EXCEPCIONES EN JAVA

Charly Cimino

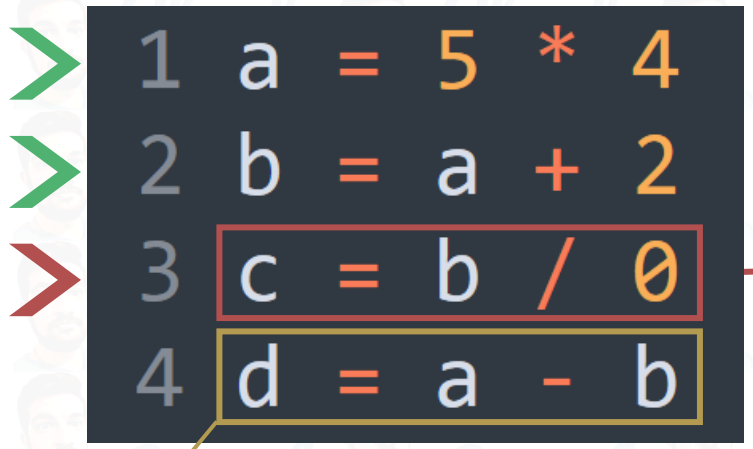
Excepciones en Java

Charly Cimino

Versión 2023.03.15

Excepción

Evento ocurrido al generarse un error en un programa durante el tiempo de ejecución.

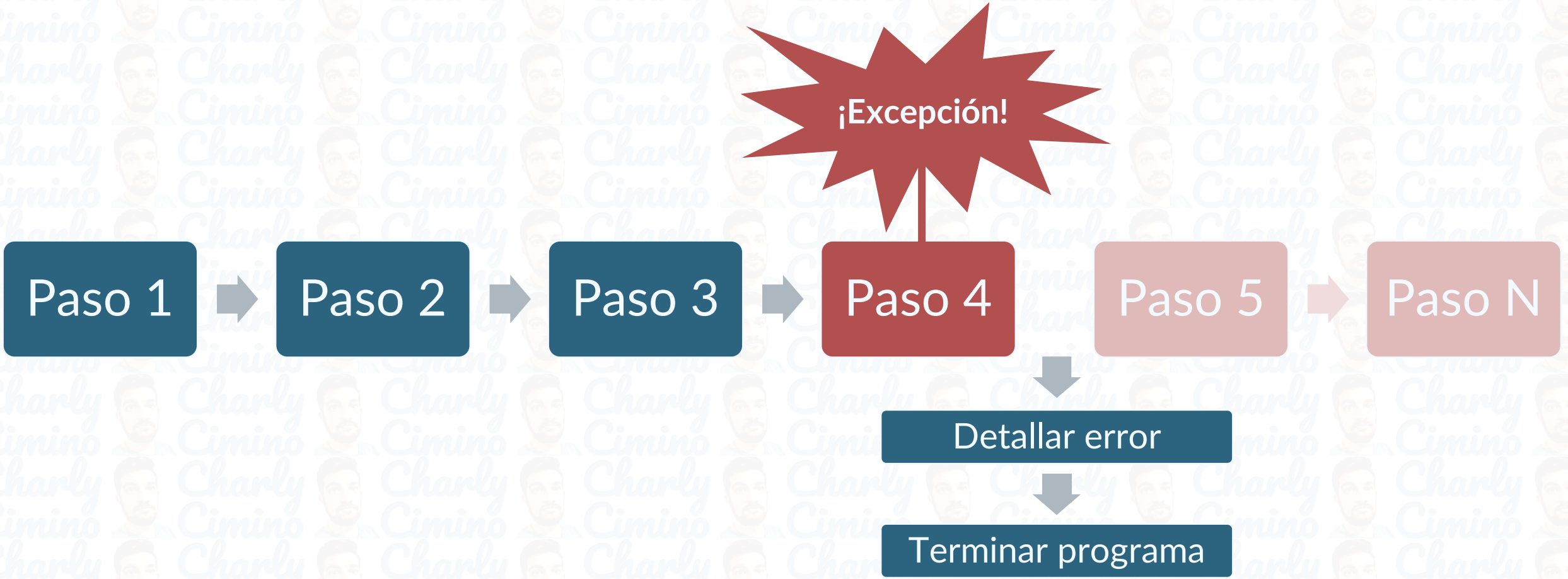


Falla en tiempo de ejecución: No se puede dividir por cero.

Nunca llega a ejecutarse

Las excepciones son bloqueantes

Cuando ocurre una excepción, se bloquea el normal flujo del programa, provocando en la mayoría de los casos que éste finalice prematuramente.



Las excepciones son bloqueantes

Cuando ocurre una excepción, se bloquea el normal flujo del programa, provocando en la mayoría de los casos que éste finalice prematuramente.

Si se lee un cero...

Nunca se ejecutan

```
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    y = new Scanner(System.in).nextInt();
    z = x / y;
    System.out.println("Resultado: " + z);
    System.out.println("Adiós!");
}
```

```
run:
Ingrese y: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at prueba.Prueba.main(Prueba.java:17)
```

Manejo de excepciones

Si se prevé que ocurra una excepción y ésta finalmente es lanzada, entonces puede capturarse y tratarse de una manera particular, mediante los bloques `try...catch...finally`.

```
public void algunMetodo() {
    /*
        Instrucciones sin posibles
        excepciones.
    */
    try {
        // Instrucción con posible excepción
        /* Instrucciones que dependen de
           la que posiblemente lance una
           excepción */
    } catch(UnTipoDeExcepcion ex) {
        /* Alguna lógica en caso de capturar
           una excepción de tipo 'UnTipoDeExcepcion' */
    } finally {
        /* Bloque opcional: se ejecuta en
           cualquiera de los casos */
    }
    /*
        Instrucciones sin posibles
        excepciones.
    */
}
```

Flujo del programa con manejo de excepciones

Si todas las instrucciones del bloque `try` tienen éxito, se salta directamente al `finally` (si hubiese) y luego se continúa con la normal ejecución del programa.

Si se lee entero (por ejemplo, un 5)...

Muestra 2

```
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    try {
        y = new Scanner(System.in).nextInt();
        z = x / y;
        System.out.println("Resultado: " + z);
    } catch (Exception ex) {
        System.out.println("Ocurrió un error");
    } finally {
        // OPCIONAL
        System.out.println("Adiós!");
    }
    System.out.println("Adiós, nuevamente");
}
```

```
run:
Ingrese y: 5
Resultado: 2
Adiós!
Adiós, nuevamente
BUILD SUCCESSFUL (total time: 2 seconds)
```

Flujo del programa con manejo de excepciones

La primera instrucción del bloque **`try`** que genere una excepción, hará que el flujo salte directamente al bloque **`catch`** que corresponda (veremos después que puede haber más de uno). A continuación, se salta directamente al **`finally`** (si hubiese) y luego se continúa con la normal ejecución del programa.

Si se lee algo que no represente un entero...

Se saltean

Se captura y se trata la excepción

```
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    try {
        y = new Scanner(System.in).nextInt();
        z = x / y;
        System.out.println("Resultado: " + z);
    } catch (Exception ex) {
        System.out.println("Ocurrió un error");
    } finally {
        // OPCIONAL
        System.out.println("Adiós!");
    }
    System.out.println("Adiós, nuevamente");
}
```

```
run:
Ingrese y: hola
Ocurrió un error
Adiós!
Adiós, nuevamente
BUILD SUCCESSFUL (total time: 4 seconds)
```

Flujo del programa con manejo de excepciones

La primera instrucción del bloque **`try`** que genere una excepción, hará que el flujo salte directamente al bloque **`catch`** que corresponda (veremos después que puede haber más de uno). A continuación, se salta directamente al **`finally`** (si hubiese) y luego se continúa con la normal ejecución del programa.

Si se lee un cero...

Falla la operación

Se saltea

Se captura y se trata la excepción

```
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    try {
        y = new Scanner(System.in).nextInt();
        z = x / y;
        System.out.println("Resultado: " + z);
    } catch (Exception ex) {
        System.out.println("Ocurrió un error");
    } finally {
        // OPCIONAL
        System.out.println("Adiós!");
    }
    System.out.println("Adiós, nuevamente");
}
```

```
run:
Ingrese y: 0
Ocurrió un error
Adiós!
Adiós, nuevamente
BUILD SUCCESSFUL (total time: 2 seconds)
```

Excepciones en los lenguajes de programación

En algunos lenguajes no existe un mecanismo de tratamiento de excepciones.

En otros lenguajes, el mecanismo de excepciones es bastante sencillo...

```
function probar(x) {
    if (x == 0)
        throw "No puede ser cero";
}
```

```
try {
    probar(0);
} catch (e) {
    console.log(e);
}
```

```
No puede ser cero    VM121:4
```

Manejo de excepciones en JavaScript

Java cuenta con un robusto esquema de excepciones que se detallará a continuación...

Las excepciones en Java son objetos

Lo que se captura a través del bloque **`catch`** es un objeto que extiende de la clase **`Throwable`**. Se lo puede consultar para obtener información detallada sobre el error.

```java
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    try {
        y = new Scanner(System.in).nextInt();
        z = x / y;
        System.out.println("Resultado: " + z);
    } catch (Exception ex) {
        System.out.println("Error: " + ex.getMessage());
        ex.printStackTrace(System.out);
    }
    System.out.println("Adiós");
}
```

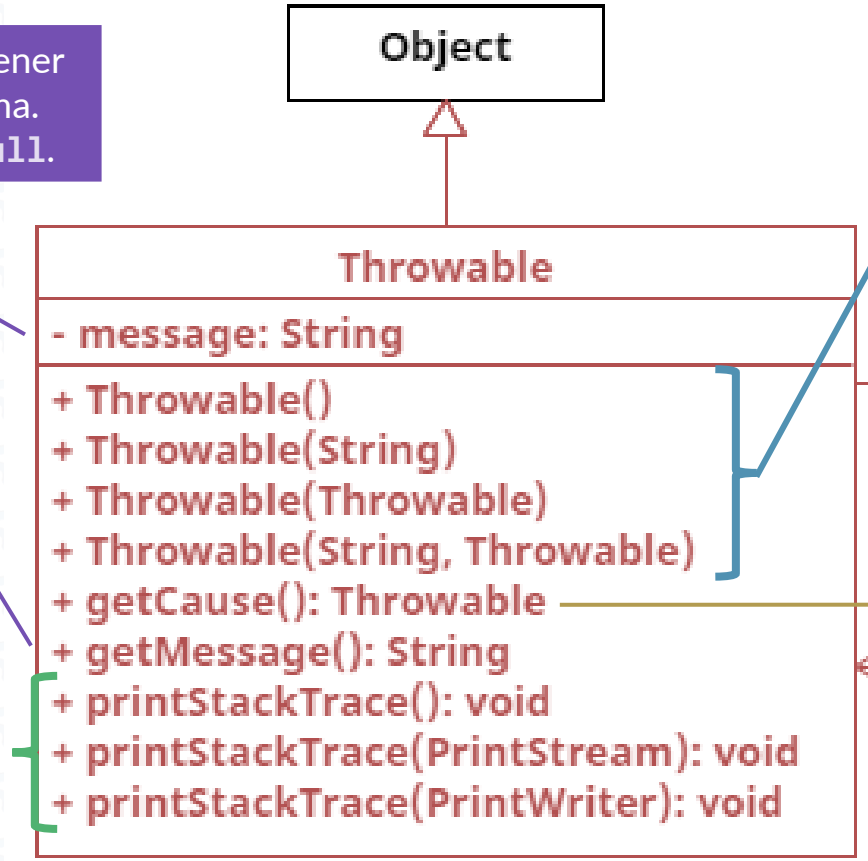
```
run:
Ingrese y: hola
Error: null
java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at prueba.Prueba.main(Prueba.java:18)
Adiós
BUILD SUCCESSFUL (total time: 2 seconds)
```

```
run:
Ingrese y: 0
Error: / by zero
java.lang.ArithmeticException: / by zero
        at prueba.Prueba.main(Prueba.java:19)
Adiós
BUILD SUCCESSFUL (total time: 2 seconds)
```

La clase `Throwable`

`Throwable` es la superclase de todos los errores y excepciones en el lenguaje Java. Solo las instancias de esta clase (o de sus subclases) pueden ser lanzados por la JVM o manualmente con la sentencia **`throw`**.

Todo objeto arrojado debería contener un mensaje que detalle el problema. Si no hay mensaje, su valor será `null`.



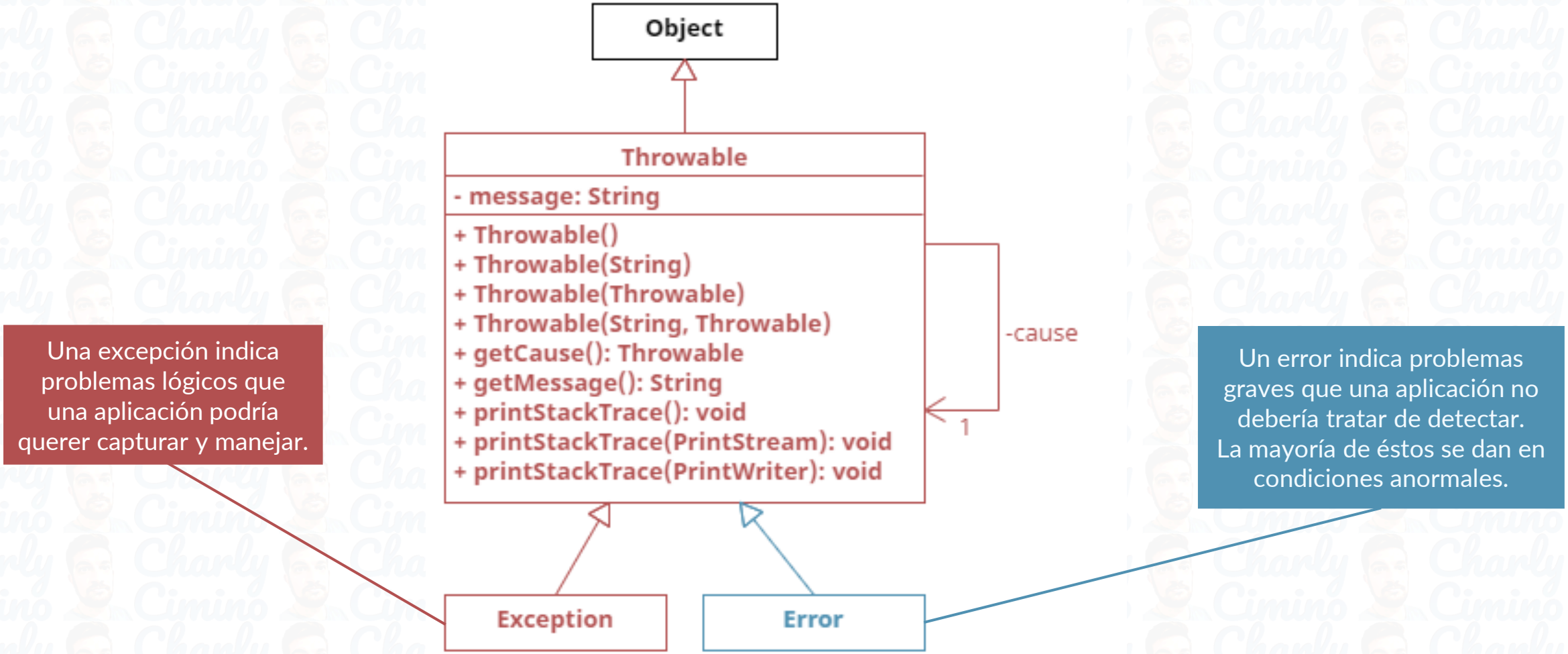
Por convención, **`Throwable`** y sus subclases tienen dos constructores, uno que no toma parámetros y otro con uno de tipo **`String`** que representa un mensaje detallado. Las subclases que probablemente tengan una causa asociada deberían tener dos constructores más: uno que tome un **`Throwable`** (la causa) y otro que tome un **`String`** (el mensaje detallado) y un **`Throwable`** (la causa).

Opcionalmente, se puede tener referencia a una causa (otro **`Throwable`**), para permitir un encadenamiento de excepciones que describa más y mejor el problema.

Permiten imprimir el *call stack* en la consola (sin argumentos) o en un recurso externo (por ejemplo, un archivo de texto).

Diferencia entre excepciones y errores

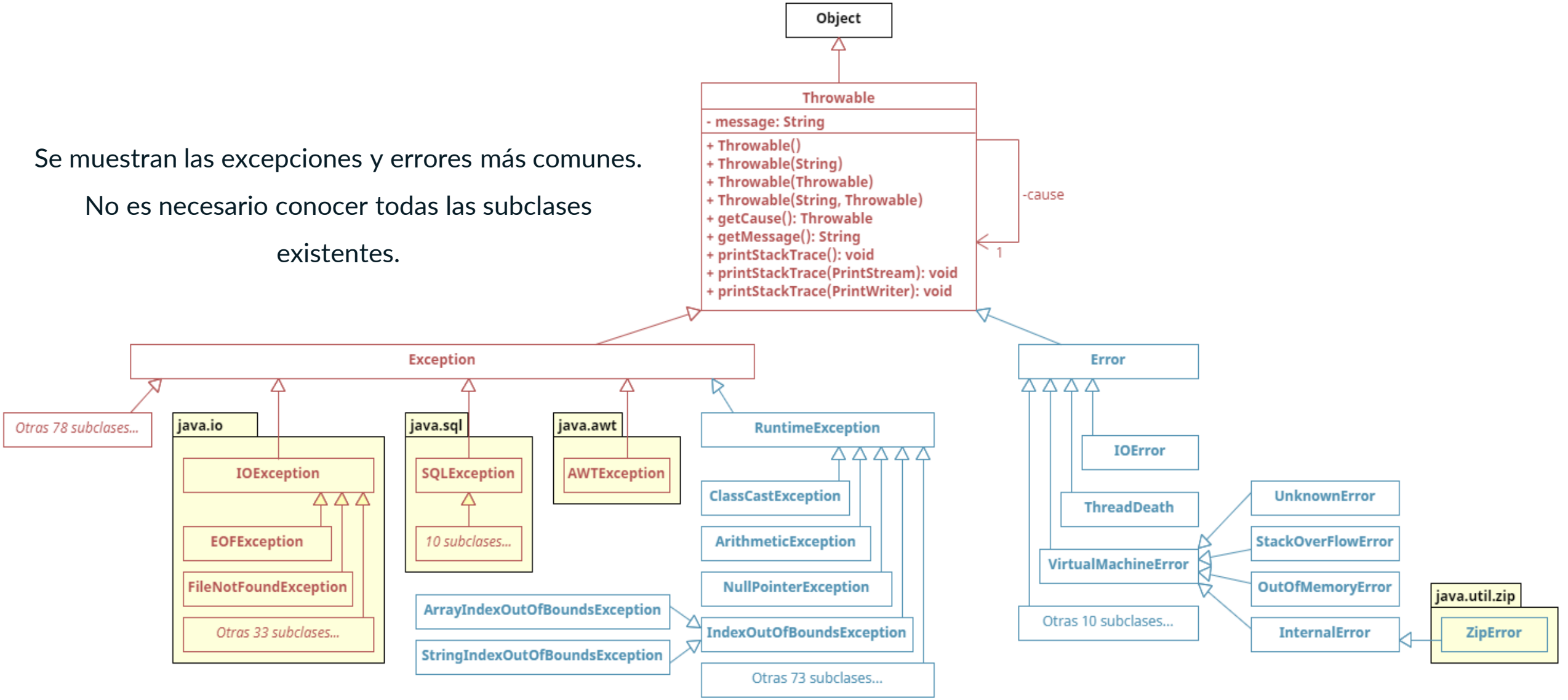
Las dos subclases de `Throwable` son `Exception` y `Error`.



Jerarquía 'completa' de errores y excepciones

Se muestran las excepciones y errores más comunes.

No es necesario conocer todas las subclases existentes.



Multicatch

En caso de querer discriminar ciertos tipos de excepciones para tomar acciones diferentes, se puede usar más de un bloque **`catch`**.

Si se lee algo que no represente un entero...

Se saltean

Se captura y se trata la excepción

```
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    try {
        y = new Scanner(System.in).nextInt();
        z = x / y;
        System.out.println("Resultado: " + z);
    } catch (InputMismatchException ex) {
        System.out.println("No es un entero!");
    } catch (ArithmeticException ex) {
        System.out.println("No se puede dividir por cero...");
    } catch (Exception ex) {
        System.out.println("Error desconocido");
    }
    System.out.println("Adiós");
}
```

```
run:
Ingrese y: hola
No es un entero!
Adiós
BUILD SUCCESSFUL (total time: 2 seconds)
```

Multicatch

En caso de querer discriminar ciertos tipos de excepciones para tomar acciones diferentes, se puede usar más de un bloque **`catch`**.

Si se lee un cero...

Falla la operación

Se saltea

Se captura y se trata la excepción

```
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    try {
        y = new Scanner(System.in).nextInt();
        z = x / y;
        System.out.println("Resultado: " + z);
    } catch (InputMismatchException ex) {
        System.out.println("No es un entero!");
    } catch (ArithmeticException ex) {
        System.out.println("No se puede dividir por cero...");
    } catch (Exception ex) {
        System.out.println("Error desconocido");
    }
    System.out.println("Adiós");
}
```

```
run:
Ingrese y: 0
No se puede dividir por cero...
Adiós
BUILD SUCCESSFUL (total time: 2 seconds)
```

Multicatch

En caso de querer discriminar ciertos tipos de excepciones para tomar acciones diferentes, se puede usar más de un bloque `catch`.

```java
public static void main(String[] args) {
    int x = 10, y, z;
    System.out.print("Ingrese y: ");
    try {
        y = new Scanner(System.in).nextInt();
        z = x / y;
        System.out.println("Resultado: " + z);
    } catch (InputMismatchException ex) {
        System.out.println("No es un entero!");
    } catch (ArithmeticException ex) {
        System.out.println("No se puede dividir por cero...");
    } catch (Exception ex) {
        System.out.println("Error desconocido");
    }
    System.out.println("Adiós");
}
```

Se debe intentar capturar primero las excepciones más específicas y luego las más generales, de acuerdo con la jerarquía de excepciones de Java.

¿Quién lanza las excepciones?

Hay excepciones cuya lógica de aparición ya está programada en las clases incluidas en Java o compiladas en la JVM.

```
public static void main(String[] args) {
    int z = 8 / 0;
}
```

```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at prueba.Prueba.main(Prueba.java:15)
```

```
public static void main(String[] args) {
    int z = Math.incrementExact(2147483647);
}
```

```
run:
Exception in thread "main" java.lang.ArithmeticException: integer overflow
        at java.base/java.lang.Math.incrementExact(Math.java:1023)
        at prueba.Prueba.main(Prueba.java:15)
```

Vayamos a espiar esto...

Dónde se lanza la excepción...

```
run:
Exception in thread "main" java.lang.ArithmeticException: integer overflow
        at java.base/java.lang.Math.incrementExact(Math.java:1023)
        at prueba.Prueba.main(Prueba.java:12)
```

3

2

1

```
10  public class Prueba {
11      public static void main(String[] args) {
12          int x = Math.incrementExact(2147483647);
13          System.out.println(x); // No se ejecuta
14      }
15  }
```

La línea 12 del método **main** de la clase **Prueba** provoca una excepción, pero ésta viene de un nivel más profundo. Vayamos a espiar la clase **Math**...

La línea 1023 del método **incrementExact** de la clase **Math** **arroja** (*throw*) una excepción cuando el parámetro coincide con el máximo valor posible para un **int**.

```
1021    public static int incrementExact(int a) {
1022        if (a == Integer.MAX_VALUE) {
1023            throw new ArithmeticException("integer overflow");
1024        }
1025
1026        return a + 1;
1027    }
```

2

3

En este caso, la excepción es una instancia de la clase **ArithmeticException**, que representa un error matemático, cuyo mensaje es ***"Desbordamiento de entero"***

Cómo lanzar una excepción manualmente

Como se ha visto, para lanzar una excepción, basta con colocar la sentencia `throw` seguida de una instancia de una clase que extienda de `Throwable`.

```
throw new AlgunaExcepcion("Un mensaje que detalle el error");
```

Calculadora.java

```
public int factorial (int num) {
    if (num < 0) {
        throw new ArithmeticException("No se puede calcular el
                                       factorial de un negativo");
    }
    if (num > 12) {
        throw new IllegalArgumentException("Por limitaciones técnicas, no
                                            se puede calcular el factorial
                                            de un entero mayor que 12");
    }
    int fact = 1;
    for (int i = 2; i <= num; i++) {
        fact = fact * i;
    }
    return fact;
}
```

No hace falta un **else**, pues arrojar una excepción rompe el normal flujo del programa (es como un **break** o **return**).

Cómo lanzar una excepción manualmente

Calculadora.java

```
public int factorial (int num) {
    if (num < 0) {
        throw new ArithmeticException("No se puede calcular el
                                factorial de un negativo");
    }
    if (num > 12) {
        throw new IllegalArgumentException("Por limitaciones técnicas, no
                                    se puede calcular el factorial
                                    de un entero mayor que 12");
    }
    int fact = 1;
    for (int i = 2; i <= num; i++) {
        fact = fact * i;
    }
    return fact;
}
```

Principal.java

```
public static void main(String[] args) {
    Calculadora c = new Calculadora();
    System.out.println(c.factorial(-1));
}
```

```
run:
Exception in thread "main" java.lang.ArithmeticException: No se puede calcular el factorial de un negativo
        at prueba.Calculadora.factorial(Calculadora.java:13)
        at prueba.Prueba.main(Prueba.java:15)
```

Cómo leer el *call stack*

Cuando ocurre una excepción, por defecto se imprime en la consola el *call stack* (pila de llamadas) hasta el momento en que ocurrió un fallo.
No es más que el camino que tomó el flujo del programa hasta encontrarse con un problema.

```
10  public class Persona {
11      private int edad;
12
13      public Persona(int edad) {
14  2       setEdad(edad);
15      }
16
17      private void setEdad(int edad) {
18  3       checkEsNegativo(edad);
19          this.edad = edad;
20      }
21
22      private void checkEsNegativo(int x) {
23          if (x < 0)
24  4           throw new IllegalArgumentException("No puede ser negativo");
25      }
26  }
```

```
12  public class Prueba {
13      public static void main(String[] args) {
14  1       Persona p = new Persona(-1);
15      }
16  }
```

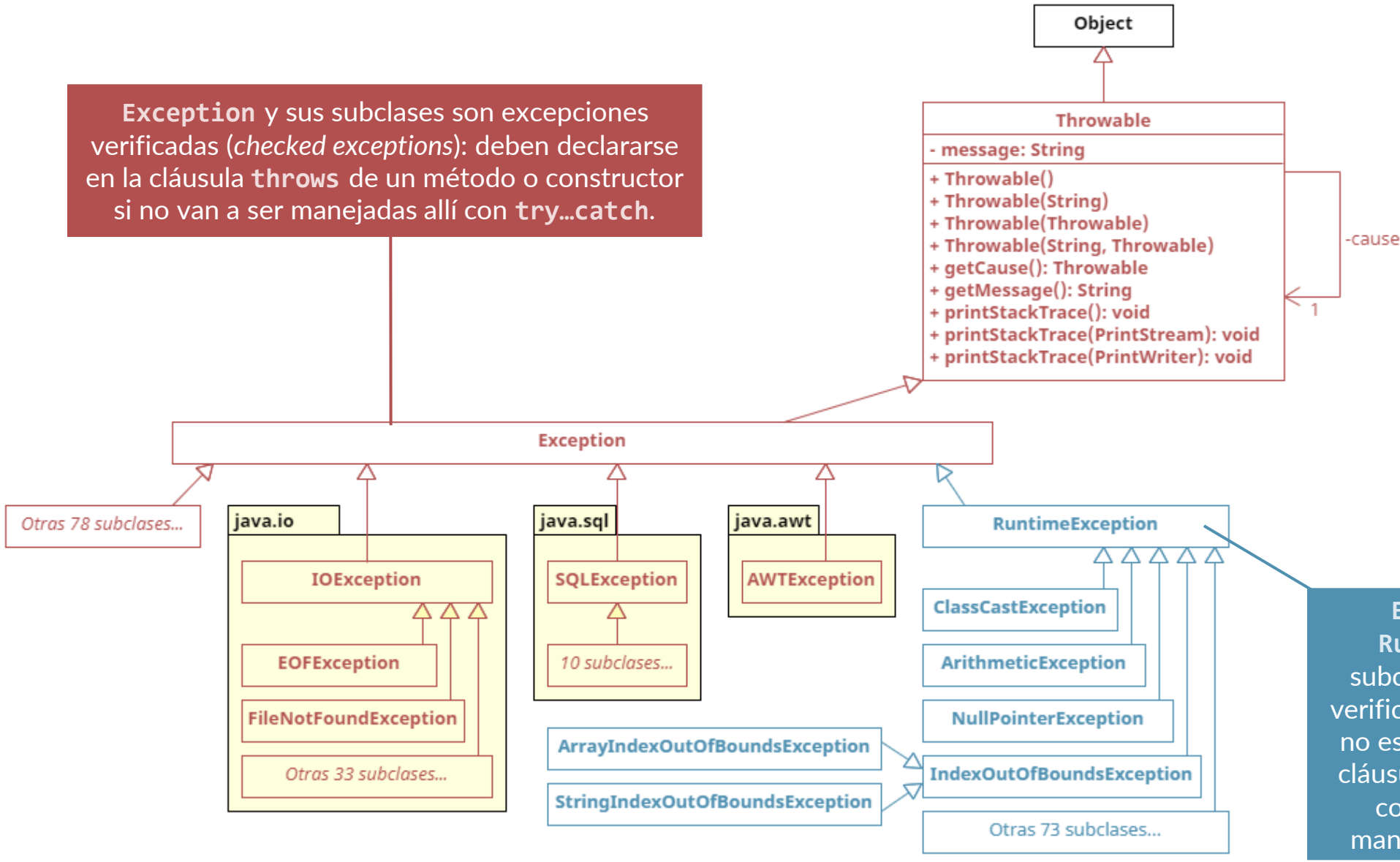
```
run:
Exception in thread "main" java.lang.IllegalArgumentException: No puede ser negativo
    at prueba.Persona.checkEsNegativo(Persona.java:24)  4
    at prueba.Persona.setEdad(Persona.java:18)          3
    at prueba.Persona.<init>(Persona.java:14)           2
    at prueba.Prueba.main(Prueba.java:14)               1
```

La pila de llamadas del programa hasta fallar.

Recordá leerlo de abajo hacia arriba, pues es una pila (*stack*): Primer método llamado, último método finalizado.

Excepciones *checked* vs. *unchecked*

`Exception` y sus subclases son excepciones verificadas (*checked exceptions*): deben declararse en la cláusula **`throws`** de un método o constructor si no van a ser manejadas allí con **`try…catch`**.



`Error` y sus subclases y **`RuntimeException`** y sus subclases son excepciones NO verificadas (*unchecked exceptions*): no es necesario declararlos en la cláusula **`throws`** de un método o constructor si no van a ser manejadas allí con **`try…catch`**.

Excepciones *checked* vs. *unchecked*

Calculadora.java

```
public int factorial (int num) throws ArithmeticException {
    if (num < 0) {
        throw new ArithmeticException("No se puede calcular el
                                       factorial de un negativo");
    }
    int fact = 1;
    for (int i = 2; i <= num; i++) {
        fact = fact * i;
    }
    return fact;
}
```

No se suele colocar para las *unchecked exceptions*

`ArithmeticException` es una subclase de **`RuntimeException`**, por lo que es una *unchecked exception*: el programador **puede opcionalmente** manejarla con **`try…catch`** o simplemente dejar que se propague sin más (en caso de ocurrir).

VisorDeTXT.java

```
public void mostrarTxt(String ruta) throws FileNotFoundException, IOException {
    File elArchivo = new File(ruta);
    BufferedReader br = new BufferedReader(new FileReader(elArchivo));
    System.out.println( br.readLine());
}
```

Las *checked exceptions* son más estrictas: el programador **debe obligatoriamente** manejarlas con **`try…catch`** o dejar que se propaguen, pero declarando cada una en la firma del método con la sentencia **`throws`**.

El método **`readLine`** podría arrojar una **`IOException`**, que es una subclase de **`Exception`** (pero no de **`RuntimeException`**), por lo que es una *checked exception*.

El constructor de **`FileReader`** podría arrojar una **`FileNotFoundException`**, que es una subclase de **`IOException`**, por lo que es una *checked exception*.

FIN DE LA PRESENTACIÓN

Encontrá más como estas en mi sitio web.