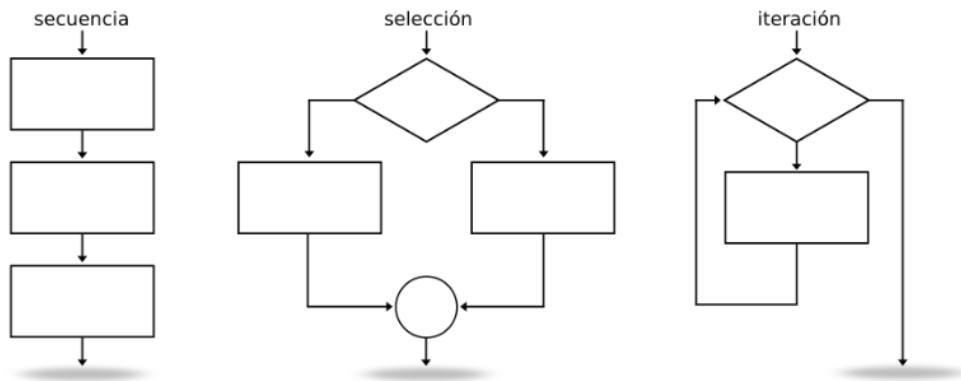


# UD03

## Anexo I - Excepciones



Este material está bajo una [Licencia Creative Commons Atribución-Compartir-Igual 4.0 Internacional](https://creativecommons.org/licenses/by-nc-sa/4.0/).  
Derivado a partir de material de David Martínez Peña (<https://github.com/martinezpenya>).

## 1. **Qué son las excepciones**

## 2. **Jerarquía de excepciones**

- 2. 1. Declaración de excepciones de un método
- 2. 2. [try/catch en excepciones verificadas](#)
- 2. 3. Creación de excepciones propias

## 3. **Fuentes de información**

# 1. Qué son las excepciones

Cuando un programa Java viola las restricciones semánticas del lenguaje (se produce un error) la máquina virtual Java comunica este hecho al programa **mediante una excepción**.

Muchas tipos de errores pueden provocar una excepción: un desbordamiento de memoria, un disco duro estropeado, un intento de dividir por cero o intentar acceder a un vector fuera de sus límites.

Cuando esto ocurre, la máquina virtual Java **crea un objeto de clase Exception**.

## Ejemplo 1:

```

1 public class PruebaExcepciones {
2
3     public static void main(String[] args) {
4         int numero1 = 5, numero2 = 0;
5
6         int resultado = numero1 / numero2;
7
8         System.out.println("El resultado es: " + resultado);
9     }
10 }
```

En el anterior ejemplo, Java no detecta un error en la línea de código 6 (aunque sabemos que no se podría realizar esta operación).

Mostrará la siguiente excepción (`ArithmeticException` del paquete **lang**) en línea de ejecución:

```

1 Exception in thread "main" java.lang.ArithmeticException: / by zero
2     at PruebaExcepciones.main(PruebaExcepciones.java:6)
```

Si accedemos a la información de la API de Java en la web, podemos comprobar que, en el paquete `java.lang`, encontramos, en su sección de excepciones, la excepción *ArithmeticException*; que hereda de la clase `Exception`.

The screenshot shows the Oracle Java API documentation for `java.lang.ArithmeticException`. The browser address bar displays `docs.oracle.com/javase/7/docs/api/`. The left sidebar lists the package hierarchy, with `java.lang` selected. The main content area shows the class `ArithmeticException`, its inheritance hierarchy (Object, Throwable, Exception, RuntimeException, ArithmeticException), implemented interfaces (Serializable), and a constructor summary table.

Constructor and Description
<code>ArithmeticException()</code> Constructs an <code>ArithmeticException</code> with no detail message.
<code>ArithmeticException(String s)</code> Constructs an <code>ArithmeticException</code> with the specified detail message.

Siguiendo con el ejemplo 1, ¿qué ocurriría si añadimos más líneas de código justo después de la línea de código que provoca una excepción aritmética?

```

1 public class PruebaExcepciones {
2
3     public static void main(String[] args) {
4         int numero1 = 5, numero2 = 0; int resultado = numero1 / numero2;
5
6         System.out.println("El resultado es: " + resultado);
7
8         System.out.println("Adiós"); // se añade más código ...
9     }
10 }
```

El siguiente código provocaría, exactamente, la misma excepción; y la ejecución pararía en el mismo punto, mostrando el mismo código que el ejemplo anterior. Además, la línea 8 ya no se mostraría:

```

1 | Exception in thread "main" java.lang.ArithmeticException: / by zero
2 |         at PruebaExcepciones.main(PruebaExcepciones.java:6)

```

**Ejemplo 2:** ¿Qué resultado mostrará el siguiente código si al ejecutarlo introducimos un número en forma de cadena?

```

1 | import java.util.Scanner;
2 |
3 | public class PruebaExcepciones {
4 |
5 |     public static void main(String[] args) {
6 |         Scanner entrada = new Scanner(System.in);
7 |
8 |         System.out.print("Introduce un número entero: ");
9 |         int numero = entrada.nextInt();
10 |
11 |         System.out.println(numero);
12 |     }
13 | }

```

```

1 | Introduce un número entero: quince

```

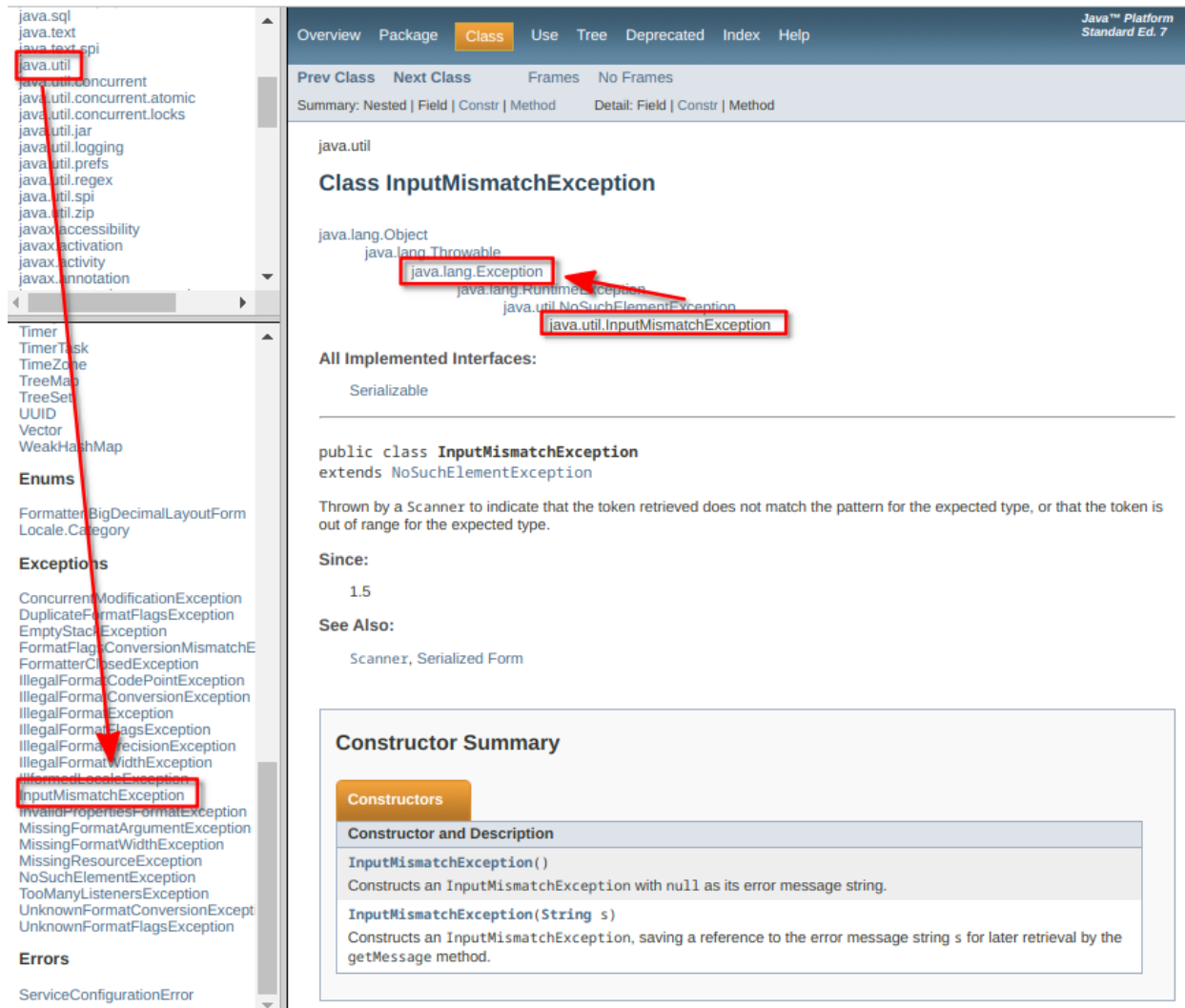
Mostrará la siguiente excepción (`InputMismatchException` del paquete `util`) en línea de ejecución:

```

1 | Exception in thread "main" java.util.InputMismatchException
2 | ...
3 |         at PruebaExcepciones.main(PruebaExcepciones.java:9)

```

Si accedemos a la información de la API de Java en la web, podemos comprobar que, en el paquete `java.util`, encontramos, en su sección de excepciones, la excepción `InputMismatchException`; que también hereda, por supuesto, de la clase `Exception`.



Overview Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

## Class InputMismatchException

java.lang.Object  
java.lang.Throwable  
**java.lang.Exception**  
java.lang.RuntimeException  
java.util.NoSuchElementException  
**java.util.InputMismatchException**

All Implemented Interfaces:  
Serializable

```
public class InputMismatchException
extends NoSuchElementException
```

Thrown by a Scanner to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type.

Since:  
1.5

See Also:  
Scanner, Serialized Form

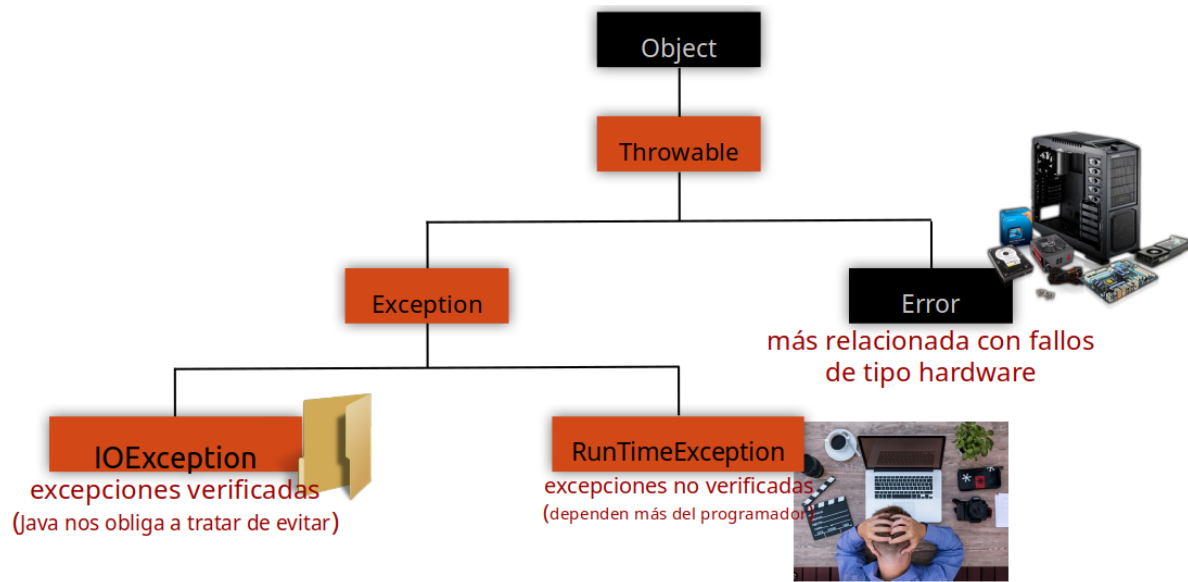
### Constructor Summary

Constructors

Constructor and Description
<b>InputMismatchException()</b> Constructs an InputMismatchException with null as its error message string.
<b>InputMismatchException(String s)</b> Constructs an InputMismatchException, saving a reference to the error message string s for later retrieval by the getMessage method.

El manejo de excepciones **va a permitir que el programa no se “frene”**, evadiendo los diferentes errores que encuentre para que, el código que existe después de la línea de error, se pueda ejecutar sin problemas.

## 2. Jerarquía de excepciones



En la jerarquía que ofrece Java para los objetos Exception (excepciones) se puede observar que Exception hereda de la clase Throwable que, a la vez y como todas las clases en Java, hereda de la clase Object.

A partir de la clase Throwable se tienen dos tipos de clases. Por una parte la clase Error en la que encontraríamos errores más comunes al ámbito hardware; y por otra la clase Exception (clase de la que hablaremos a continuación y de la que dependen más los errores software).

De la clase Exception heredan otros dos tipos de clases:

- La clase IOException: que tratará excepciones verificadas, y que Java nos obliga a tratar de evitar. No dependen directamente del programador (aunque sí se puede solucionar).

Tienen más que ver con las entradas-salidas de los programas. Un ejemplo podría ser que, por error, el usuario haya borrado una carpeta o fichero del que depende el propio programa.

- La clase RuntimeException: que tratará excepciones NO verificadas y que dependerán más del programador. Ejemplos de este tipo de errores serían dividir un número entero entre cero, recorrer un array con más posiciones que en principio se declaró el array o guardar un String dentro de un valor entero.

Estos tipos de errores Java no obliga a evadir estos errores. En cambio, los errores que generan objetos de la clase IOException sí que estaremos obligados a evitar.

**Ejemplo 3:** Las siguientes líneas de código muestran una **excepción verificada**. En estas, se desea leer un fichero debería encontrarse en una ruta determinada ("/home/abc/texto.txt") pero que no existe:

```
1 import java.io.BufferedReader;
```

```

2  import java.io.FileNotFoundException;
3  import java.io.FileReader;
4  import java.io.IOException;
5
6  public class PruebaExcepciones {
7
8      public static void main(String[] args) throws FileNotFoundException,
IOException {
9          // Excepciones verificadas (IOException)
10         BufferedReader bf = new BufferedReader(new
FileReader("/home/abc/texto.txt"));
11         String linea;
12         while ((linea = bf.readLine()) != null){
13             System.out.println(linea);
14         }
15     }
16 }

```

```

1  Exception in thread "main" java.io.FileNotFoundException: /home/abc/textos.txt
(No such file or directory)
2      at java.base/java.io.FileInputStream.open0(Native Method)
3      at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
4      at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
5      at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
6      at java.base/java.io.FileReader.<init>(FileReader.java:60)
7      at UD06.PruebaExcepciones.main(PruebaExcepciones.java:10)

```

Se observa que en la línea 10 de `PruebaExcepciones` mostrará el mensaje de excepción `FileNotFoundException`.

Cuando exista una excepción verificada, tendremos dos formas de tratamiento de ésta:

1. Declarar la excepción que se puede dar en el método.

En la misma declaración del método que puede provocar el error se introducen las excepciones:

```

1  public static void main(String[] args) throws FileNotFoundException,
IOException {

```

2. Capturarla con un try-catch.



```

1  ...
2  try {
3      while ((linea = bf.readLine()) != null){
4          System.out.println(linea);
5      }
6  } catch (IOException e) {
7      // TODO Auto-generated catch block
8      e.printStackTrace();
9  }
10 ...

```

**Ejemplo 4:** En el siguiente código:

```

1  import java.io.File;
2  import java.io.FileReader;
3
4  public class PruebaExcepciones {
5
6      public void leerArchivo(){
7          File archivo = new File("/home/abc/texto.txt");
8          FileReader fr = new FileReader(archivo); Unhandled exception type
FileNotFoundException
9      }
10
11     public static void main(String[] args) {
12     }
13 }

```

En la línea 8 aparece un mensaje (*Unhandled exception type FileNotFoundException*) en el que nos indica que no hemos tratado este tipo de excepción, del tipo `FileNotFoundException` que hereda de `IOException`:

```

import java.io.File;
import java.io.FileReader;

```

```

public class PruebaExcepciones {

```

```

    public void leerArchivo(){
        File archivo = new File("/home/abc/texto.txt");
        FileReader fr = new FileReader(archivo);
    }

```

Unhandled exception type FileNotFoundException

```

    public static void main(String[] args) {
    }
}

```

el siguiente error nos obliga a elegir una de estas dos opciones:  
- Add throws declaration → añadir declaración  
- Surround with try/catch → capturar excepción con este bloque

#### Quick Fix...

- ⚡ Add throws declaration
- ⚡ Surround with try-with-resources
- ⚡ Surround with try/catch

#### More Actions...

- ⚡ Add Javadoc for 'leerArchivo'
- ⚡ Add final modifier for 'fr'

Si, en Visual Studio Code, pulsamos en el icono azul (en otros IDEs como Eclipse también aparecen este tipo de advertencias) nos ofrece dos tipos de opciones para manejar el error/excepción:

- **Add throws declaration**, en la que se declara la excepción en la misma declaración del método y se importa la clase `java.io.FileNotFoundException`, o
- **Surround with try/catch**, en la que introduce un bloque de código que va a contener la excepción.

## 2.1. Declaración de excepciones de un método

Si elegimos la primera opción, declarar:

```

1  import java.io.File;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4
5  public class PruebaExcepciones {
6
7      public void leerArchivo()throws FileNotFoundException{
8          File archivo = new File("/home/abc/texto.txt");
9          FileReader fr = new FileReader(archivo);
10     }
11     public static void main(String[] args) {
12
13     }
14 }
```

### Recuerda:

Utilizaremos la declaración de excepciones cuando en dicho método no se quiere capturar el error, sino que dicho método se va a utilizar en otro método (en este otro ya se capturaría el error).

```

1  import java.io.File;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4
5  public class PruebaExcepciones {
6      public void leerArchivo()throws FileNotFoundException{
7          File archivo = new File("/home/abc/texto.txt");
8          FileReader fr = new FileReader(archivo);
9      }
10     public void leerArchivo2(){
11         leerArchivo();
12     }
13     public static void main(String[] args) {
14     }
15 }
```

## 2.2. try/catch en excepciones verificadas

Esta opción se utilizará para usar, desde otro método, una excepción que hayamos declarado. En este ejemplo:

```

1  import java.io.File;
2  import java.io.FileReader;
3  import java.io.FileNotFoundException;
4
5  public class PruebaExcepciones {
6      public static void leerArchivo()throws FileNotFoundException{
7          File archivo = new File("/home/abc/texto.txt");
8          FileReader fr = new FileReader(archivo);
9      }
10     public static void leerArchivo2(){
11         try {
12             leerArchivo();
13         } catch (FileNotFoundException ex){
14             JOptionPane.showMessageDialog(null, "archivo no encontrado");
15
16         } catch (IOException ex) {
17             JOptionPane.showMessageDialog(null, "ha ocurrido una excepción
18             verificada");
19         }
20     }
21     public static void main(String[] args) {
22         leerArchivo2();
23     }
24 }

```

El método *leerArchivo2()* usa el método *leerArchivo()*. Como *leerArchivo()* tiene una excepción en su declaración, en *leerArchivo2()* vamos a intentar capturarla (catch). Además, como tenemos dos tipos de excepciones, vamos a tener de capturar estas dos. Se da el caso que *FileNotFoundException* hereda de *IOException*; por lo que sería correcto capturar solo la excepción *IOException*. Pero es recomendable capturar los dos tipos; y además con el orden de: la clase más cercana.

## 2.3. Creación de excepciones propias

Si queremos crear nuestra propia excepción, deberá de heredar de *Exception* y crear dos constructores; uno vacío y otro que tendrá como parámetro un *String*.

**Ejemplo:**

```

1  static class LongitudMailErronea extends Exception {
2      public LongitudMailErronea(){}
3      public LongitudMailErronea(String msg_error){
4          super(msg_error);
5      }
6  }

```

Después podremos utilizar esta clase desde el `throws` del método que necesite controlar esta excepción:

```

1  static void examinaMail(String mail) throws LongitudMailErronea {
2      int arroba = 0;
3      boolean punto = false;
4
5      if (mail.length() <= 3){
6          throw new LongitudMailErronea();
7      } else {
8          for(int i=0; i < mail.length(); i++){
9              if (mail.charAt(i)=='@'){
10                 arroba++;
11             }
12             if (mail.charAt(i)=='.'){
13                 punto = true;
14             }
15         }
16         if (arroba==1 && punto==true){
17             System.out.println("Es correcto");
18         } else {
19             System.out.println("No es correcto");
20         }
21     }
22 }

```

Ahora, desde donde llamo a este método

```

1  import javax.swing.*;
2
3  public class CompruebaMail{
4      public static void main(String[] arg){
5          String elMail = JOptionPane.showInputDialog("Introduce mail:");
6
7          try{
8              examinaMail(elMail);
9          } catch (Exception e){
10             System.out.println("La dirección de email no es correcta.");
11             //e.printStackTrace();
12         }
13     }
14     ...
15 }

```

## 3. Fuentes de información

---

- [Wikipedia](#)
- [Udemy - Aprende Programación en Java \(de Básico a Avanzado\).](#) ]