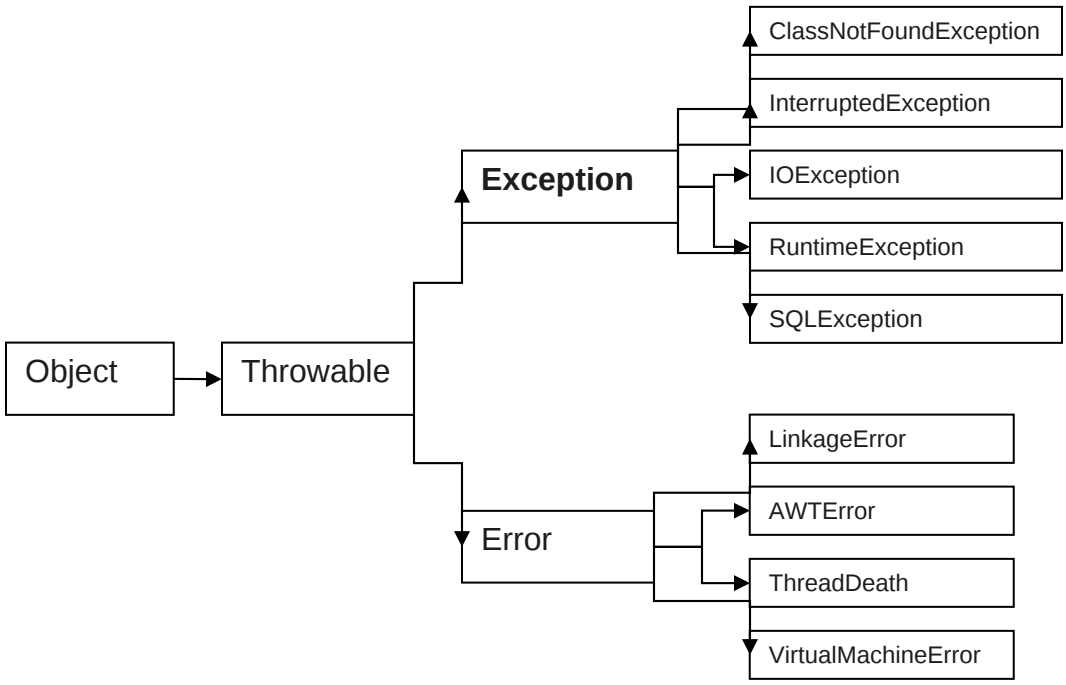
Jerarquía de de Errores y Excepciones

De la ejecución de nuestros programas va a surgir una serie de errores que debemos manejar de alguna manera. Para esto debemos determinar qué los provoca y, una vez definido eso, analizar qué camino tomar, evitar el error, mostrar un mensaje comprensible para el usuario, abortar la ejecución del programa, etc. Java define una excepción como un objeto que es una instancia de la clase **Throwable,** o alguna de sus subclases. Es decir que estamos ante una jerarquía de excepciones. Para comenzar, de Throwable heredan las clases **Error** y **Exception**

.



Por lo general, un error tiende a provocarse en condiciones anormales, eso significa que no tiene sentido o no es posible capturarlo. Por ejemplo, “OutOfMemoryError” indica que no hay más memoria. Debido a la imposibilidad de capturar los errores, el programa termina. Las excepciones, en cambio, se producen dentro de condiciones normales, es decir que no sólo es posible capturarlas, sino que también el lenguaje nos obliga a capturarlas y manejarlas.

Nunca olvidemos que una excepción no es un error, sino un evento que nuestro programa no sabe cómo manejar. Siempre que se provoque una excepción, podemos capturarla y manejarla, o evitarla cambiando la lógica del programa.

Control de Excepciones (try… catch… finally)

Para capturar y manejar las excepciones, Java proporciona las siguientes sentencias: **try, catch** y **finally**

Entre un try y un catch vamos a escribir el código de programa que pueda provocar una excepción. La estructura general es la siguiente:

```
try{
      // Acá va el bloque que puede provocar una excepción
}
catch(NombreDeLaExcepcion instancia){
      // Acá va el bloque que maneja la excepción en caso de producirse
}
```

En el catch va el nombre de la excepción que deseamos capturar. Si no sabemos qué excepción es, recordemos que hay una jerarquía de excepciones, de tal forma que se puede poner la clase padre de todas las excepciones: **Exception**

Veamos un ejemplo que produce una excepción, pero no vamos a capturarla, para analizar un poco la salida de error

```
 1 public class EjemploException {
 2
 3     private static int[] arreglo = new int[5];
 4
 5     public static void main(String[] args) {
 6         for(int x=0; x<5; x++){
 7             arreglo[x] = x*2;
 8         }
 9         MuestraArray();
10     }
11
12     public static void MuestraArray(){
13         for(int x=0; x<=5; x++){
14             System.out.println(arreglo[x]);
15         }
16     }
17 }
```

Al compilar y ejecutar el programa se obtiene lo siguiente:

```
0
2
4
6
8
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at ejemploexception.EjemploException.MuestraArray(EjemploException.java:14)
        at ejemploexception.EjemploException.main(EjemploException.java:9)
Java Result: 1
```

El programa no causa ningún error de compilación, pero sí un error de ejecución. Esto es un **RuntimeException**. Analizando el error vemos, primero, el nombre de la excepción provocada, en este caso **ArrayIndexOutOfBoundsException**, y a continuación, el mensaje de error. Luego de informar la excepción, muestra la línea exacta en que fue provocada. Es conveniente leer esto de abajo hacia arriba, de manera tal que podamos ver dónde empezó todo e ir siguiéndolo con nuestro código. Vemos que todo empezó en la línea 9, de paquete "ejemploexception", en la clase "EjemploException", en el método "main". Si vamos a esta línea vemos que es donde llamamos el método MuestraArray(). Seguimos hacia arriba, y ahora vemos que el error está en la línea 14 del paquete "ejemploexception", en la salida por pantalla, en esta línea es donde sacamos un elemento del arreglo, es entonces cuando determinamos cuál es la línea de código que provoca el error y por qué. Ahora debemos elegir una solución. Podemos cambiar el límite del `for` de 5 a 4, para asegurarnos de no querer sacar más elementos, o podemos capturar la excepción y manejarla. Para capturar la excepción debemos poner las líneas que provocaron la excepción entre un **try** y un **catch**

En el ejemplo anterior debemos encerrar al ciclo for del método MuestraArray():

```
 1 public class EjemploException {
 2
 3    private static int[] arreglo = new int[5];
 4
 5    public static void main(String[] args) {
 6        for(int x=0; x<5; x++){
 7            arreglo[x] = x*2;
 8        }
 9        MuestraArray();
10    }
11
12    public static void MuestraArray(){
13          try{
14                for(int x=0; x<=5; x++){
15                      System.out.println(arreglo[x]);
16                }
17          }
18          catch(ArrayIndexOutOfBoundsException ex){
19                System.out.print("Se intentó sacar más elementos");
20          }
21    }
22 }
```

Ahora la salida es la siguiente:

```
0
2
4
6
8
Se intentó sacar más elementos
```

Como vemos, lo único que logramos es mostrar el mismo error, sólo que de manera mas comprensible. En ciertos casos, vamos a usar el bloque **catch** para evitar el error, no sólo

para mostrarlo. El lenguaje no nos obliga a capturar las excepciones que se provocan en tiempo de ejecución, sin embargo, éstas se producen, y muy a menudo. Entonces, empezará ejecutando el **try**, si se produce alguna excepción, corta el **try** y ejecuta el **catch**. Finalmente, lo que va en el bloque **finally**, se ejecutará siempre, haya habido excepciones o no. Algo muy común es usar dicho bloque para limpiar variables, cerrar archivos, restablecer conexiones, es decir, tareas que hayan podido quedar truncadas según se haya ejecutado el **try** completo o no.

La estructura de un bloque con try, catch y finally es la siguiente:

```
try{
      // Acá va el bloque que puede provocar una excepción
}
catch(NombreDeLaExcepcion instancia){
      // Acá va el bloque que maneja la excepción en caso de producirse
}
finally{
      // Este bloque se ejecuta siempre sin importar si se ejecuto el
      // bloque del try o del catch
}
```

Excepciones Genéricas:

Podemos tener cuantos **catch** creamos necesarios, cada uno para capturar una excepción especifica, con la ventaja que podremos manejar cada una de ellas de manera distinta, pero puede suceder que como resultado de la ejecución del programa se provoque una excepción que el programa no estaba preparado para manejar. Aprovechando la jerarquía de excepciones, podemos reemplazar todos los **catch** de excepciones particulares por la superclase **Exception**, haciendo una captura genérica. Para tener una idea del problema ocurrido se pueden utilizar los objetos **getMessaje()** o **printStackTrace()**.

```
11
12    public static void MuestraArray(){
13          try{
14                for(int x=0; x<=5; x++){
15                      System.out.println(arreglo[x]);
16                }
17          }
18          catch(Exception ex){
19                ex.printStackTrace();
20          }
21    }
22 }
```

Aseveraciones (assert)

Las aseveraciones permiten comprobar las presunciones que haces sobre el código mientras se esta desarrollando, sin tener que manejar excepciones que se cree que nunca, ocurrirán.

Sintaxis:

```
assert (condición);
ó
assert (condición) : “Mensaje”
```

Supongamos que se escribe un código según el cual si x>0, sucede una cosa, y si x==0, sucede otra. Se puede expresar de la siguiente manera:

```
if (x > 0){
     // Código en caso que se cumpla la condición
}
else{
     // Código en caso que NO se cumpla la condición
}
```

Se supone que nunca sucederá que **x<0**; pero, ¿y si algo va mal y resulta que x tiene un valor negativo? En este caso, se ejecutaría la parte de código correspondiente a else, como si **x==0**, con resultados impredecibles.

Para evitarlo, se puede usar una aseveración:

```
if (x > 0){
     // Código en caso que se cumpla la condición
}
else{
     assert (x == 0);
     // Código en caso que NO se cumpla la condición
}
```

Activación de las Aseveraciones en NetBeans 7.0:

"File" → "Project Properties (nombre del proyecto)"
Elegir la opción “Run”, y en “VM Options” agregar el siguiente parámetro “-enableassertions” (sin comillas)