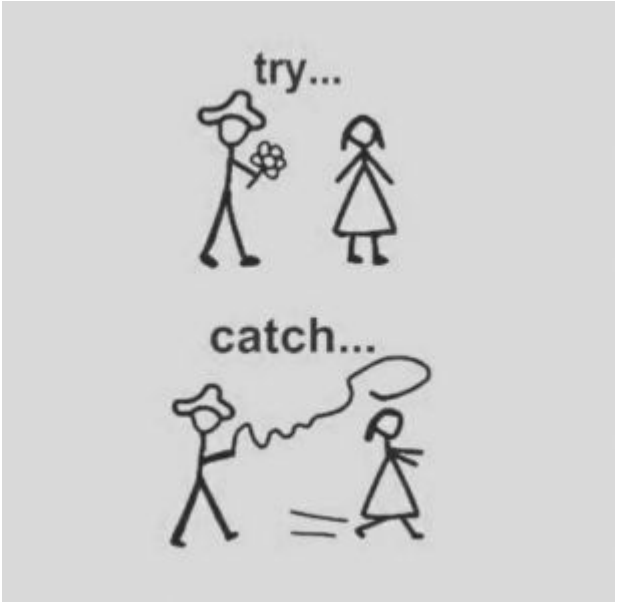
En resumen, las excepciones checked nos obliga el propio IDE a gestionarlas de forma explícita para evitar problemas en tiempo de ejecución, mientras que las excepciones unchecked no son necesarias de gestionar (el IDE no nos obliga a hacer nada), pero es recomendable hacerlo para mejorar la calidad del código.

Tratamiento de excepciones

Java tiene dos mecanismos para el tratamiento de excepciones: el bloque `try-catch` y la declaración `throws`.

Bloque try-catch

El bloque `try-catch` permite ejecutar un código que puede generar una excepción dentro del bloque `try`, y en caso de que se produzca, el control se pasa al bloque `catch` correspondiente, donde se puede tomar acción para tratar la excepción. Este mecanismo permite tratar excepciones específicas y personalizar la respuesta ante ellas.



La sintaxis del `try-catch` es la siguiente:

```
...
try {
    // Bloque de instrucciones que pueden generar una excepción
    // Sentencia 1
    // Sentencia 2
    // ...
} catch (Exception e) {
    // Bloque que se ejecutará cuando se produzca una excepción
} finally {
    // Bloque opcional que se ejecutará SIEMPRE
}
...
```

Un ejemplo de uso del `try-catch` en Java:

```java
public class Main {
    public static void main(String[] args) {
        int[] numeros = {1, 2, 3};
        try {
            System.out.println(numeros[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Ocurrió un error: " + e.getMessage());
        }
        System.out.println("Continuamos con la ejecución del programa");
    }
}

```

En este ejemplo, tratamos de acceder a una posición fuera del rango del array `numeros`. Esto causará una `ArrayIndexOutOfBoundsException`, que será capturada por el bloque `catch` y mostrará un mensaje de error en la consola. Después, el programa continúa con su ejecución normal.

El bloque `catch` captura la excepción y la guarda en una variable llamada `e` del tipo `ArrayIndexOutOfBoundsException`. Esa variable tiene información concreta de la excepción producida (como el mensaje de error) y podrá usarse dentro del bloque `catch` para mostrar esa información o conocer detalles concretos (por ejemplo, la sentencia SQL que provocó la excepción de tipo `SQLException`).

Sin usar un bloque `try-catch`, el intentar acceder a la posición 5 del array provocará la excepción del tipo `ArrayIndexOutOfBoundsException` y el programa terminará la ejecución. El usarlo nos ha permitido "contener la explosión" de una forma segura y continuar la ejecución normal del programa.

Bloque Finally

El bloque `finally` es una sección opcional en un `try-catch` que se ejecuta después de que se haya ejecutado el bloque `try` y cualquier bloque `catch` correspondiente.

Es útil para realizar tareas de limpieza o cierre, como cerrar recursos abiertos o liberar recursos, independientemente de si se ha producido una excepción o no.

Por ejemplo, si abres un archivo en un bloque `try`, puedes cerrarlo en el bloque `finally` para garantizar que se cierra siempre, incluso si se produce una excepción.

```java
public class Main {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
            System.out.println("El resultado es: " + result);
        } catch (ArithmeticException e) {
            System.out.println("No se puede dividir por cero");
        } finally {
            System.out.println("Este mensaje se imprimirá siempre, independientemente de si hubo una excepción o no.");
        }
    }
}

```

¿Es lo mismo imprimir la línea en el `finally` que fuera del `try-catch`? Si colocas una sentencia fuera del bloque `try-catch`, se ejecutará siempre, independientemente de si se produce una excepción o no. En ese sentido, el bloque `finally` no aporta ninguna ventaja en este caso.

Sin embargo, el bloque `finally` puede ser útil en situaciones en las que quieras asegurarte de que se ejecute un código determinado después de un bloque `try-catch`, independientemente de si se produce una excepción o no. Por ejemplo, puede ser útil para cerrar un archivo o una conexión con una base de datos que se abrió dentro del bloque `try`, o para liberar recursos que fueron asignados en el bloque `try`.

Try-catch múltiple

El `try-catch` múltiple permite capturar más de un tipo de excepción dentro de un mismo bloque `try`. Esto se hace agregando múltiples bloques `catch` después del bloque `try`. Cada bloque `catch` especifica una excepción diferente que se quiere capturar.

De esta manera, se pueden manejar de forma específica las excepciones que pueden ocurrir en el código dentro del bloque `try`. La sintaxis para el `try-catch` múltiple es la siguiente:

```java
try {
    // Código que puede generar excepciones
} catch (Excepción1 e1) {
    // Código para manejar la excepción 1 (La más específica)
} catch (Excepción2 e2) {
   // Código para manejar la excepción 2
} catch (Excepción3 e3) {
   // Código para manejar la excepción 3 (La más genérica)
}
```

> ⚠️ **IMPORTANTE**: Hay que tener en cuenta que cada bloque `catch` debe especificar una excepción diferente y que se deben colocar en orden de la más específica a la más genérica, ya que si una excepción ya ha sido capturada por un bloque `catch`, las siguientes excepciones ya no se ejecutarán.

Un ejemplo de código con `try-catch` múltiple:

```java
public static void main(String[] args) {
    int[] numeros = {1, 2, 3};
    Scanner scanner = null; // jijiji
    try {
        System.out.println(numeros[3]);
        scanner.nextInt();
    } catch (ArrayIndexOutOfBoundsException e1) {
        System.out.println("Error: se ha intentado acceder a un índice fuera del array");
    } catch (Exception e2) {
        System.out.println("Se ha producido una excepción no esperada");
    }
}
```

En este caso, en el bloque `try` se intenta acceder a un índice fuera del array, lo que provocará una excepción de tipo `ArrayIndexOutOfBoundsException`. En el primer `catch` se maneja esa excepción específica y se muestra un mensaje de error. En el segundo catch se maneja cualquier otra excepción que pudiera producirse (en este caso no sucederá, ya que no llegaría a ejecutarse), y se muestra un mensaje genérico.

Si no se produjera la primera excepción (por ejemplo mostramos `numeros[1]`), se produciría la segunda, que es de tipo `NullPointerException`, no entraría en el primer `catch` puesto que no es de ese tipo, pero si entraría en el segundo, que ya que `NullPointerException` hereda de `Exception`.

> 🤓 Ya lo veremos más adelante, pero TODAS las excepciones heredan de la clase base `Exception`. De ahí que si no estuviese el primer `catch`, **cualquier** excepción que se produzca (y que no tenga su propio `catch` específico), entraría en el segundo `catch`.

Declaración throws

Por otro lado, la declaración `throws` permite propagar una excepción hacia el método que la invocó, permitiendo a ese método manejar la excepción o propagarla hacia el siguiente. Con esta declaración, se puede tratar la excepción en un punto más general de la aplicación, en lugar de tener que manejarla en cada método que la pueda generar.

Para declarar que un método puede lanzar una excepción, se añade la palabra reservada `throws` seguido de la clase de la excepción.

```java
public void métodoQueLanzaLaExcepción() throws TipoExcepción {
    ...
}
```

Aquí un ejemplo de uso de la declaración `throws` en Java:

```java
public class Main {
    public static void main(String[] args) {
        try {
            divisiónPorCero();
        } catch (ArithmeticException e) {
            System.out.println("Ocurrió un error de aritmética: " +
e.getMessage());
        }
    }

    public static void divisiónPorCero() throws ArithmeticException {
        int x = 5;
        int y = 0;
        int resultado = x / y;
        System.out.println(resultado);
    }
}
```

En este ejemplo, el método `divisiónPorCero` lanza una `ArithmeticException` usando la declaración `throws`. Si se produce una excepción en el método, se propagará hasta el método `main`, donde se captura con un bloque `try` - `catch`.

La salida de este programa sería:

```
Ocurrió un error de aritmética: / by zero
```

> 💣 Con `throws` lo que hacemos es no tener que capturar la excepción, si no "lanzarla para atrás" para que ese método sea el que tenga que capturarla.
>
> Todo código en Java empieza con el método `main()`. En ese método se llamará a otro método y ese otro a otro. Si el último produce la excepción y la lanza con `throws` sin ocuparse de ella, la excepción va hacia atrás hasta que llega al método `main()`. Si en el mismo método `main()` usamos `throws`, al no haber sido la excepción tratada por ningún método de la cadena que lo invocó, se producirá la excepción y el programa terminará.
>
> Como una patata caliente que nadie quiere ocuparse, si la vamos lanzando y nadie se ocupa de ella (`throws`), explotará. Para ocuparse de ella, deberemos usar `try-catch`.

Excepciones personalizadas

También podemos crear nuestras propias excepciones para poder lanzarlas en el momento que queramos.

Crear una excepción personalizada en Java es un proceso que implica los siguientes pasos:

1. Crea una nueva clase que extienda de la clase `Exception` o una de sus subclases. La clase deberá sobrescribir el método `getMessage()` y/o el constructor que se ajuste a tus necesidades.
2. Crea la excepción como un objeto más y lánzala con la sentencia `throw` cuando quieras que se produzca.
3. Realiza el tratamiento de excepciones tal y como hemos visto anteriormente con una de las siguientes opciones:
    - O bien captura la excepción personalizada en un bloque `try-catch`.
    - O bien declara el `throws` en el método que la pueda producir para que otro la tenga que capturar.

> ⚠️ **¡OJO!** No confundir la sentencia `throws` con la sentencia `throw`. La primera **declara** que en un método puede producirse una excepción. Con `throw` estamos **lanzando** la excepción desde ése método al método que lo haya invocado.

Contexto: Tenemos una clase llamada `Animal` que tiene un atributo que determina el género del animal. También tiene un método llamado `cruzar()` que recibe un animal. Queremos que si el método `cruzar()` recibe un animal del mismo género se produzca una excepción. No queremos comprobar manualmente si son del mismo género y mostrar un mensaje. QUEREMOS UNA EXCEPCIÓN.

Vamos a ver los pasos detalladamente para crear una excepción personalizada:

1. Crea una clase que representará la excepción nueva. Debe heredar de la clase `Exception`. El constructor de la superclase recibe un `string` con el mensaje de la excepción. Ese mensaje se mostrará cuando se produzca la excepción, o bien en el método `.getMessage()`.

```
1  public class MismoGeneroException extends Exception {
2     public MismoGeneroException(String mensaje) {
3        super(mensaje);
4     }
5  }
```

> 🤦 **Capitán Obvio**: La clase `MismoGeneroException` sigue siendo una clase normal y corriente. Puede tener sus propios atributos, métodos y constructores si así lo necesitas.

2. Esta es nuestra clase `Animal`.
    1. En el método `cruzar()`, cuando comprobemos que los animales son del mismo género, debemos crear un objeto de la clase `MismoGeneroException`, pasándole a su constructor el mensaje que queremos que exista en la excepción.
    2. Debemos utilizar la sentencia la sentencia `throw` para lanzar la excepción personalizada. El método no seguiría ejecutándose (actúa como un `return`).

```
1  public class Animal {
2      private String genero;
3
```

```
    public Animal(String genero) {
        this.genero = genero;
    }

    //Getter y setters
    ...

    public void cruzar(Animal otroAnimal) throws MismoGeneroException {
        if (this.genero.equals(otroAnimal.getGenero())) {
            throw new MismoGeneroException("Los animales deben ser de diferente género");
        }
        // Aquí continúa el código para realizar la acción de cruzar animales
    }
}
```

3. Finalmente, en el trozo de código que llama al método `cruzar`, se deben manejar las excepciones con un bloque `try-catch` o declarar la excepción con `throws`.

```
...
Animal animal1 = new Animal("Macho");
Animal animal2 = new Animal("Hembra");
try {
   animal1.cruzar(animal2);
} catch (MismoGeneroException e) {
   System.out.println(e.getMessage());
}
...
```