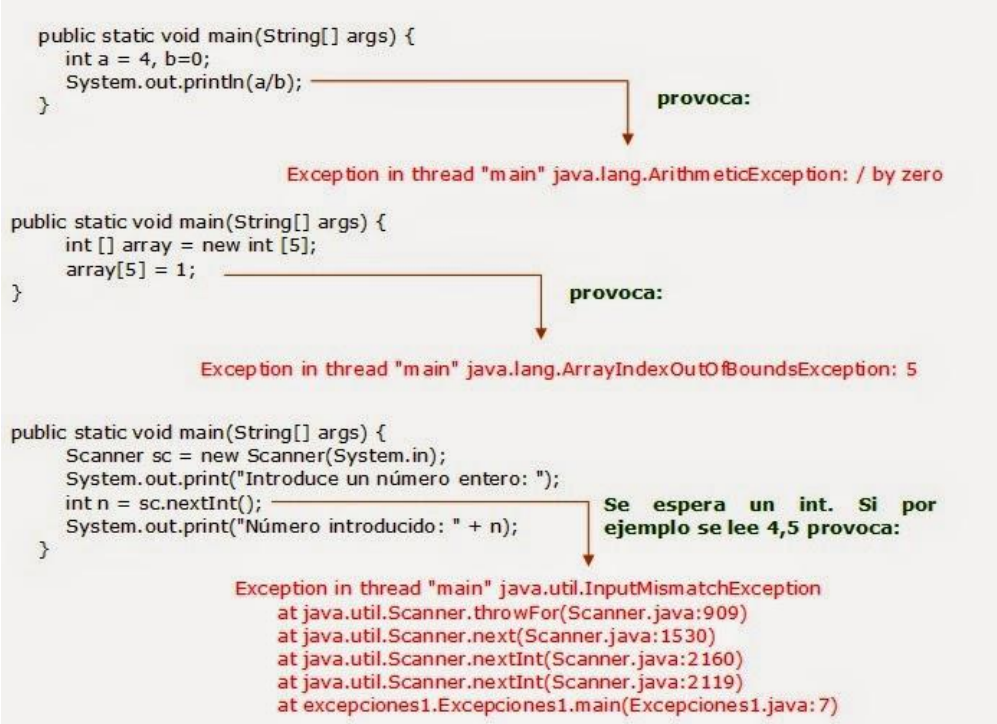
EXCEPCIONES Java.

1.- Manejo de Excepciones en Java.

Una Excepción (**Exception** en Java) indica una situación anómala en la ejecución de un programa. Si no se "maneja" esta excepción, la JVM interrumpirá de forma **abrupta** la ejecución del programa, informando de la excepción "culpable" de la interrupción. **Ejemplos:**



¿Qué ocurre cuando se produce una excepción?

Cuando se produce una excepción, Java realizará la secuencia de acciones que vamos a ver a continuación. Si como programadores/as no hacemos nada, el programa se interrumpe y muestra información sobre la excepción, tal y como se muestra en los ejemplos anteriores.

Acciones tras una excepción (antes de interrumpir definitivamente el programa):

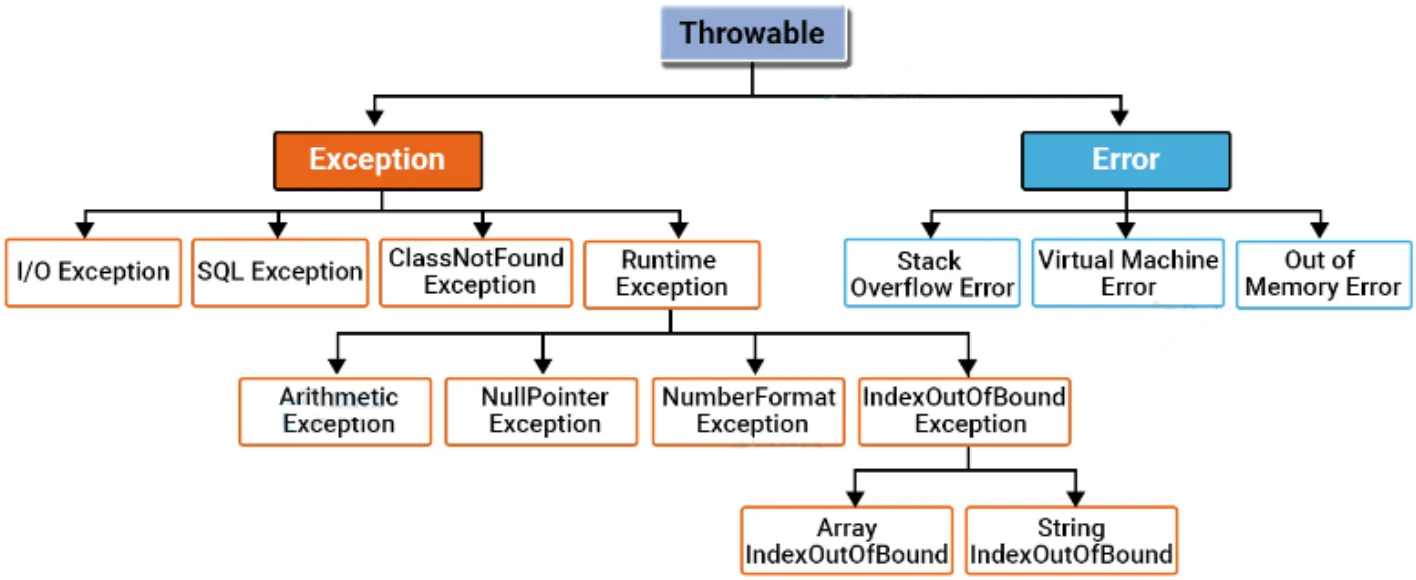
1. Java crea un objeto de tipo **Exception** y lo lanza. El objeto **Exception** creado contiene información sobre la excepción. El programa aún no se interrumpe, pero su ejecución **normal** se detiene y la JVM ejecuta el paso 2.
2. Se busca en el método donde se ha producido la excepción un **manejador de excepciones (bloque try-catch)** que capture ese objeto **Exception** generado, y que gestione la excepción.
3. Si el método dónde se produjo el problema no incluye un manejador de excepciones apropiado, se devuelve el control al método que lo llamó, y así sucesivamente hasta llegar al **main**().
4. Si en algún momento se encuentra un manejador de excepciones compatible con el tipo de excepción que se ha producido, se le pasa la excepción para que la trate.
5. Si no se encuentra un manejador de excepciones compatible, **Java muestra el error y acaba el programa, que es lo que ha sucedido en los 3 casos que se muestran la imagen inicial.**

EN RESUMEN: Una **Exception** en Java nos indica una situación anómala en la ejecución de un programa, pero en todo caso se trata de una anomalía **tipificada** y **recuperable,** lo que nos va a permitir en muchos casos **manejarla** y evitar el cierre de la aplicación.

El manejo de **excepciones** en Java facilita enormemente el tratamiento de errores, y contribuye a que se creen programas más robustos y que toleren mejor los fallos. **Capturando** y **manejando** excepciones escribiremos código para la gestión de errores limpio, claro y fácil de modificar y mantener

En programación, a menudo la aparición de errores inevitables puede ser **previsible**, y a través de las excepciones podremos prevenir y "curar". Con la experiencia, aprenderemos a identificar los "puntos negros" en nuestro código en los que es probable que se produzcan excepciones.

Errores y **Excepciones** java. La clase **Throwable**



Un Error es bastante diferente, pues nos advierte que se ha generado un **problema no gestionable dentro del programa**, con lo que la ejecución normal del programa no se podrá recuperar. En ese caso poco se puede hacer (Memoria agotada, error Hardware o a nivel de SO, etc.)

Una **Exception,** como hemos comentado, tiene muchas más posibilidades de **manejo** y **solución**, pudiendo evitarse en muchas ocasiones la interrupción del programa. En otras, como mal menor, al menos la gestión de la **Exception** va a permitirnos lograr una interrupción "suave" y no abrupta del programa.

- **RuntimeException:** excepciones originadas durante la ejecución del programa: ArithmeticExcepcion, NullPointerException, ArrayIndexOutOfBoundsException, etc.
- **ClassNotFoundException:** excepción originada cuando una aplicación intenta cargar una clase pero no se encuentra el fichero .class correspondiente
- **SQL Exception:** problemas en conexiones JDBC con los diferentes Sistemas de Gestión de Bases **de Datos.**
- **IOException:** excepciones al ejecutar una operación de entrada-salida. Veremos más adelante cuando trabajemos con archivos que estas excepciones están marcadas **(checked)**, por lo que estamos obligados a manejarlas. El IDE nos avisará y ayudará en este sentido.

TRATAMIENTO - "MANEJO" DE EXCEPCIONES. try – catch - finally

```
try {
     /* un bloque try siempre debe rodear los "puntos negros" en el código (instrucciones en las
     que es probable que se produzcan excepciones). Si se producen anomalías, Java lanza la
     correspondiente Exception y busca un bloque catch compatible con ella */
}
catch (tipoException1 e ){
    /* un bloque catch actúa como "manejador" para cada tipo de excepción (1 bloque por Exception)
     como programadores/as, aprenderemos a identificar cuando pueden producirse
     las excepciones más comunes, para tener preparado su bloque catch() correspondiente */
}
catch (tipoException2 e ){

}
…
// Se pueden escribir tantos bloques catch como creamos necesario, 1 por Exception.

finally {
  /* Es un bloque OPCIONAL. Sus instrucciones se ejecutarán siempre después del try.
     Un bloque finally se usa para dejar un estado consistente después de ejecutar el bloque try.
     Ejemplo: Los archivos, las conexiones de bases de datos y las conexiones de red que no se cierren
     apropiadamente podrían no estar disponibles para su uso en otros programas. Podemos escribir un
     bloque finally para cerrarlos. Se ejecutará siempre, se produzca o no una excepción */
 }
```

A partir de **Java 7** hay disponible un recurso más práctico y sencillo que el bloque **finally.** Se trata del **try-with resources**, que permite cerrar recursos que implementen el interface **Closeable** automáticamente**. Volveremos a hablar de ello en los ejemplos.**

Ejemplo 1: Leer un **int** por teclado y mostrarlo. Si **nextInt**() recibe cualquier cosa que no sea un **int**, Java lanza una **InputMismatchException,** que capturamos en el bloque **catch,** donde se maneja la situación. Con un **do-while** permitimos la entrada de un nuevo dato las veces que sea necesario. Es un ejemplo de cómo conseguir que después de una excepción un programa **se recupere,** no sólo evitando la salida abrupta del mismo, sino también repitiendo la acción que provocó la excepción hasta lograr ejecutar el código con éxito.

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    boolean terminar;
    int n=0;
    do{
      terminar=true;
      try{
           System.out.println("Introduce un número entero: ");
           n = sc.nextInt();
      }catch (InputMismatchException e){
           sc.nextLine();   // lo hacemos siempre después de un nextInt() para "limpiar" el buffer de teclado
           terminar=false;
           System.out.println("NO ES UN ENTERO. La Excepción que has originado es: " + e.toString());
      }
    }while (!terminar);
    System.out.println("El número introducido es :  " + n);
}
```

Ejemplo 2: En el siguiente ejemplo se produce una excepción en el método **muestraArray()** al intentar acceder a una posición no existente del Array (Excepción **ArrayIndexOutOfBoundsException)**.
No hay bloque **Try-catch** en el método **muestraArray**(), pero como Java propaga las excepciones hacía los métodos "llamadores", la excepción llega al **main**(), donde podemos capturarla y manejarla.

```
public static void main(String[] args) {
    int [] numeros = {3,6,8,4,9,8,2,10};
    try{
      muestraArray(numeros);
    }
    catch (ArrayIndexOutOfBoundsException e){
      System.out.println("Se ha intentado acceder a una posición no existente en el Array. Excepción Java: " + e.toString());
    }
}
public static void muestraArray(int [] arr) {
    for (int i=0; i<=arr.length;i++){
        System.out.println(arr[i]);
    }
}
```

Ejemplo 3: tratamiento/manejo de 2 posibles excepciones. En este ejemplo se capturan en el **main** 2 excepciones. La **1ª** puede producirse en el **nextInt**() si nos introducen algo que no sea un **int**. La **2ª** puede producirse en el método **muestraPosArray()** si la posición del Array a mostrar no existe.

```
public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    int [] numeros = {3,6,8,4,9,8,2,10};
    int n=0;
    System.out.println("Teclea la posición del Array que quieres consultar");
    try{
        n =sc.nextInt();
        muestraPosArray(numeros,n);
    }
    catch (InputMismatchException e1){
       System.out.println("No has tecleado un entero. Excepción Java: " + e1.toString());
    }
    catch (ArrayIndexOutOfBoundsException e2){
        System.out.println("Posición no existente en el Array. Excepción Java: " + e2.toString());                    ");
    }
}
public static void muestraPosArray(int [] arr, int num) {
      System.out.println(arr[num]);
}
```

En los ejemplos **2 y 3** capturamos las excepciones que se puedan producir, pero no damos un nuevo intento al programa para volver a ejecutar las instrucciones "culpables" de la excepción. Logramos que el programa finalice de forma controlada, pero nada más. **Otra opción sería hacer como en el Ejemplo 1:**

```
public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    int [] numeros = {3,6,8,4,9,8,2,10};
    boolean terminar;  int n=0;
    do{
       terminar=true;
       try{
            System.out.println("Teclea la posición del Array que quieres consultar: ");
            n = sc.nextInt();
            muestraPosArray(numeros,n);
       }
```

```
        catch (InputMismatchException e1){
            System.out.println("No has tecleado un entero. Excepción Java: " + e1.toString());
            sc.nextLine();   // lo hacemos siempre después de un nextInt() para "limpiar" el buffer de teclado
            terminar=false;
        }
        catch (ArrayIndexOutOfBoundsException e2){
            System.out.println("Posición no existente en el Array. Excepción Java: " + e2.toString());
            terminar=false;
        }
    }while (!terminar);
}
public static void muestraPosArray(int [] arr, int num) {
    System.out.println(arr[num]);
}
```

EXCEPCIONES MARCADAS Y NO MARCADAS (checked y unchecked)

- Excepciones **no marcadas** (unchecked), son aquellas que no estamos obligados a tratar.
- Excepciones **marcadas** (checked) son aquellas que estamos obligados a tratar.

Las excepciones que hemos probado en los ejemplos pertenecen a la clase **RuntimeException** y son **unchecked**. No estamos obligados a tratarlas, y de hecho **llevamos todo el curso sin hacerlo,** de ahí que muchas veces nuestros programas terminen con error.

Hay otras excepciones, como las que nos encontraremos cuando trabajemos con **Archivos** (clase **IOException**) que son **checked**, y Java nos obliga a manejarlas sí o sí. De lo contrario el **programa no compila**. Java "protege" así operaciones delicadas como la lectura/escritura sobre archivos.

Eclipse, NetBeans, BlueJ o cualquier IDE que empleamos para programar en Java nos avisa de este hecho, **a la vez que nos ayuda y automatiza el manejo y declaración de las excepciones marcadas.**

DECLARAR EXCEPCIONES – cláusula throws

Cuando en un método puede producirse una excepción, y no la vamos a manejar en el método, debe especificarse ese comportamiento, para que los métodos que lo llamen a él lo tengan en cuenta.

Para ello se incluye una **cláusula throws** en la cabecera/firma del método. Esta cláusula es un listado de los tipos de excepciones (separadas por comas) que se pueden originar en el método. **Ejemplo**:

```
public static void muestraArray(int [] arr) throws ArrayIndexOutOfBoundsException{}
```

Es una advertencia en toda regla. Significa que el método **muestraArray**() puede provocar una excepción de tipo **ArrayIndexOutOfBoundsException**, y que debemos proteger las llamadas a muestraArray() con un **try-catch.**

```
try{
  muestraArray(numeros);
}
catch (ArrayIndexOutOfBoundsException E){
  System.out.println("Se ha intentado acceder a una posición no existente en el Array");
}
```

Esto lo hemos hecho en el ejemplo 2, y funcionaba sin **throws**. Entonces, **¿para que hacer throws?**

No es obligatorio declarar las excepciones **unchecked** que se puedan producir en un método, pero es **recomendable** hacerlo para facilitar la reutilización de Código. Añadiendo **throws** avisamos de las posibles excepciones que se pueden producir en un método a quién lo vaya a utilizar posteriormente.

ES OBLIGATORIO throws, y **el programa NO COMPILA,** si las **excepciones que se pueden producir son marcadas (checked).** Si en un método se puede producir cualquier tipo de **IOException** y no la manejamos, es obligatorio avisar con **throws**, pues de lo contrario se produce un error de compilación.

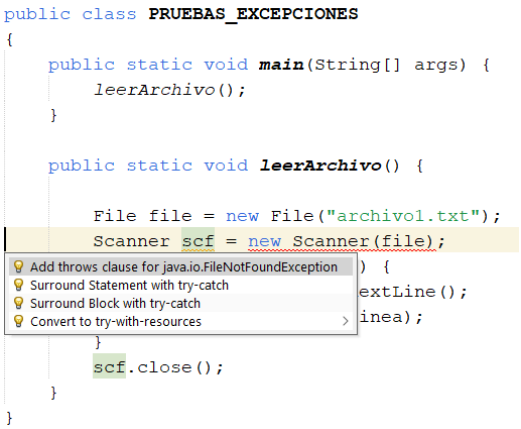
En adelante, hemos de fijarnos en las cabeceras de los métodos cuando usemos métodos del API de java o de terceros. A menudo incluirán **throws** para avisarnos de las excepciones que se pueden producir.

Ejemplo 4: Excepción marcada(checked) **FileNotFoundException.**

Para probar este ejemplo creamos con **notepad** un archivo de texto llamado "**archivo1.txt**". Insertamos unas cuantas líneas de texto y lo almacenamos en la carpeta de nuestro proyecto Netbeans/Eclipse.
El siguiente programa escribirá por pantalla el contenido del archivo "**archivo1.txt**" línea a línea.

```
public class PRUEBA_EXCEPCIONES
{
   public static void main(String[] args) {
            leerArchivo();
   }
   public static void leerArchivo() {
        File file = new File("archivo1.txt");
        Scanner scf = new Scanner(file);     // Aquí se puede producir una FileNotFoundException
        while (scf.hasNextLine()) {
            System.out.println(scf.nextLine());
         }
         scf.close();
   }
}
```

Java da **error en compilación**, pues se puede producir una excepción de **IO** y estamos **OBLIGADOS** a manejarla. Afortunadamente, Netbeans/Eclipse ofrecen opciones para resolver el error. La **1ª** añadir una cláusula **throws FileNotFoundException** al método, para avisar al método main() de que debe manejar la excepción. La **2ª y 3ª** es añadir en leerArchivo() un **try-catch** para manejar la **FileNotFoundException.**



```
public class Pruebas_Excepciones   //Posible solución
{
   public static void main(String[] args) {
      File file = new File("archivo1.txt");
      Scanner scf = null;
      try {
         scf = new Scanner(file);
         leerArchivo(scf);
      }
      catch (FileNotFoundException E){
         System.out.println("El archivo NO EXISTE");
      }
      finally {
         if (scf != null) {
             scf.close();
         }
      }
   }
   public static void leerArchivo(Scanner f) {
      while (f.hasNextLine()) {
         System.out.println(f.nextLine());
      }
   }
}
```

LANZAR EXCEPCIONES – throw

Java nos permite lanzar **nuestras propias excepciones voluntariamente** con un **throw** (muy diferente a **throws**). cada excepción es un objeto, por lo que hay que crearlo como cualquier otro objeto, mediante **new()**. Tiene sentido usar **throw** cuando queremos señalar una **situación anómala en nuestro "negocio" que** no está tipificada en el lenguaje Java *(StockCero, Libroagotado, ContactoNoEncontrado, etc)*. Lanzaremos excepciones propias en nuestros programas cuando se produzcan situaciones anómalas en el "negocio" para el que estamos desarrollando una aplicación.

Ejemplo. Lanzar con **throw** una excepción para chequear que un valor introducido en una variable **int** edad es **<18 ó >18.** Si **edad** toma un valor **15** para Java **todo está bien,** y no lanzará excepción alguna**.** Pero puede interesarnos a nosotros/as lanzar (**throw)** una excepción para señalar que ese valor **<18** no es correcto.

```
public class EjemploThrow {
 static void checkEdad(int edad) {
    if (edad < 18) {
       throw new ArithmeticException("Acceso Prohibido a menores de 18 años");
    } else {
       System.out.println("Acceso permitido");
    }
 }
 public static void main(String[] args) {
    boolean continuar=true;
    Scanner sc=new Scanner(System.in);
    try{
        System.out.println("Teclea tu edad: ");
        Int edad=sc.nextInt();
        checkEdad(edad);
    }
    catch (ArithmeticException e){
       System.out.println(e.toString());
    }
 }}
```

La excepción es lanzada **(throw)** por nosotros/as cuando el valor **edad es <18**.

- En este caso lanzamos voluntariamente una excepción propia de java **(ArithmeticException),** pero lo más correcto y habitual para estos casos es **c**rear excepciones propias para tipificar de forma clara el problema y no confundir (en este caso realmente no hay ninguna anomalía Aritmética). Sería mejor crear una Excepción propia llamada **MenorDeEdad**, por ejemplo.
- **Buena Práctica**: deberíamos haber avisado de que en el método **checkEdad()** puede saltar una excepción. No es obligatorio, pues una **ArithmeticException** es unchecked, pero hacerlo es una buena práctica.

```
static void checkEdad(int edad) throws ArithmeticException {}
```

CREAR NUESTRAS PROPIAS EXCEPCIONES.

Java proporciona una gran cantidad de excepciones **(Ver Anexo**) que como hemos visto en el ejemplo anterior podemos adaptar a nuestros fines. Aun así, cuando se desarrollan aplicaciones es bastante habitual requerir excepciones que no están definidas en el API de Java, es decir, **excepciones propias.**
Las excepciones propias deben ser **subclases de la clase Exception** y han de construirse como si de cualquier otra clase se tratara. Estas excepciones se podrán lanzar con **throw** igual que las excepciones del API, pero pueden generar información más clara de un problema debido a que las podemos hacer a nuestra medida.

Ejemplo: Excepción propia **ArticuloAgotado** para controlar en el ejemplo de la tienda de Informática si un artículo está agotado.

```
//Creamos la clase correspondiente a la nueva excepción
public class ArticuloAgotado extends Exception{
    public ArticuloAgotado(String cadena){
            super(cadena); //Llama al constructor de Exception y le pasa el contenido de cadena
    }
}

//En el método buscarId que utilizamos para buscar artículos lanzaremos la excepción si no hay existencias
public int buscarId(String id) throws ArticuloAgotado {
      int i=0;
      for (Articulo a : articulos) {
         if (a.getIdArticulo().equals(id)){
           if (a.getExistencias() <= 0) {
              throw new ArticuloAgotado ("No hay unidades del artículo: " + a.getDescripcion());
           }
           break;
         }
         i++;
      }
      return i;
   }

// manejamos la excepción un try-catch a la hora de añadir un artículo al ArrayList articulosPedido
      try{
          articulosPedido.add(articulos.get(buscarId(codigo)));
      }
      catch (ArticuloAgotado E){
         System.out.println(E.getMessage());
      }
```

try with resources (A partir de Java 7). Permite cerrar automáticamente recursos que implementan la interface **Closeable.**

Partimos del sencillo Ejemplo visto antes que nos servía para leer un archivo de texto:

```
public class Pruebas_Excepciones
{
   public static void main(String[] args) {
      File file = new File("archivo1.txt");
      Scanner scf = null;
      try {
         scf = new Scanner(file);
         leerArchivo(scf);
      }
      catch (FileNotFoundException E){
         System.out.println("El archivo NO EXISTE");
      }
      finally {
         if (scf != null) {
            scf.close();
         }
      }
   }
   public static void leerArchivo(Scanner f) {
      while (f.hasNextLine()) {
         System.out.println(f.nextLine());
      }
   }
}
```

Alternativa *try-with-resources*: La mayoría de los **IDE** nos la ofrecen automáticamente, si la máquina virtual de java con la que trabajamos es posterior a **Java 7**.

```
   public static void main(String[] args) {
      File file = new File("archivo1.txt");
      try (Scanner scf = new Scanner(file)) {
         leerArchivo(scf);
      }
      catch (FileNotFoundException E){
         System.out.println("El archivo NO EXISTE");
      }
   }
```

Se puede intentar abrir **varios recursos a la vez en la misma sentencia try.** Sólo hay que separarlos con **;**
Ejemplo: leer un archivo con un **Scanner** (Recurso 1) y simular una impresión con un objeto **PrintWriter** (Recurso 2).
Lo que haremos es imprimir de un archivo a otro.
Código simplificado al máximo y con cierre automático de recursos usando **try-with-resources**

```
    public static void main(String[] args) {
        try (Scanner scf = new Scanner(new File("archivo1.txt")); PrintWriter writer = new PrintWriter(new File("archivo2.txt")))
        {
           while (scf.hasNextLine())
           {
              writer.println(scf.nextLine());
           }
        }
        catch (FileNotFoundException E){
           System.out.println("El archivo NO EXISTE");
        }
     }
```

ANEXO. EXCEPCIONES MÁS COMUNES EN JAVA.

EXCEPCIÓNES JAVA	DESCRIPCIÓN Y USO
IndexOutOfBoundsException	Cuando se intenta acceder a un Array/String/Vector con un valor de índice inválido
ClassCastException	Lanzada cuanto intentamos convertir una referencia a variable a un tipo que falla la prueba de castro IS-A
NumberFormatException	Cuando se trata de convertir un String a un número
ArithmeticException	Se ha producido una condición aritmética excepcional
InputMismatchException	Se produce cuando intentamos cargar un tipo no adecuado en una variable numérica
NullPointerException	Lanzada cuando intentamos acceder a un objeto con una variable de referencia cuyo valor actual es NULL.
NumberFormatException	Lanzada cuando un método que convierte un String a un número recibe un String que no puede ser convertido
AssertionError	Lanzada cuando una sentencia Boolean retorna el valor falso después de ser evaluada
IOException	Lanza la excepción cuando ocurre un fallo o es interrumpida la operación en curso. 2 comunes subtipos de excepción de IOException son EOFException y FileNotFoundException
FileNotFoundException	Cuando al abrir un archivo no es encontrado
SQLException	Cuando ocurre un error en la Base de Datos
InterruptedException	Lanza la excepción cuando el Hilo es interrumpido
NoSuchMethodException	Cuando se llama a un método y este no es encontrado
IllegalArgumentException	Cuando un método recibe un argumento formateado de manera diferente a lo que el método esperaba
IllegalStateException	Lanzada cuando el estado del entorno no coincide con la operación que se intenta ejecutar
NullPointerException	Lanzada cuando intentamos acceder a un objeto con una variable de referencia cuyo valor actual es NULL
StackOverflowError	Lanzada cuando un método es invocado demasiadas veces, por ejemplo, recursivamente
NoClassDefFoundError	Cuando no se puede encontrar una clase que se necesita.
ClassNotFoundException	Cuando intentamos ejecutar un proyecto y no se encuentra una main-class
RuntimeException	Conjunto de excepciones **no marcadas** que pueden tener lugar durante el proceso de ejecución de un programa sobre la JVM
NegativeArraySizeException	Al intentar crear un array con longitud negativa
OutOfMemoryException	El intento de crear un objeto con el operador *new* ha fallado por falta de memoria
UnsatisfiedLinkException	Intento de acceder a un método nativo que no existe
InternalException	Este error se reserva para eventos que no deberían ocurrir. El usuario nunca debería lanzarla voluntariamente