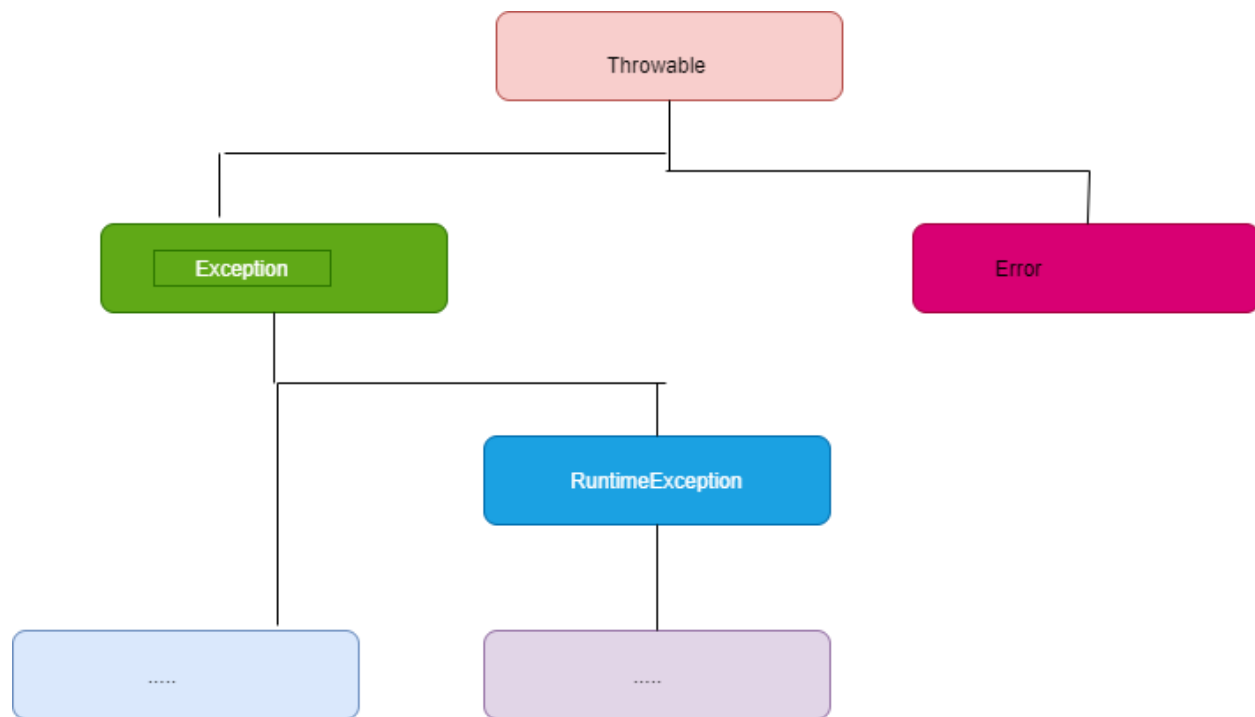


EXCEPCIONES EN JAVA

Las excepciones son un mecanismo muy importante en Java, que nos ayuda a actuar cuando tenemos un problema en la ejecución de nuestro programa. Puede ser por ejemplo tratar de acceder a un índice de un array que no existe, un problema al tratar de escribir en un archivo, tratar de cargar un dato en un tipo diferente, ejemplo un String en un entero. Entendemos por excepción a un evento que ocurre durante la ejecución de un programa e interrumpe el flujo normal de sus instrucciones. Cuando se crea un error en un método, este crea un objeto del tipo Exception, que contiene información sobre el error, por ejemplo el tipo de Exception o el estado del programa donde ocurrió. Este objeto es entregado al sistema de ejecución conocido en inglés como run time system, a este proceso se le denomina lanzar una excepción.



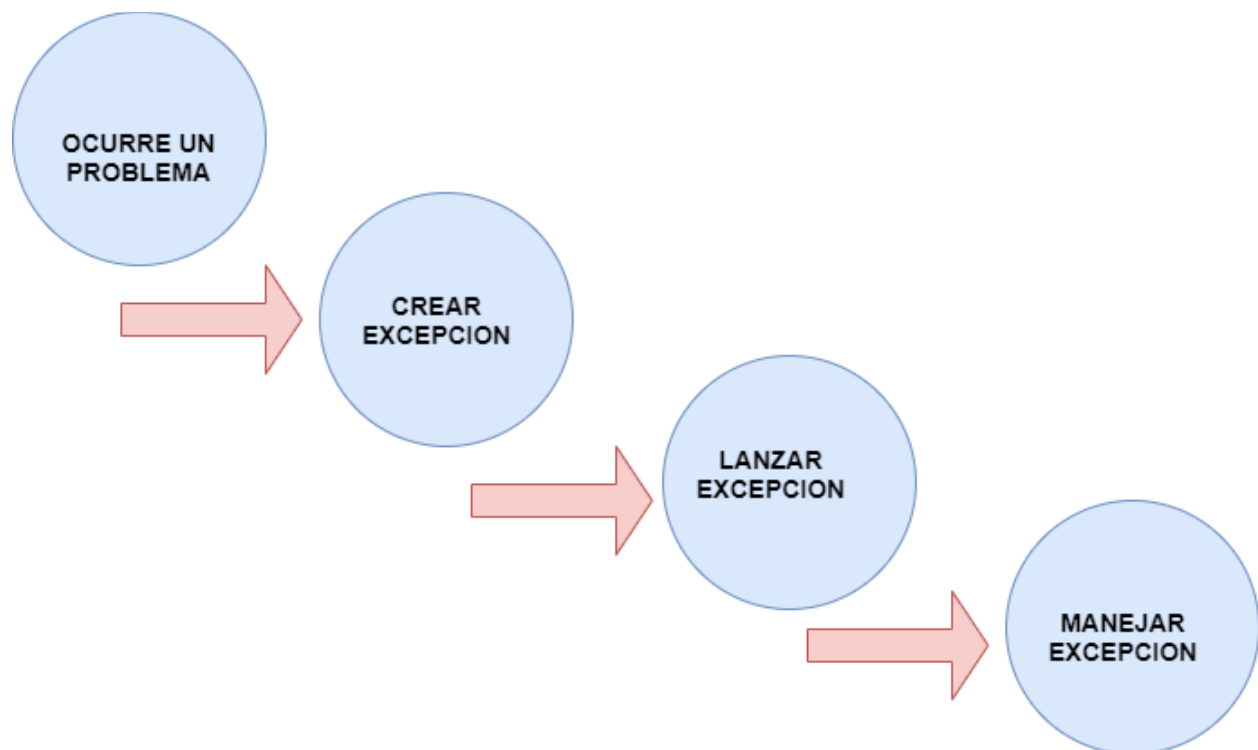
La Clase Exception de java extiende (hereda) de clase Throwable. La clase Throwable solo tiene 2 subclases, la clase Error, que es cuando ocurre un error en JVM de java, normalmente estos errores no podemos prevenirlos ni anticiparnos, ellos ocurren, ejemplo la memoria esta llena. La clase Exception tiene 83 clases que extienden de ella, por lo que hay 83 tipos de excepciones clasificadas e implementadas por java, de estas excepciones las que extienden de RuntimeException son debidas a “bugs” o errores, como errores lógicos o un uso inapropiado de una API. De acuerdo a esto java clasifica los errores que pueden ocurrir en un programa en dos:

Errores No Comprobados
Errores Comprobados

Los no comprobados son los que pertenecen a la clase Error y los que derivan de la clase RuntimeException, estas no se comprueban en tiempo de compilación.

Los comprobados, son el resto, y estos se comprueban en tiempo de compilación, en el caso del ide Eclipse estos errores los marca al escribir el código, los anteriores no.

El manejo de excepciones en Java es un truco simple y poderoso para manejar los errores de tiempo de ejecución. La excepción es un objeto que se lanza en tiempo de ejecución. Nos frena o detiene el flujo normal del programa. Después del manejo de excepciones, puede mantener el flujo normal de su programa.



Ventaja del manejo de excepciones

La excepción nos frena y para el flujo normal del programa. Por eso necesitamos manipulación. La principal ventaja del manejo de excepciones es mantener el flujo normal del programa.

Suponga que hay un programa como el siguiente.

```
public class Calculos {  
  
    public static void main(String[] args) {  
        // Ejemplos Java - Tema Excepciones  
        int num1=20;  
        int num2=30;  
        int resultado =(num1/0)*num2;  
        System.out.println(num1);  
        System.out.println(num2);  
        System.out.println(resultado);  
    }  
}
```

Se produce una excepción cuando se intenta dividir por cero un entero en este programa. Debido a esta excepción, obtendrá un error y detención del programa como se muestra a continuación,

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at excepciones.Calculos.main(Calculos.java:9)
```

También puede ver que el resto del código no se ejecutará. Pero si manejamos excepciones, se ejecutará el resto del código. Es por eso que deberíamos usar el manejo de excepciones, y también imagínense el usuario que les pase este tipo de problemas no es nada agradable que ocurran estas cosas en nuestros programas, más si lo podemos manejar.

Tipo de excepciones de Java

- 1) **Excepción comprobada**
- 2) **Excepción sin comprobar**
- 3) **Error**

El error también se considera la excepción sin comprobar. Entendamos la diferencia entre estos tres tipos.

El manejo de excepciones se usa principalmente para manejar excepciones no comprobadas. Si se produjo alguna excepción no comprobada en el programa, como `ArrayIndexOutOfBoundsException`, es culpa del programador, ya que no verificó el programa antes de usarlo. Además, si ocurre algún error (de la clase `Error`), está fuera del control del programador.

Diferencia entre excepción marcada, excepción no marcada y error

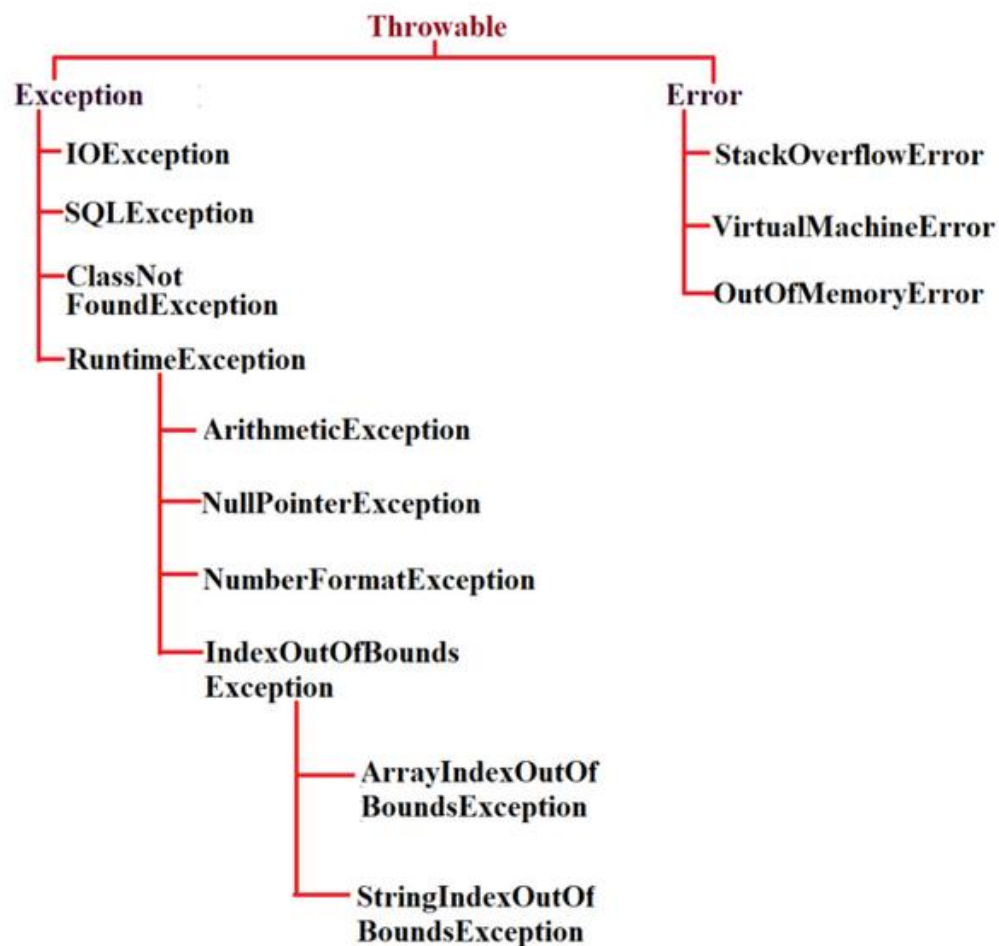
Excepción comprobada: las clases `Throwable` heredadas directamente, excepto la excepción de `RuntimeException` y `Error`, se denominan **excepciones comprobadas**. Las excepciones comprobadas se controlan en tiempo de compilación. (Por ejemplo: - `IOException`, `SQLException`,

ClassNotFoundException). En estos casos el compilador te obliga que debes poner o throws en el cabecal de la función o tratarlas con try/catch.

Excepción no comprobadas: las clases que heredan la excepción RuntimeException se denominan excepciones no comprobadas. Las excepciones no comprobadas se dan en tiempo de ejecución. (Por ejemplo: -ArithmeticException, NullPointerException, NumberFormatException, etc.). En estos casos el compilador no te obliga a tratarlas, pero es bueno tratarlas para que el usuario no tenga problemas al momento de que se puedan producir durante la ejecución del programa.

Error: el error es irrecuperable (por ejemplo: - StackOverflowError, OutOfMemoryError), esos son imposibles de predecir o controlar de antemano.

Jerarquía de clases de excepción de Java



Palabras clave en el manejo de excepciones

Hay 5 palabras clave en Manejo de excepciones. Estas son **try**, **catch**, **finally**, **throw** y **throws**.

Consideremos cada una de ellas a continuación.

Bloque try

Usamos la palabra clave **"try"**, para especificar donde o en qué lugar colocar el bloque de la excepción. Debe usarse dentro del método y debe ir seguido de la palabra clave **"catch"** o la palabra clave **"finally"**.

Sintaxis del bloque try-catch

```
try {  
    // (aca va codigo que puede lanzar una excepcion)  
} catch (Exception nombre referencia) {  
  
}  
}
```

Sintaxis del bloque try-finally

```
try {  
    // (aca va codigo que puede lanzar una excepcion)  
} finally {  
  
}
```

El resto del código del bloque no funcionará si se produce una excepción en la instrucción del bloque try. Por lo tanto, tenga en cuenta que el código, que no es una excepción, no debe mantenerse en un bloque try.

Bloque catch

Usamos el bloque **catch** para manejar la excepción. En el bloque catch dentro del parámetro debe declarar el tipo de excepción. Debe ser la excepción de la clase principal o el tipo de excepción generado. El bloque catch solo se usa después de un bloque **try** y podemos usar cualquier número de bloques de **catch** en un solo bloque **try**.

Consideremos un ejemplo en el bloque try-catch

Recuerdan el ejemplo visto anteriormente al principio:

```
public class Calculos {  
    public static void main(String[] args) {  
        // Ejemplos Java - Tema Excepciones  
        int num1=20;  
        int num2=30;  
        int resultado =(num1/0)*num2;  
        System.out.println(num1);  
        System.out.println(num2);  
        System.out.println(resultado);  
    }  
}
```

Aquí teníamos un error que nos decía : “ Exception in thread “main” java.lang.ArithmeticException: / by zero at Calculos.main(Calculos.java:9)”.

Veamos cómo podemos manejar esta excepción.

```
5 public static void main(String[] args) {  
6     // Ejemplos Java - Tema Excepciones  
7     int num1=20;  
8     int num2=30;  
9     int resultado=0;  
10    try {  
11        resultado =(num1/0)*num2;  
12        System.out.println("resto del codigo debajo del bloque try");  
13    } catch(ArithmeticException e) {  
14        System.out.println("resto del codigo debajo del bloque catch");  
15        System.out.println(e);  
16        System.out.println(num1);  
17        System.out.println(num2);  
18        System.out.println(resultado);  
19    }  
20  
21  
22  
23  
24  
25 }
```

Y la salida por consola de ejecutar este programa java es:

```
resto del codigo debajo del bloque catch  
java.lang.ArithmeticException: / by zero  
20  
30  
0
```

La sentencia de impresión `System.out.println` de la **línea 12** que está debajo del bloque `try`, no se verá en la salida. Porque en la **línea 11**, se produjo una excepción. Así que el resto del bloque `try` no se ejecutará.

Aquí usamos el tipo de excepción generado para manejar la excepción. Y también podemos usar la excepción de clase principal fácilmente agregando la palabra "Excepción" en lugar de la palabra "ArithmeticException" en la línea 13.

En este caso resolvemos la excepción en el bloque `catch`.

Si hay algún código que produzca un error de excepción en el bloque `catch`, el bloque `catch` no manejará la excepción.

Veremos un ejemplo de esto:

```
3 public class Calculos {
4
5     public static void main(String[] args) {
6         // Ejemplos Java - Tema Excepciones
7         int num1=20;
8         int num2=30;
9         int resultado=0;
10        try {
11            resultado =(num1/0)*num2;
12            System.out.println("resto del codigo debajo del bloque try");
13        } catch(ArithmeticException e) {
14            resultado = (num1/0); // lanzar una excepción
15
16            System.out.println("resto del codigo debajo del bloque catch");
```

Tenemos el siguiente mensaje en la consola:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at excepciones.Calculos.main(Calculos.java:14)
```

Y también si usa un tipo de excepción diferente para manejar la excepción generada, no manejará la excepción. Veremos el caso en el siguiente ejemplo:

```
3 public class Calculos {
4
5     public static void main(String[] args) {
6         // Ejemplos Java - Tema Excepciones
7         int num1=20;
8         int num2=30;
9         int resultado=0;
10        try {
11            resultado =(num1/0)*num2;
12            System.out.println("resto del codigo debajo del bloque try");
13        } catch(NullPointerException e) {
14            System.out.println("resto del codigo debajo del bloque catch");
15            System.out.println(e);
16            System.out.println(num1);
17            System.out.println(num2);
18            System.out.println(resultado);
19        }
20    }
```

Tenemos la siguiente salida por la consola:


```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at excepciones.Calculos.main(Calculos.java:11)
```

MULTIPLES BLOQUES CATCH

Un solo bloque try puede tener más de un bloque catch y cada bloque catch debe tener un manejador de excepciones diferente.

Puede haber varios bloques de captura, pero a la vez solo se produce una excepción y solo se ejecuta un bloque de captura.

Todos los bloques de captura bajo un bloque de intento deben ordenarse del más específico al más general.

Veamos un ejemplo,

```
3 public class MultiplesBolquesCatch {
4
5     public static void main(String[] args) {
6         // Multiples bloques catch
7         try {
8             String nombre=null;
9             int vector[] = new int[10];
10
11             System.out.println(nombre.length());
12             System.out.println(vector[20]);
13
14         } catch (NullPointerException e) {
15             System.out.println("Capturamos el error de Null Pointer Exception");
16         }
17         catch (ArrayIndexOutOfBoundsException e) {
18             System.out.println("Capturamos el error Indice del Array");
19         }
20         catch (Exception e) {
21             System.out.println("Excepcion Principal (Padre)");
22         }
23         System.out.println("Resto del codigo debajo del método main");
24         System.out.println("Fuera del bloque try-catch");
25         System.out.println("---Fin del Programa---");
26     }
27 }
```

Salida por Consola:

```
Capturamos el error de Null Pointer Exception
Resto del codigo debajo del método main
Fuera del bloque try-catch
---Fin del Programa---
```

Aquí podemos ver dos excepciones, una en la línea 11 y otra en la línea 12, pero a la vez ocurrió una excepción(en la 11) y se ejecuta el bloque de captura correspondiente.

Suponga que hay `ArithmeticException` en el bloque `try`. Pero puede ver que no hay un tipo de excepción correspondiente. En tal caso, se ejecutará la clase Excepción principal (padre)(`Exception`) cuya "Exception" contenía el bloque `catch`.

Y también recuerde si intenta manejar la excepción sin tener en cuenta el orden de las excepciones, que van de las más específicas a las generales, definitivamente obtendrá un error de tiempo de compilación.

Bloque try anidado

En algún programa, en parte de un bloque ocurrirá una excepción y en el bloque completo ocurrirá otra excepción. En ese caso usamos bloques `try` anidados. Simplemente el bloque `try` anidado es un bloque `try` dentro de un bloque `try`. Recuerdan el caso de los `for` anidados esto es algo similar.

```
3 public class BloquesAnidadosTry {
4
5     public static void main(String[] args) {
6         // Bloques anidados try
7         try {
8             try {
9                 String nombre=null;
10                System.out.println(nombre.length());
11            }catch(NullPointerException e) {
12                System.out.println("Puntero fuera de un null");
13            }
14            try {
15                int vector [] = new int[5];
16                vector[20]=5;
17            }catch(ArrayIndexOutOfBoundsException e) {
18                System.out.println("Excepcion rango fuera del indice");
19            }
20            System.out.println("Fin del bloque interno del try anidado");
21
22        }catch(Exception e) {
23            System.out.println("Error controlado por la excepcion padre");
24            System.out.println("Block externo del catch");
25        }
26        System.out.println("Fin del bloque externo del try");
27    }
}
```

Salida por Consola:

Puntero fuera de un null
Excepcion rango fuera del indice
Fin del bloque interno del try anidado
Fin del bloque externo del try

BLOQUE FINALLY

Este bloque se utiliza para ejecutar el código importante del programa. Ya sea que se maneje una excepción o no, el bloque finally se ejecutará. El bloque finally debe colocarse debajo del bloque try y del bloque catch.

Si no se produjo una excepción o se produjo u ocurrió una excepción manejada o no manejada, bajo cualquier circunstancia se ejecutará el bloque.

Un bloque try solo puede tener un bloque finally.

Si el programa sale por algún otro problema, el bloque finally no se ejecuta.

```
3 public class BlockFinally {
4
5     public static void main(String[] args) {
6         // Block Finally siempre se ejecuta
7
8         try {
9             String nombre=null;
10            System.out.println(nombre.length());
11        }catch (ArrayIndexOutOfBoundsException e) {
12            System.out.println("Erro array fuera del indice");
13        }finally {
14            System.out.println("Soy el bloque Finally. Yo siempre me ejecuto");
15        }
16
17        System.out.println("Resto del código del programa");
18
19    }
20 }
```

Salida por Consola:

```
Soy el bloque Finally. Yo siempre me ejecuto
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "nombre" is null
at excepciones.BlockFinally.main(BlockFinally.java:10)
```

En este ejemplo al controlar la excepción en el catch(línea 11) controlamos otro tipo de excepción, que es para un array o vector, no para este caso debería ser NullPointerException, por eso la excepción no es controlada, se sale del programa, pero como ven lo mismo se ejecuta el bloque finally.

palabra clave throw

Esta palabra clave se usa para lanzar una excepción. Al usar la palabra clave throw, puede lanzar excepciones comprobadas o no comprobadas. La palabra clave throw utilizada principalmente para lanzar excepciones personalizadas. Las excepciones personalizadas son creadas por los propios programadores.

```
3 public class PalabraClaveThrow {
4
5     public static void main(String[] args) {
6         // Palabra clave throw para lanzar excepciones personalizadas
7         // creadas por el programador
8         int contador=0;
9         double suma = 2358.85;
10        promedio(suma,contador); // llamamos a un método para hacer promedio
11        System.out.println("Resto del código debajo del main");
12        System.out.println("fin del programa");
13    }
14
15    public static void promedio(double b, int a) {
16        if(a==0) {
17            throw new ArithmeticException("error no se puede dividir entre 0");
18        } else {
19            System.out.println("El promedio es: "+b/a);
20        }
21    }
22 }
```

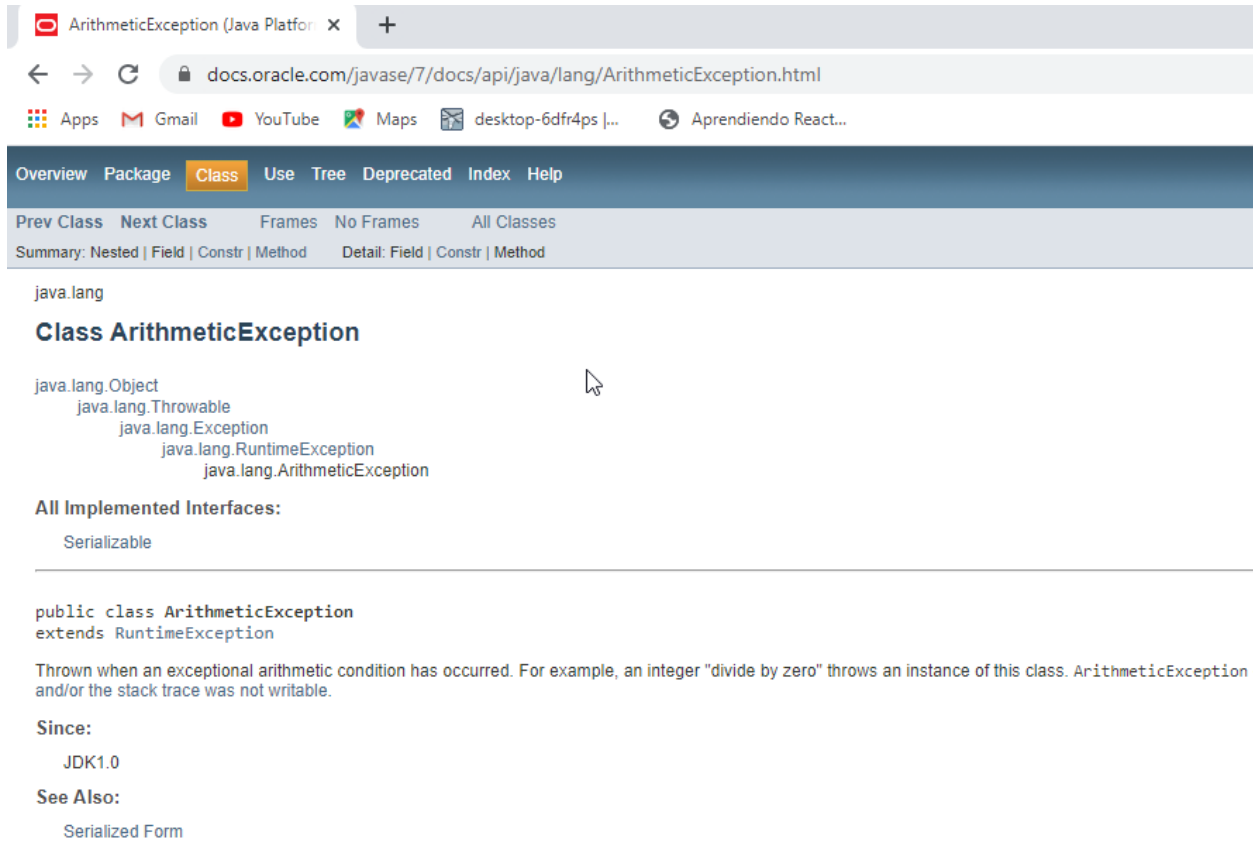
Salida por Consola:

```
Exception in thread "main" java.lang.ArithmeticException: error no se puede dividir entre 0
    at excepciones.PalabraClaveThrow.promedio(PalabraClaveThrow.java:17)
    at excepciones.PalabraClaveThrow.main(PalabraClaveThrow.java:10)
```

En este caso, en la línea 17 llamamos a ArithmeticException que es una clase que hereda de "Exception", que de acuerdo a lo que vimos en el gráfico de la página 3, ahí vemos su nombre y es la que controla este tipo de error, y le pasamos por parámetro un mensaje para que muestre en caso que se de la excepción.

A continuación, vemos un detalle de su clase que entrando en la página de Oracle podemos investigar más. El link para acceder es:

<https://docs.oracle.com/javase/7/docs/api/java/lang/ArithmeticException.html>



The screenshot shows the Oracle Java API documentation for the `ArithmeticException` class. The browser address bar shows the URL `docs.oracle.com/javase/7/docs/api/java/lang/ArithmeticException.html`. The page has a navigation bar with tabs for Overview, Package, Class (selected), Use, Tree, Deprecated, Index, and Help. Below the navigation bar, there are links for Prev Class, Next Class, Frames, No Frames, and All Classes. The main content area shows the class hierarchy: `java.lang.Object` → `java.lang.Throwable` → `java.lang.Exception` → `java.lang.RuntimeException` → `java.lang.ArithmeticException`. It also lists the implemented interfaces: `Serializable`. The class declaration is shown as `public class ArithmeticException extends RuntimeException`. A description states: "Thrown when an exceptional arithmetic condition has occurred. For example, an integer 'divide by zero' throws an instance of this class. ArithmeticException and/or the stack trace was not writable." It also mentions "Since: JDK1.0" and "See Also: Serialized Form".

Propagación de excepciones en Java.

Primero se lanza una excepción desde la parte superior de la pila y, si no se detecta, desciende de la pila de llamadas al método anterior; si no se detecta allí, la excepción vuelve a descender al método anterior, y así sucesivamente hasta que se detecta. o hasta que llegue al final de la pila de llamadas. A esto se le llama propagación de excepciones.

Las excepciones no comprobadas se reenvían en la cadena de llamadas y las excepciones comprobadas no se reenvían en la cadena de llamadas.

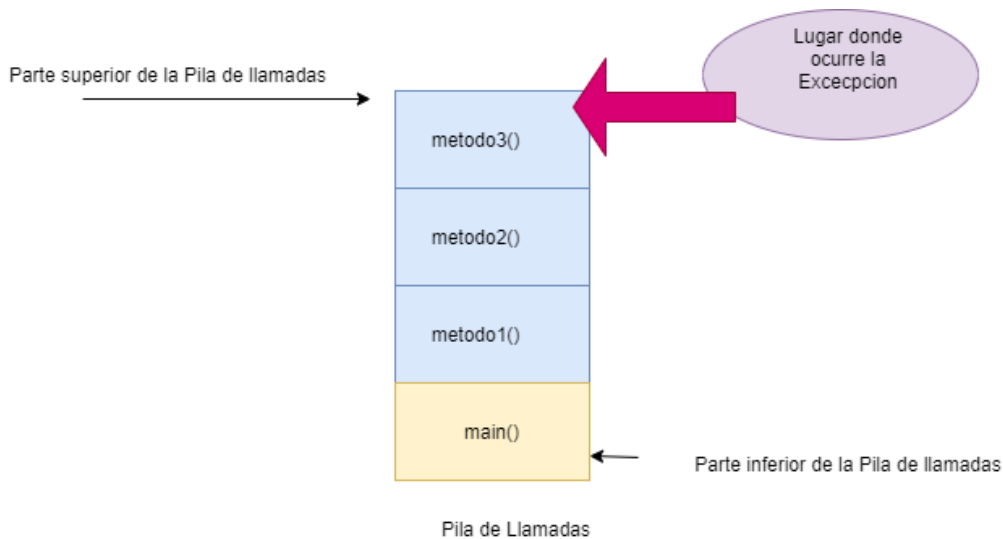
Veamos un ejemplo de propagación de excepciones no comprobadas,

```
3 public class PropagacionExcepciones {
4
5     public static void main(String[] args) {
6         // Propagación de Excepciones
7
8         System.out.println("Hola en el main antes de que ocurran excepciones");
9         PropagacionExcepciones obj = new PropagacionExcepciones();
10
11         try {
12             obj.metodo1();
13         } catch (ArithmeticException e) {
14             System.out.println("Aca manejamos la excepcion");
15         }
16
17         System.out.println("Hola main luego de manejar la excepcion");
18         System.out.println("fin del programa");
19
20     } // Fin del main
21
22     public void metodo1() {
23         System.out.println("Hola metodo1 antes de que ocurran errores");
24         metodo2();
25         System.out.println("Volviendo al metodo1");
26     }
27
28     public void metodo2() {
29         System.out.println("Hola metodo2 antes de que ocurran errores");
30         metodo3();
31         System.out.println("Volviendo al metodo2");
32     }
33
34     public void metodo3() {
35         System.out.println("Hola metodo3 antes de que ocurran errores");
36         System.out.println(100/0);
37         System.out.println("A ocurrido una excepcion");
38     }
39 }
```

Y la salida por consola es:

```
Hola en el main antes de que ocurran excepciones
Hola metodo1 antes de que ocurran errores
Hola metodo2 antes de que ocurran errores
Hola metodo3 antes de que ocurran errores
Aca manejamos la excepcion
Hola main luego de manejar la excepcion
fin del programa
```

En este programa, puede ver que se produce una excepción en el método3. Pero no se maneja en el método3. Entonces se propagó al método anterior 2. No se maneja allí también. Por lo tanto, se propaga al método anterior en el que tampoco se maneja la excepción. Finalmente se propagó al método main y aquí se maneja la excepción.



Recuerde esto, la excepción puede manejarse en cualquier método de pila de llamadas, puede manejarse en `main()`, `metodo1()`, `metodo2()` o `metodo3()`.

palabra clave throws

Esta palabra clave **"throws"** se utiliza para declarar una excepción. Señala específica que puede ocurrir una excepción en el método. Nunca arroja excepciones.

Sintaxis de throws

```
Tipo_retorno nombre_metodo() throws Nombre_de_la_Excepcion {  
    // código del metodo  
}
```

El Nombre_de_la_Excepcion es una de las clases que puede ser Exception o alguna de las que heredan de ella, ver figura de la página 4.

Bajo este solo declara excepciones comprobadas. Debido a que las excepciones no comprobadas están bajo el control del programador, el programador debe corregirlas y la clase Error está fuera del control del programador.

La principal ventaja del uso de la palabra clave throws es que se puede propagar la excepción comprobada.

Hay dos tipos,

1. Atrapó la excepción.
2. Declara la excepción.

Veamos un ejemplo para cada tipo,

Capturó la excepción (maneje la excepción usando try / catch)

```
3 import java.io.IOException;
4
5 public class PalabraClaveThrows {
6
7     public static void main(String[] args) {
8         // Palabra Clave throws
9
10        PalabraClaveThrows obj = new PalabraClaveThrows();
11        obj.metodo1();
12        System.out.println("Estamos en el main()");
13        System.out.println("Fin del Programa");
14    }
15
16
17    public void metodo3() throws IOException {
18        throw new IOException("Excepcion entrada/salida");// excepcion chequeada
19    }
20
21    public void metodo2() throws IOException {
22        metodo3();
23    }
24 }
```



```
25 public void metodo1() {  
26     try {  
27         metodo2();  
28     } catch (Exception e) {  
29         System.out.println("Excepcion manejada");  
30     }  
31 }  
32
```

Salida de la consola:

```
Excepcion manejada  
Estamos en el main()  
Fin del Programa
```

Declaras la excepción (especificando la palabra throws con el método)

Aquí, si no se produce una excepción, el código se ejecutará correctamente.

Pero si ocurre alguna excepción, se lanzará una excepción en tiempo de ejecución porque la palabra throws no maneja la excepción.

Vemos el siguiente ejemplo si no ocurre una excepción,

```
5 public class PalabraClaveThrows2 {  
6  
7     public static void main(String[] args) throws IOException {  
8         // Palabra Clave throws  
9         // Aca no se maneja la excepcion  
10  
11         metodo1();  
12         System.out.println("Estamos en el main()");  
13         System.out.println("Fin del Programa");  
14     }  
15  
16  
17     public static void metodo3() throws IOException {  
18         System.out.println("La excepcion no ocurre");  
19     }  
20  
21     public static void metodo2() throws IOException {  
22         metodo3();  
23     }  
24  
25     public static void metodo1() throws IOException {  
26         metodo2();  
27     }  
28 }
```

La salida por la consola es:

```
La excepcion no ocurre  
Estamos en el main()  
Fin del Programa
```

Ahora vemos un ejemplo si la excepción ocurre:

```

7 public static void main(String[] args) throws IOException {
8     // Palabra Clave throws
9     // No se controla o maneja la excepcion y ocurre una excepcion
10
11     metodo1();
12     System.out.println("Estamos en el main()");
13     System.out.println("Fin del Programa");
14
15 }
16
17 public static void metodo3() throws IOException {
18     throw new IOException("Excepcion entrada/salida");// excepcion chequeada
19 }
20
21 public static void metodo2() throws IOException {
22     metodo3();
23 }
24
25 public static void metodo1() throws IOException {
26     metodo2();
27 }

```

Y la salida por la consola sería:

```

Exception in thread "main" java.io.IOException: Excepcion entrada/salida
    at excepciones.PalabraClaveThrows3.metodo3(PalabraClaveThrows3.java:18)
    at excepciones.PalabraClaveThrows3.metodo2(PalabraClaveThrows3.java:22)
    at excepciones.PalabraClaveThrows3.metodo1(PalabraClaveThrows3.java:26)
    at excepciones.PalabraClaveThrows3.main(PalabraClaveThrows3.java:11)

```

Diferencias entre throw y throws en Java

throw	throws
Utilizado para lanzar explícitamente una excepción	Utilizado para declarar una excepción
La excepción comprobada no se puede propagar usando throw	La excepción comprobada se puede propagar con throws
Seguido de una instancia	Seguido de una clase
Utilizado dentro del método	Utilizado dentro de la firma del método
No puede lanzar múltiples excepciones	Puede declarar múltiples excepciones

Diferencia entre final, finally y finalize

final	finally	finalize
Se utiliza para aplicar restricciones a clase, método o variable. La clase final no se puede heredar, el método no se puede sobrescribir y el valor de la variable no se puede cambiar.	Utilizado para colocar código importante, se ejecutará si las excepciones se manejan o no	Se utiliza para realizar el proceso de limpieza justo antes de que el objeto se recoja como basura.
Es una Palabra clave. Una constante.	Es un bloque.	Es un método.

Final

Veamos el siguiente ejemplo:

```

3 public class CasoFinal {
4
5     public static void main(String[] args) {
6         // Variable del tipo final (constantes)
7
8         final int valor=10;
9
10        if (valor<15) {
11            valor=55; // No se le puede modificar el valor a
12                    // una constante.
13        }
14
15    }
16
17 }
```

Finally

```
3 public class ClaseFinally {  
4  
5     public static void main(String[] args) {  
6         // Bloque finally  
7         try {  
8             int cuenta = 245/0;  
9         } catch (Exception e) {  
10             System.out.println(e);  
11             System.out.println("Se maneja la excepcion");  
12         } finally {  
13             System.out.println("El bloque finally es ejecutado");  
14         }  
15     }  
16 }  
17  
18 }
```

La salida por consola es:

```
java.lang.ArithmeticException: / by zero  
Se maneja la excepcion  
El bloque finally es ejecutado
```

Finalize

```
3 public class ClaseFinalize {
4
5     public static void main(String[] args) {
6         // Clase Finalize
7         // Necesidad de eliminar memoria
8
9         ClaseFinalize obj1 = new ClaseFinalize();
10        ClaseFinalize obj2 = new ClaseFinalize();
11
12        obj1=null;
13        obj2=null;
14        System.gc();// gargarage collector
15
16    }
17    @Override
18    protected void finalize() {
19        System.out.println("Metodo finalize()");
20    }
21
22 }
```

Excepción personalizada en Java

En la programación, a veces necesitamos nuestra propia excepción. Por eso necesitamos una excepción personalizada. Si está creando su propia excepción que se conoce como excepción personalizada o excepción definida por el usuario. Excepciones personalizadas, personalice la excepción según las necesidades del usuario.

Veamos algunos ejemplos,

```
3 public class Ejemplo1 {
4
5     public static void main(String[] args) {
6         // ejemplo de utilizar mi propia excepcion
7
8         // Voy a utilizar mi excepcion creada llamada MiExcepcion
9         int a=5;
10        int b=0;
11        try {
12            int resultado=dividir(a,b);
13            System.out.println("El resultado es:"+resultado);
14        } catch (Exception e) {
15            System.out.println("Ocurrio una excepcion");
16            System.out.println("El mensaje es :"+e);
17        }
18
19    } // fin del main
20
21    public static int dividir(int a,int b) throws MiExcepcion {
22        int result=0;
23        if (b==0) {
24            throw new MiExcepcion("No se puede dividir por 0");
25        } else {
26            result=a/b;
27        }
28        return result;
29    }
30}
```

Y el resultado por consola es:

```
Ocurrio una excepcion
Esto es :excepciones.MiExcepcion: No se puede dividir por 0
```

```
3 public class MiExcepcion extends Exception{  
4     // Mi propia Excepcion  
5     MiExcepcion (String mensaje){  
6         super(mensaje);  
7     }  
8  
9 }  
10
```

Acá esta última clase es otra forma de hacer una clase para el manejo de excepciones que hereda de la clase “Exception” que es la clase padre de todas las excepciones según el gráfico de la figura que está en la página 4.

En esta clase ponemos nuestro propio constructor para controlar la Excepción que queramos.

Si se trata de Manejo de excepciones en Java. Creo que esto le ayudará a obtener un buen conocimiento sobre el manejo de excepciones en Java de una manera fácil.