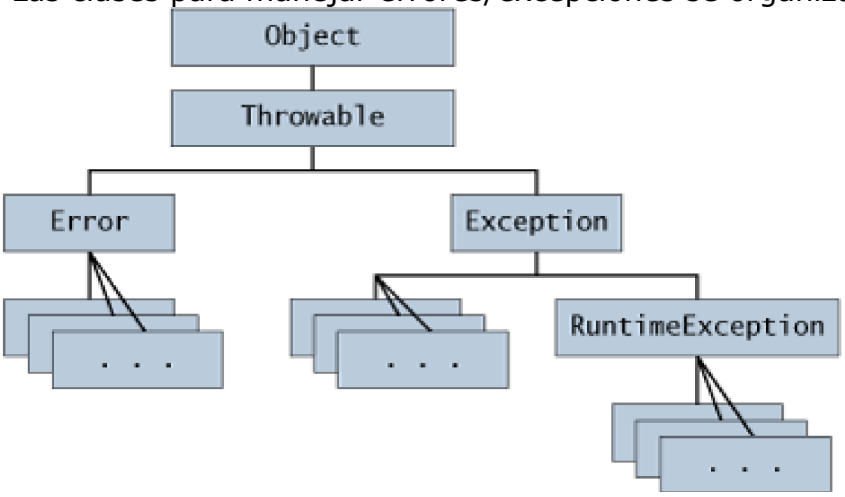
EXCEPCIONES

http://manuais.iessanclemente.net/index.php/Excepci%C3%B3ns

Tipos de Anomalías(no necesariamente "Errores") en los programas:

- Errores detectables en tiempo de compilación(javac)
- "Anomalías" detectables en tiempo de ejecución (java).En tiempo de ejecución hay dos tipos de "anomalías":
  - Errores asociados a la clase *Error*. Si ocurre uno, por ejemplo no hay suficiente memoria para ejecutar el programa, el programa se interrumpe sin que el programador pueda hacer nada.
  - *Excepciones* asociados a la clase *Exception*. Si ocurre, el programador tiene la oportunidad de que el programa "capture y trate" la excepción y tener así la posibilidad de que el programa siga ejecutándose.

Las clases para manejar errores/excepciones se organizan en jerarquía Throwable:



Throw significa "Lanzar". Throwable significa "Lanzable". Quiere esto decir que todos los objetos con superclase Throwable son "Lanzables". El subsistema java de tratamiento de errores y excepciones se basa en que cuando ocurre en ejecución un error o excepción se "Lanza" uno de estos objetos para tratar el error/excepción. ¿Quién lanza el objeto?. Dos posibilidades según contexto:

- La JVM
- La instrucción throw que está contenida en:
  - el código de nuestros programas
  - el código de las clases que usa nuestra programa(clases del API y otras escritas por nosotros o por terceros)

Throwable es la superclase del tratamiento de excepciones y errores. La subclase *Error* trata errores que se generan internamente en la máquina java(por ejemplo que no tiene suficiente memoria para ejecutarse) y no la utilizaremos. La clase *Exception* es con la que interaccionan nuestros programas y es la que nos interesa. Debajo de *Exception* sigue la jerarquía de clases y fundamentalmente nos vamos a encontrar con dos grupos de subclases:

- las subclases de RunTimeException, que soportan excepciones muy habituales como
  - *ArithmeticException* que "salta" por ejemplo al dividir por 0
  - *ArrayIndexOutOfBoundsException* que "salta" si el índice con el que se intenta

acceder a un Array está fuera de límite.
- Todas las demás.

Más adelante en este boletín aprenderemos que las RuntTimeExcepction son un tipo de Excepciones denominadas *no verificadas* y veremos qué significa esto.

El término excepción tiene en este contexto dos acepciones:
- el hecho de que se produjo en tiempo de ejecución una anomalía tratable por el programador.
- El objeto que tiene por superclase a la clase Exception del API java y se genera cuando se produce la anomalía

El mecanismo de tratamiento de excepciones hace básicamente dos cosas:
- crear el objeto excepción
- lanzar el objeto excepción.

EXCEPCIONES PARA LOS PRIMEROS EJEMPLOS.

Necesitamos generar excepciones en tiempo de ejecución. Para hacer ejemplos sencillos vamos a utilizar:
- Divisiones por cero
- Integer.parseInt("String") a este método estático de la clase *Integer* (no confundir con el tipo primitivo *int*), se le pasa un string que representa un entero con signo por ejemplo "+10" , "-10", "10". Pero si pasamos cadenas con contenido no entero del tipo "hola" o "10.5" el método genera una excepción

Ejemplo: Ejecuta el siguiente programa

```
class App {
   public static void main(String[] args) {
        int pesoPaquete;
        pesoPaquete= Integer.parseInt("-10");
        pesoPaquete= Integer.parseInt("+10");
        pesoPaquete= Integer.parseInt("10");
        pesoPaquete= Integer.parseInt("10.5");
        pesoPaquete= Integer.parseInt("hola");
   }
}
```

La ejecución se para en

```
pesoPaquete= Integer.parseInt("10.5");
```

Tras observar esto, comenta esta instrucción para ver que la instrucción

```
pesoPaquete= Integer.parseInt("hola");
```

también genera excepción.

Es decir: en tiempo de ejecución el programa se para en el momento que se produce una excepción, las instrucciones que siguen a la que produjo la excepción no se ejecutan. Más adelante veremos que esto se puede evitar si "tratamos" la excepción

Observa que java avisa que se produjo una excepción perteneciente a la clase *java.lang.NumberFormatException*.

Podemos observar en el API que es una subclase de Exception, no es un "hijo" directo, pero subclase al fin.

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException
```

Al ser hija de Throwable sus objetos son "lanzables", pero además al ser más concretamente hija de Exception es lanzable y tratable por el programador. Irás descubriendo poco a poco que es "lanzable" y "tratable por el programador".

¿Quién lanzó la excepción?
Las excepciones se generan en las instrucciones de los métodos al ejecutarse. El mecanismo de excepciones me informa que se produjo en la línea 7

```
at App.main(App.java:7)
```

que es

```
pesoPaquete= Integer.parseInt("10.5");
```

es decir, la excepción que se lanzó "viene de" el método ParseInt(), si consultamos este método en el API

```
public static int parseInt(String s)
                    throws NumberFormatException
```

comprobamos que efectivamente este método puede lanzar "throws" una NumberFormatException. Más adelante entenderás mejor la palabra clave "throws" y comprenderás cómo puede un método en su interior crear la excepción y lanzarla,

Ejemplo: Ejecuta el siguiente programa y comprueba que genera una excepción

```
class App {
   public static void main(String[] args) {
        int pesoPaquete;
        pesoPaquete= 10/0;
   }
}
```

Observa que java avisa que se produjo una excepción perteneciente a la clase *java.lang.ArithmeticException.*

IMPORTANTE: muchos de los ejemplos que siguen a continuación **no** deberían escribirse con try/catch si no simplemente con bloque if. Por ejemplo, todo programador debe de saber que no se puede dividir por 0 y debe incorporar un if para controlar si el denominador es 0. Idem al respecto al control de los límites de un array. El try/catch es para manejar algo excepcional, no algo previsible. Además el try/catch es mucho más ineficiente que la sentencia if

De todas formas en estos primeros ejemplos por motivos didácticos nos permitimos la licencia de usar try/catch indebidos.

RECUPERARSE DE UNA EXCEPCIÓN: LOS BLOQUES TRY Y CATCH

En el primer ejemplo, parseInt() lanzó una excepción, pero como nuestro código(lo que escribimos en el main()) no hizo nada por tratar esta excepción, el programa simplemente terminó "antes de tiempo". Si queremos tratar una excepción en nuestro código debemos

hacerlo dentro de los bloques try y catch.
Try: especifica el bloque de instrucciones en el que queremos controlar si se produce una excepción.
Catch: el bloque que contiene las instrucciones que queremos que ejecuten cuando se detecta una excepción en el bloque try.
try podemos traducirlo por "intentar" o "tratar" y catch por "capturar". Por tanto, "intentamos(tratamos de)" ejecutar las instrucciones del bloque try y si se produce una excepcion, recibiremos un objeto de la jerarquía Exception que contiene información de dicho excepción y se pasa por parámetro a un bloque catch, esto se suele describir como que el catch "coge" o "captura" la excepción.

Ejemplo: No se puede escribir un try sin un catch(o finally) asociado.
Observa cómo el compilador nos indica que falta el catch.

```
class App {
   public static void main(String[] args) {
        int pesoPaquete;
        try{
           System.out.println("antes de que se genere la excepción");
           pesoPaquete=10/0;
           System.out.println("esto no se imprime");
        }
   }
}
```

Ejemplo: añadimos el catch asociado al try anterior y ejecutamos el programa para observar el funcionamiento.

```
class App {
   public static void main(String[] args) {
      int pesoPaquete;
      try{
         System.out.println("antes de que se genere la excepción");
         pesoPaquete=10/0;
         System.out.println("esto jamás  se imprime");
      }catch(ArithmeticException miExcepcion){
         System.out.println("muy, muy, mu  mal:  no se puede dividir por cero");
      }
   }
}
```

En el bloque try se produce una excepción con la instrucción pesoPaquete=10/0; Antes de dicha excepción el programa se ejecutó con normalidad y se hizo el primer print(). Al producirse una excepción ArithmeticException, java crea un objeto de ese tipo y lo "lanza" fuera del bloque try buscando un catch que tenga como parámetro un objeto de dicho tipo.

Por cuestiones sintácticas el catch debe ser una continuación del try, no puede haber otras instrucciones en el medio observa el error en :

```
class App {
   public static void main(String[] args) {
        int pesoPaquete;
        try{
           System.out.println("antes de que se genere la excepción");
           pesoPaquete=10/0;
           System.out.println("esto jamás  se imprime");
        }
        int x=2;
        catch(ArithmeticException miExcepcion){
            System.out.println("muy, muy, mu  mal:  no se puede dividir por cero");

        }
  }
```

```
}
```

Respecto "al estilo de codificación java", los catch se prefieren escritos detrás de la llave del try, de la siguiente forma(observa diferencia con código de arriba):

```
class App {
   public static void main(String[] args) {
        int pesoPaquete;
        try{
           System.out.println("antes de que se genere la excepción");
           pesoPaquete=10/0;
           System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
            System.out.println("muy, muy, mu  mal:  no se puede dividir por cero");

        }
  }
}
```

Es decir, similar a los else de los if

Ejemplo: Observa cómo el programa puede tratar dicha excepción(en nuestro caso "tratar" es imprimir el mensaje *muy, muy, muy mal: no se puede dividir por cero)* y a continuación puede proseguir la ejecución del programa, es decir, "se recuperó de la excepción".

```
class App {
   public static void main(String[] args) {
        int pesoPaquete;
        try{
           System.out.println("antes de que se genere la excepción");
           pesoPaquete=10/0;
           System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
            System.out.println("muy, muy, mu  mal:  no se puede dividir por cero");

        }

        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}
```

Ejemplo: El bloque try "controla" también las excepciones de los métodos llamados dentro su bloque. Ejecuta los siguientes ejemplos

```
class App {
  static void dividePorCero(){
      int pesoPaquete;
      pesoPaquete=10/0;
  }
   public static void main(String[] args) {
        try{
           System.out.println("antes de que se genere la excepción");
           dividePorCero();
           System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
           System.out.println("muy, muy, muy, mal no se puede dividir por cero");
        }
        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}
```

O por ejemplo

```
class Otra{
   void dividePorCero(){
      int pesoPaquete;
      pesoPaquete=10/0;
   }
}
class App {
```

```
   public static void main(String[] args) {
        try{
           System.out.println("antes de que se genere la excepción");
           Otra obj1= new Otra();
           obj1.dividePorCero();
           System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
           System.out.println("muy, muy, muy, mal no se puede dividir por cero");
        }
        System.out.println("el programa sigue su ejecución se recupero de la excepción ...");
   }
}
```

Esto lo entenderemos mejor cuando veamos el mecanismo de propagación de excepciones.

UN CATCH SÓLO TRATA SU TIPO DE EXCEPCIÓN

Ejemplo: En el siguiente ejemplo el catch "captura" una excepción aritmética, pero vamos a provocar una excepción de tipo NumberFormatException. Como la excepción "controlada" por el try no es del mismo tipo que la del catch, la trata la JVM(java virtual machine) con la correspondiente terminación anormal del programa. Observa que ahora no se llega a hacer el último println().

```
class App {
   public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=2;
        try{
           System.out.println("antes de que se genere la excepción");
           pesoPaquete= Integer.parseInt("10.5");
           System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
           System.out.println("muy, muy, muy, mal no se puede dividir por cero");
        }
        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}
```

En un ejemplo posterior veremos que también es posible especificar que un catch controle varios tipos de excepciones, pero nunca va a controlar tipos de excepciones que no se incluyen en la definición del catch.

UN TRY PUEDE ESTAR ASOCIADO A VARIOS CATCH

Ya que pueden producirse varios tipos de errores y un catch sólo captura su tipo de excepción, es lógico que un try pueda asociarse a más de un catch.

Ejemplo: En el siguiente ejemplo lógicamente no se llega a producir una división por cero y "salta" el catch asociado al NumberFormatException.

```
class App {
   public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=2;
        try{
           System.out.println("antes de que se genere la excepción");
           pesoPaquete=pesoPaquete/divisor;
           pesoPaquete= Integer.parseInt("10.5");
           System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
           System.out.println("muy, muy, muy, mal no se puede dividir por cero");
        }catch(NumberFormatException miExcepcion){
           System.out.println("imposible convertir en entero ese string");
        }
```

```
        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}
```

Ejemplo: En el siguiente ejemplo hay división por cero y por producirse primero esta excepción, es la que se trata. Recuerda que tras producirse una excepción se abandona la ejecución del resto de instrucciones del try.

```
class App {
    public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=0;
        try{
            System.out.println("antes de que se genere la excepción");
            pesoPaquete=pesoPaquete/divisor;
            pesoPaquete= Integer.parseInt("10.5");
            System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
            System.out.println("muy, muy, muy, mal no se puede dividir por cero");
        }catch(NumberFormatException miExcepcion){
            System.out.println("imposible convertir en entero ése string");
        }

        System.out.println("el programa sigue su ejecución se recupero de la excepción ...");
    }
}
```

Ejemplo: Varias excepciones en el mismo catch
Observa que si el tratamiento de varias excepciones es el mismo, se duplica código

```
class App {
    public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=2;
        try{
            System.out.println("antes de que se genere la excepción");
            pesoPaquete=pesoPaquete/divisor;
            pesoPaquete= Integer.parseInt("10.5");
            System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
            System.out.println("mi código es excepcional...");
        }catch(NumberFormatException miExcepcion){
            System.out.println("mi código es excepcional...");
        }

        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}
```

Mejor con la sintaxis

```
class App {
    public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=0;
        try{
            System.out.println("antes de que se genere la excepción");
            pesoPaquete=pesoPaquete/divisor;
            pesoPaquete= Integer.parseInt("10.5");
            System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException|NumberFormatException miExcepcion){
            System.out.println("mi código es excepcional....");
        }

        System.out.println("el programa sigue su ejecución se recupero de la excepción ...");
    }
}
```

Ejemplo: En el siguiente ejemplo me aseguro de tratar las dos excepciones, haciendo try separados. En este caso es lo que deseo que haga mi programa, técnicamente no tiene que ser mejor ni peor que el ejemplo anterior, eso depende del efecto que quiera conseguir con mi programa.

```
class App {
   public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=0;
        try{
           System.out.println("antes de que se genere la excepción division por cero ");
           pesoPaquete=pesoPaquete/divisor;
           System.out.println("esto jamás  se imprime");
        }catch(ArithmeticException miExcepcion){
           System.out.println("muy, muy, muy, mal no se puede dividir por cero");
        }
        try{
           System.out.println("antes de que se genere la excepción de formato");
           pesoPaquete= Integer.parseInt("10.5");
           System.out.println("esto jamás  se imprime");
        }catch(NumberFormatException miExcepcion){
           System.out.println("imposible convertir en entero ese string");
        }

        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}
```

Ejercicio U5_B9A_E1: Si al trabajar con un array x de tamaño n, el último elemento es x[n-1]. Si sobrepasamos los límites del array por ejemplo intentando usar x[n], java genera una excepción.

```
class App{
         public static void main(String[] args){
                  int[] x= {0,1,2,3,4};
                  x[5]=5;
                  System.out.println("El programa se recupera de la excepción y continua");
         }
}
```

reforma el código para que capture la excepción y el programa no rompa.

Ejercicio U5_B9A_E2: Queremos calcular el factorial de un número pero asegurándonos que el usuario introduce un entero por teclado. Hasta que no introduzca un entero le estaremos pidiendo repetitivamente que introduzca el entero.

Se pide añadir código a App

```
class App{
   public static void main(String[] args) {
      Scanner teclado= new Scanner(System.in);

      int n=0; //Numero entero introducido por teclado.

      //añadir código para obtener número entero correcto usando mecanismo excepciones


      //calculamos factorial
      int factorial=n;
```

```
        String salida=n+"! =  "+n;
        for(int i=n-1;i>0;i--){
            salida+="*"+i;
            factorial*=i;
        }
        salida+=" = "+factorial;
        System.out.println(salida);
    }
}
```

Y se pide que lo soluciones de dos formas:

- leyendo con nextInt() y por tanto manejando InputMismatchException. ¡Consulta el API de este método y observa que genera esta excepción!
- leyendo un String con nextLine() y luego traduciendo el String a entero con ParseInt() y por tanto manejando NumberFormatException ¡Consulta el API de este método y observa que genera esta excepción!