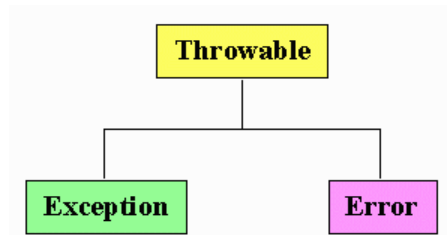


LA JERARQUÍA DE CLASES PARA EL MANEJO DE EXCEPCIONES. IMPLICACIONES.

Las excepciones son una jerarquía de clases, donde la parte más alta de esta jerarquía es:



Ocurre que una referencia de tipo Throwable puede almacenar referencias a cualquier tipo de excepción. Esto es posible debido a la siguiente propiedad de las variables referencia:

una referencia de una superclase puede almacenar una referencia a un objeto de tipo subclase. ¡Ya lo sabíamos!

Ejemplo: Ahora una excepción *NumberFormatException* es tratada por el catch con una referencia de tipo superclase Throwable.

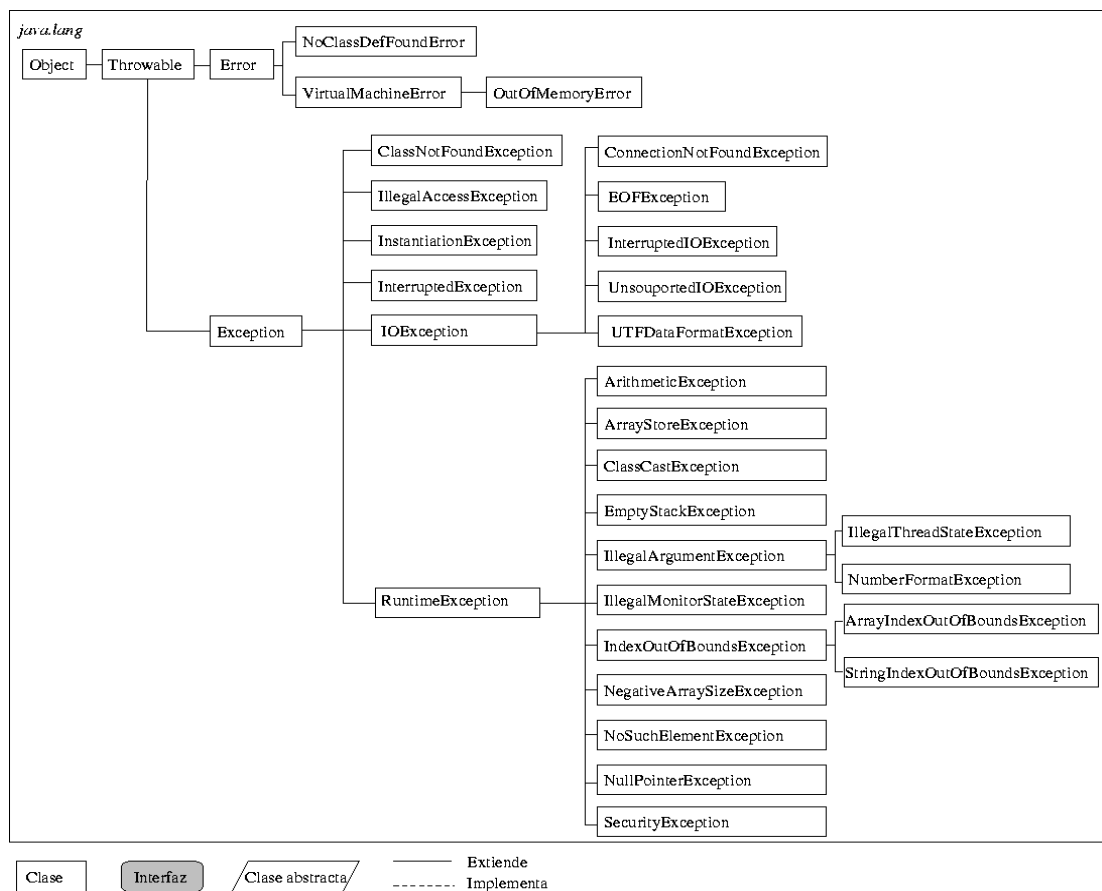
```
class App {
    public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=0;
        try{
            System.out.println("antes del error ");
            pesoPaquete= Integer.parseInt("10.5");
            pesoPaquete=pesoPaquete/divisor;
            System.out.println("esto jamás se imprime");
        }catch(ArithmeticException miExcepcion){
            System.out.println("muy, muy, muy, mal no se puede dividir por cero");
        }catch(Throwable miExcepcion){
            System.out.println("algo va mal");
        }
        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}
```

Ejemplo: En caso de colocar el catch de la referencia superclase en primer lugar, se produce un error de compilación ya que no tiene sentido colocar los catch de subclases después del de superclase, los catch de subclases jamás se ejecutarán, por que "toda excepción" encaja con el primer catch. Comprueba el error.

```
class App {
    public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=0;
        try{
            System.out.println("antes del error ");
            pesoPaquete= Integer.parseInt("10.5");
            pesoPaquete=pesoPaquete/divisor;
            System.out.println("esto jamás se imprime");
        }catch(Throwable miExcepcion){
            System.out.println("algo va mal");
        }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 ...");
    }
}
```

```
}
}
```

Ya se explicó que las excepciones que se pueden generar en nuestros programas son siempre de la clase `Exception`, que es subclase de `Throwable`. Observa el siguiente gráfico un poco más detallado de la jerarquía de clases de excepciones:



Por tanto, ya que todas las anomalías con las que vamos a trabajar son subclases de `Exception`, lo usual es declarar un `catch` con variable referencia tipo `Exception`. Reescribimos por tanto el ejemplo anterior a su forma más común que usa la clase `Exception` el lugar de `Throwable`. **ES MEJOR NO USAR NUNCA `Throwable`** pues el programa también capturaría excepciones de la clase `Error` ante las que no tiene nada que hacer y dejará funcionando mal la MVJ perjudicando a otras aplicaciones.

```

class App {
    public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=0;
        try{
            System.out.println("antes del error ");
            pesoPaquete= Integer.parseInt("10.5");
            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");
        }catch(Exception miExcepcion){
            System.out.println("algo va mal");
        }
    }
}
  
```

```

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

```

USAR LA VARIABLE REFERENCIA DEL CATCH

Por el momento la referencia del catch sólo la utilizamos para que JVM eligiera el catch apropiado, pero lógicamente, tenemos referenciada una excepción, que es un objeto, y dicho objeto tendrá asociados métodos y atributos. Por ejemplo, como dicha excepción siempre es subclase de throwable y uno de los métodos de throwable es getMessage(), puedo utilizar dicho método.

Ejemplo: probamos getMessage()

```

class App {
    public static void main(String[] args) {
        int pesoPaquete=10;
        int divisor=0;
        try{
            System.out.println("antes del error ");
            pesoPaquete= Integer.parseInt("10.5");
            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");
        }catch(Exception miExcepcion){
            System.out.println("algo va mal veamos lo que dice getMessage(): "+ miExcepcion.getMessage());
        }
        System.out.println("el programa sigue su ejecución se recuperó de la excepción ...");
    }
}

```

Observa que utilizo como es habitual la clase Exception, pero esta hereda de Throwable, entre otras cosas, el método getMessage(). Consulta esto en el API JAVA.

Vuelve a ejecutar el programa comentando

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

Ahora en lugar de getMessage() indicamos nuestro mensaje personalizado cuando se produce la división por cero.

Para obtener las descripciones del sistema de los errores producidos, se usan mucho dos métodos:

- o getMessage(), devuelve un String con un mensaje descriptivo
- o printStackTrace(), devuelve lo mismo que getStackTrace() pero esa info la imprime en la salida estándar de error(pantalla normalmente).

Ejercicio: En el ejemplo anterior, sustituye getMessage() por printStackTrace(), observa que no precisas en este caso System.out.println(); Sigue la traza de printStackTrace hasta llegar a la línea de código que originó la excepción.

LA OBLIGACIÓN DE TRATAR LAS EXCEPCIONES

Hasta ahora no nos vimos obligados a tratar una excepción, simplemente, si una excepción ocurría y no la tratábamos, la JVM describe la excepción y finaliza la ejecución del programa. Ahora vamos a analizar excepciones que sí requieren obligatoriamente tratamiento, de forma que, todo método que puede generar una excepción debe o bien manejarla con try/catch o bien declararla en una cláusula throws

Ejemplo: Observa el error para el siguiente código. El código pretende leer un carácter de teclado.

```
class App {  
    public static void main(String[] args) {  
        char letra;  
        System.out.print("Escriba una letra: " );  
        letra= (char) System.in.read();  
    }  
}
```

el compilador informa:

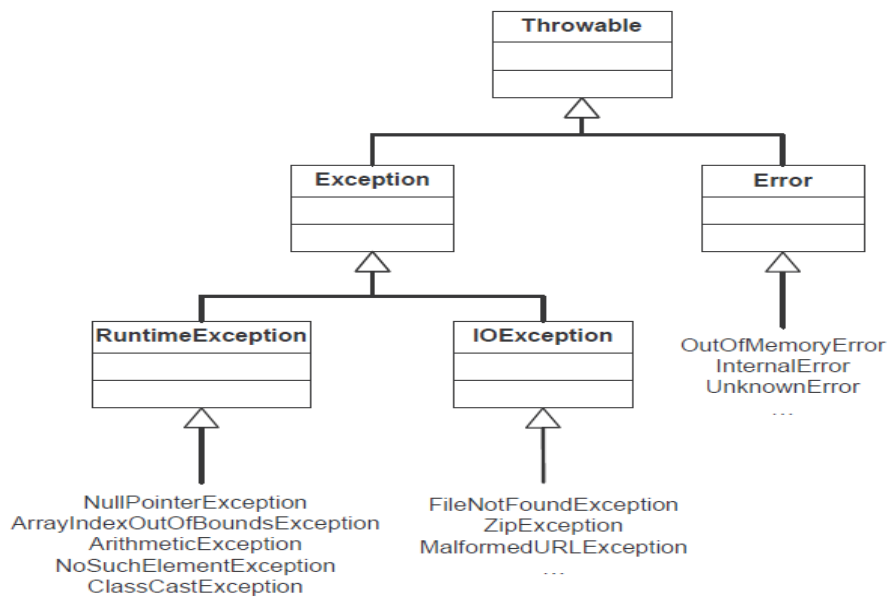
unreported java.io.IOException: must be caught or declared to be thrown.

Es decir como el método read() puede provocar una excepción, el método main() se tiene que responsabilizar de dicha excepción pudiendo hacer dos cosas:

1. Manejarla con try/catch
2. declararla en una cláusula throws

la excepción IOException

Como vimos en el mensaje del compilador, la excepción producida es de la clase IOException que pertenece al paquete java.io. IOException es una superclase de la que cuelgan otras clases parte de las cuales indicamos como ejemplo en el siguiente gráfico.



La E/S en java es compleja, veremos más cosas más adelante, por el momento asumimos que el método read() puede provocar una excepción del tipo indicado y debemos tratarla.

manejarla con try / catch

Ejemplo: observa cómo desaparece el error anterior

```
class App {  
    public static void main(String[] args) {  
        try {  
            char letra;  
            System.out.print("Escriba una letra: ");  
            letra = (char) System.in.read();  
        }  
    }  
}
```

```

        System.out.println("adios");
    }catch (java.io.IOException ex) {
        System.out.println("mucho ojo con las excepciones de entrada salida");
    }
}
}

```

Las excepciones de entrada/salida se provocan cuando surge un fallo o una interrupción en la entrada/salida. En el caso de la entrada/salida con el teclado, es algo muy interno y no es fácil provocar un excepción, podemos forzarlo cerrando el flujo System.in antes de leer

```

class App {
    public static void main(String[] args) {
        try {
            char letra;
            System.out.print("Escriba una letra: ");
            System.in.close();//para provocar excepción
            letra = (char) System.in.read();

            System.out.println("adios");
        }catch (java.io.IOException ex) {
            System.out.println("\nmucho ojo con las excepciones de entrada salida");
        }
    }
}

```

La palabra clave throws ("cláusula throws")

Ejemplo: observa cómo desaparece el error anterior

```

class App {
    public static void main(String[] args) throws java.io.IOException{
        char letra;
        System.out.print("Escriba una letra: ");
        letra = (char) System.in.read();
        System.out.println("adios");
    }
}

```

throws indica que se propague la excepción.

En un método, throws se utiliza para indicar que si se produce una excepción, dicho método no la va a manejar y la propaga, es decir, throws traspasa la responsabilidad al método que llamó al actual. En nuestro caso, a main() lo llamó JVM, y por tanto la excepción pasaría a ser tratada por la propia JVM. Veamos ejemplos para entender mejor esto.

Ejemplo: El siguiente código tiene un error de compilación

```

class App {
    public static char aviso (){
        System.out.print("Escriba una letra: ");
        return (char) System.in.read();
    }
    public static void main(String[] args) {
        aviso();
        System.out.println("adios");
    }
}

```

read() genera una IOException y la propaga a aviso(). Aviso() no hace nada para tratar la excepción y se genera un error de compilación

Ejemplo: `aviso()` propaga la excepción a quien lo llama, el método `main()`, y por tanto ahora el error de compilación surge en el `main()`

```
class App {
    public static char aviso () throws java.io.IOException{
        System.out.print("Escriba una letra: ");
        return (char) System.in.read();
    }
    public static void main(String[] args) {
        aviso();
        System.out.println("adios");
    }
}
```

Ejemplo: Observa que desaparece el error si `main()` propaga la excepción

```
class App {
    public static char aviso () throws java.io.IOException{
        System.out.print("Escriba una letra: ");
        return (char) System.in.read();
    }
    public static void main(String[] args) throws java.io.IOException{
        aviso();
        System.out.println("adios");
    }
}
```

Ejemplo: Observa que desaparece el error si `main` captura(catch) la excepción.

```
class App {
    public static char aviso () throws java.io.IOException{
        System.out.print("Escriba una letra: ");
        return (char) System.in.read();
    }
    public static void main(String[] args) {

        try{
            aviso();
        }catch (java.io.IOException miexec) {
            System.out.println("algo va mal en la E/S");
        }
        System.out.println("adios");
    }
}
```

Ejemplo: lavado de manos, normalmente no conveniente.

Observa que si en el `main()` hago una propagación para la clase `Exception`, de la forma *`public static void main(String[] args) throws Exception`*

```
class App {
    public static char aviso () throws Exception{
        System.out.print("Escriba una letra: ");
        return (char) System.in.read();
    }
    public static void main(String[] args) throws Exception{
        aviso();
        System.out.println("adios");
    }
}
```

“me lavo las manos” y traspaso toda excepción, sea del tipo que sea (aritmética, de E/S, etc.) a la JVM, pero, ipierdo la oportunidad de tratar una excepción y de que mi programa se

recupere de ella!

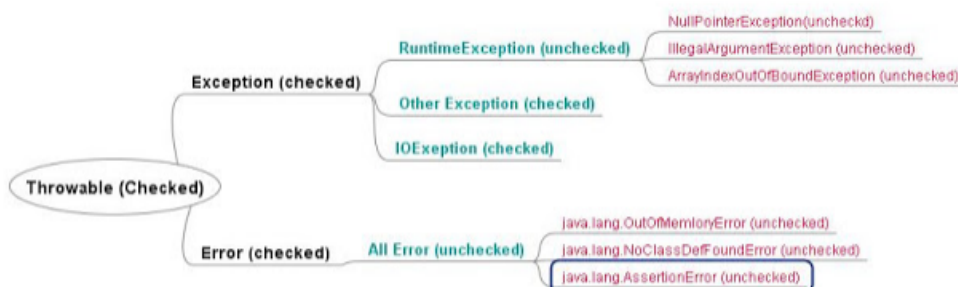
Excepciones marcadas y no marcadas.

Si pensamos en los ejemplos iniciales, al trabajar con *ArithmeticException* o *NumberFormatException* no nos daba el error de compilación *must be caught or declared to be thrown*, es decir, el compilador no nos obligaba a tratar/propagar las excepciones.

Bajo este punto de vista tenemos dos tipos de excepciones:

- Excepciones *marcadas*(*checked*): aquellas cuya captura/declaración es obligatoria (si no lo hacemos ya obtenemos un error en tiempo de compilación). A estas excepciones también se le llaman *declarativas* ya que los métodos que pueden provocarlas deben declararlas con la cláusula *throws*. También se llaman *verificadas* ya que cuando escribimos un método el compilador verifica si este método trabaja con una excepción declarativa. Estas excepciones son todas aquellas que no tienen de superclase a *RuntimeException* o *Error*.
- Excepciones *no marcadas*(*unchecked*): no es necesaria su captura/declaración. Si durante la ejecución del programa se produce una excepción de este tipo la JVM provoca una finalización inmediata del programa y envía a la salida de error un volcado de pila que informa del error. Son todas las excepciones de superclase *RuntimeException* y *Error*. También se llaman no declarativas y no verificadas.

Muchas de las excepciones *unchecked* están relacionadas con los problemas que ocasiona "el código mal escrito" que provoca los típicos errores lógicos de programación que hacen saltar excepciones como *NullPointerException*, *ArrayIndexOutOfBoundsException*, etc. Estos errores son previsibles por el programador y se prefiere "prevenir que tratar" tanto por claridad como por eficiencia (una solución con excepciones es más costosa que usar un simple *if*). Por ejemplo, si trabajamos con un array de tamaño 3, el programador antes de indexar el array tendrá que comprobar que el índice es menor que 3. Como conclusión, en general, se debe evitar el tratamiento de estas excepciones *unchecked*, aunque en este boletín veamos muchos ejemplos que si lo hacen.



Ejemplo: las excepciones no verificadas puede evitarse normalmente

```
public class App{
    public static char devuelvePrimerCaracter(String s){
        if(s==null){
            return '\0';
        }else{
            return s.charAt(0);
        }
    }
}

public static void main(String args[]){
    String s=null;
    System.out.println(devuelvePrimerCaracter(s));
    s="hola";
    System.out.println(devuelvePrimerCaracter(s));
}
```

```
}
}
```

El método `devuelvePrimerCaracter()` tiene un referencia como parámetro, y por tanto, uno de los posibles valores puede ser `null`. El método `devuelvePrimerCaracter()` tiene la obligación “de calidad” de tener esto en cuenta. También pudo controlarse con `try/catch` pero es más ineficiente y no aporta nada (en principio)

```
public class App{
    public static char devuelvePrimerCaracter(String s){
        char x;
        try{
            x= s.charAt(0);
        }catch(NullPointerException e){
            x='\0';
        }
        return x;
    }

    public static void main(String args[]){
        String s=null;
        System.out.println(devuelvePrimerCaracter(s));
        s="hola";
        System.out.println(devuelvePrimerCaracter(s));
    }
}
```

CREACIÓN DE SUBCLASES DE EXCEPCIONES

Hasta ahora estuvimos utilizando subclases de `Exception` integradas en la librería del `jdk`, pero es posible crear nuestras propias excepciones personalizadas creando una subclase de la superclase `Exception`. Antes de ver un ejemplo necesitamos conocer el uso de la instrucción `throw`.

La instrucción `throw` lanza una excepción.

Para lanzar una excepción, hay que:

1. Crear un objeto de tipo `Exception`(o subclase de `Exception`) con `new`
2. Lanzar la excepción con la instrucción `throw`

Se pueden crear objetos de un tipo de excepción con `new`

```
class App {
    public static void main(String[] args) {
        ArithmeticException e = new ArithmeticException();
    }
}
```

Si ejecutas el `main` anterior no ocurre nada ya que tal y como escribimos tenemos un objeto de tipo `ArithmeticException` “suelto”, es decir, aislado del mecanismo de propagación/captura. Además de crear una excepción(`new`) ¡hay que lanzarla (`throw`)!

```
class App {
    public static void main(String[] args) {
        throw new ArithmeticException();
    }
}
```

Podemos lanzar la excepción que queramos por ejemplo:

```
class App {
    public static void main(String[] args) {
        throw new NumberFormatException();
    }
}
```



```
}  
}
```

Recuerda que el método `Integer.parseInt()` también lanzaba una excepción *NumberFormatException* así que debe contener dentro de su código al menos la instrucción

```
throw new NumberFormatException();
```

Puedes comprobarlo observando el código fuente en google "java source Integer" y luego con la opción de buscar del navegador buscamos `throw new NumberFormatException();`

Allí observamos que `parseInt` llama a otra versión `se parseInt()`

```
public static int parseInt(String s){  
    return parseInt(s, 10, false);  
}
```

Y en esta otra versión observamos cómo su cuerpo lanza las *NumberFormatException*

```
static int parseInt(String str, int radix, boolean decode)
```

```
{  
    if (! decode && str == null)  
        throw new NumberFormatException();  
    int index = 0;
```

Ejemplo: un método lanza una excepción y el mismo la captura

Cuando un método lanza una excepción la propaga. También podría capturarla, aunque no tiene mucho sentido ya que se avisa a sí mismo, lo que es una estupidez. De todas formas observamos la posibilidad.

```
import java.io.IOException;
```

```
class App {  
    public static void main(String[] args) {  
        try {  
            throw new IOException();  
        } catch (IOException ex) {  
            System.out.println("main a lanzado y capturado una IOException");  
        }  
    }  
}
```

Ejemplo: un método lanza una excepción y la propaga

Lo habitual es la combinación `throw + throws`. Es decir, un método genera una excepción para avisar de una anomalía a otro método que lo llamó y por tanto tras generarla (`throw + new`) la propaga (`throws`). En este mini ejemplo `main` propaga a `jvm`

```
import java.io.IOException;
```

```
class App {  
    public static void main(String[] args) throws IOException {  
        throw new IOException();  
    }  
}
```

Ejemplo: después de `throw` se detiene la ejecución normal.

Suponiendo que el método propaga (throws) la excepción una vez que se produce se abandona el método. Observa el error de compilación de la sentencia inalcanzable.

```
import java.io.IOException;

class App {
    public static void main(String[] args) throws IOException {
        throw new IOException();
        System.out.println("Jamás se imprime");
    }
}
```

No tiene sentido el println() detrás del throw . Dada la lógica del flujo de programa anterior se sabe al 100% que jamás se va a ejecutar System.out.println("jamás se imprime");

Normalmente el throw siempre está dentro de un if para que tenga sentido que haya otras instrucciones adicionales "debajo" del throw. Ahora hay algo debajo del throw pero si se puede alcanzar, dependiendo del valor de x.

```
import java.io.IOException;

class App {
    public static void main(String[] args) throws IOException {
        int x;
        //x=10;
        x=0;
        if(x==0){
            throw new IOException();
        }
        System.out.println("X vale: "+x);
    }
}
```

Excepciones personalizadas

Podemos crear nuestras propias excepciones creando una clase que extiende a Exception. Una clase que extiende a *Exception* hereda la capacidad de ser un objeto lanzable con throw , propagable con throws y capturable con try/catch.

```
class DivisionInexactaException extends Exception{
    int n;
    int d;
    DivisionInexactaException(int i, int j){
        n=i;
        d=j;
    }
    String mensaje(){
        return "El resultado de dividir " + n + " entre " + d + " no es un exacto";
    }
}

class RacionalExacto{
    int numerador;
    int denominador;
    RacionalExacto(int numerador,int denominador) throws DivisionInexactaException{
        if(numerador%denominador!=0)
            throw new DivisionInexactaException(numerador,denominador);
        this.numerador=numerador;
        this.denominador=denominador;
    }
}

class App {System.out.println(exc.mensaje());System.out.println(exc.mensaje());System.out.println(exc.mensaje());
    public static void main(String[] args) {
        RacionalExacto r1=null;
        RacionalExacto r2=null;
        try{
            r1 = new RacionalExacto(10,2);
            System.out.println("NUMERO RACIONAL EXACTO O.K.");
            r2 = new RacionalExacto(10,3);
            System.out.println("NUMERO RACIONAL EXACTO O.K.");
        }
    }
}
```

```

    }catch(DivisionInexactaException exc){
        //aquí instrucciones con r si el Racional NO es exacto
        System.out.println(exc.mensaje());
    }
    System.out.println("r1: "+r1);
    System.out.println("r2: "+r2);
}
}

```

Observa cómo r2 vale al final del código null ya que al interrumpirse la ejecución del constructor del segundo objeto, este no se llegó a crear

Ejercicio: elimina el `extends Exception` y observa como no se puede usar *DivisionInexactaException* en `throw` y `catch`

finally

Si hay un `try/catch` o incluso un `try` sin `catch`, podemos añadir un tercer bloque de instrucciones llamado **finally**. Este bloque incluye las instrucciones que siempre queremos que se ejecuten al finalizar un `try` se produjera excepción o no. Finally será importante entre otras situaciones cuando se trabaja con la E/S de java. Por el momento veamos un ejemplo:

```

class UsoFinally{
    static void generarExcepcion(int paraSwitch){
        int t=10;

        System.out.println("Recibiendo " + paraSwitch);
        try{
            switch(paraSwitch){
                case 0:
                    t=10/paraSwitch;//forzamos division por zero
                    break;
                case 1:
                    t=Integer.parseInt("10.5");//provocamos error de formato
                    break;
                case 2:
                    break;
            }
        }catch(ArithmeticException exc){
            System.out.println("no se puede dividir por cero ¡animal!");
        }catch(NumberFormatException miExcepcion){
            System.out.println("imposible convertir en entero ese string");
        }finally{
            System.out.println("dejando try");
            System.out.println();
        }
    }
}

class App {
    public static void main(String[] args) {
        for(int i=0;i<3;i++){
            UsoFinally.generarExcepcion(i);
        }
    }
}

```

En la salida del programa observamos que siempre veremos "dejando try", es decir que una vez que se pasa por el `try`, sea como sea, jvm siempre ejecutará el `finally`.

Un último ejemplo. Como sabes cuando en la ejecución de un método se encuentra la JVM una instrucción `return`, le estamos indicando en ese punto, finaliza la ejecución del método.

Vamos a introducir return en generarExcepción para observar que incluso se sigue ejecutando el finally antes de abandonar la ejecución del método.

```
class UsoFinally{
    static void generarExcepcion(int paraSwitch){
        int t=10;

        System.out.println("Recibiendo " + paraSwitch);
        try{
            switch(paraSwitch){
                case 0:
                    t=10/paraSwitch;//forzamos division por zero
                    break;
                case 1:
                    t=Integer.parseInt("10.5");//provocamos error de formato
                    break;
                case 2:
                    return;
            }
        }catch(ArithmeticException exc){
            System.out.println("no se puede dividir por cero ¡animal!");
            return;
        }catch(NumberFormatException miExcepcion){
            System.out.println("imposible convertir en entero ese string");
        }finally{
            System.out.println("dejando try");
            System.out.println();
        }
    }
}

class App {
    public static void main(String[] args) {
        for(int i=0;i<3;i++){
            UsoFinally.generarExcepcion(i);
        }
    }
}
```

Pero... ante un System.exit(0) nada se resiste

```
class UsoFinally{
    static void generarExcepcion(int paraSwitch){
        int t=10;

        System.out.println("Recibiendo " + paraSwitch);
        try{
            switch(paraSwitch){
                case 0:
                    System.exit(0);
                    t=10/paraSwitch;//forzamos division por zero
                    break;
                case 1:
                    t=Integer.parseInt("10.5");//provocamos error de formato
                    break;
                case 2:
                    return;
            }
        }catch(ArithmeticException exc){
            System.out.println("no se puede dividir por cero ¡animal!");
            return;
        }catch(NumberFormatException miExcepcion){
            System.out.println("imposible convertir en entero ese string");
        }finally{
            System.out.println("dejando try");
            System.out.println();
        }
    }
}
```

```

    }
}
class App {
    public static void main(String[] args) {
        for(int i=0;i<3;i++){
            UsoFinally.generarExcepcion(i);
        }
    }
}

```

Throws vs catch

Supongamos el siguiente código en el que metodoA() genera una excepción. La excepción puede propagarse y llegar hasta App y allí hacerle un catch o bien podemos capturar la excepción en un punto intermedio

```

class A {
    public void metodoA(){
        throw new Exception("el origen es metodoA");
    }
}
class B {
    public void metodoB(){
        A a = new A();
        a.metodoA();
    }
}
class C {
    public void metodoC(){
        B b = new B();
        b.metodoB();
    }
}
class App{
    public static void main(String[] args){
        C c = new C();
        c.metodoC();
    }
}

```

Como verás en los ejercicios que viene a continuación en este boletín la situación más habitual es propagar (throws) la excepción hasta llegar al método más externo (el main() en nuestro ejemplo) y allí capturarla. Dejamos aquí el modelo habitual y en los ejercicios se puede razonar porque este esquema es normalmente el más deseable, pero hay todo tipo de situaciones.

```

import java.util.logging.Level;
import java.util.logging.Logger;

class A {
    public void metodoA() throws Exception{
        throw new Exception("el origen es metodoA");
    }
}
class B {
    public void metodoB() throws Exception{
        A a = new A();
        a.metodoA();
    }
}
class C {
    public void metodoC() throws Exception{
        B b = new B();
        b.metodoB();
    }
}
class App{
    public static void main(String[] args){
        C c = new C();
    }
}

```

```

    try {
        c.metodoC();
    } catch (Exception ex) {

    }
}
}
}

```

EJERCICIOS

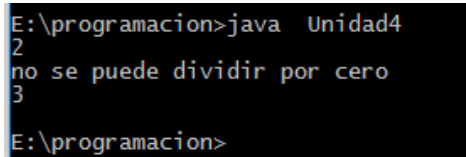
Ejercicio U5_B8B_E1: Añadir código de forma que el método dividir() si detecta una división por cero avisa a main() lanzándole un objeto de tipo Exception. Esto también provocará cambios en el main(). Observa que la excepción es del tipo de la superclase Exception. Se usa mucho para contextos sencillos como este.

```

class App{
    public static void main(String[] args){
        System.out.println(dividir(4,2));
        System.out.println(dividir(4,0));
        System.out.println(dividir(6,2));
    }
    static int dividir(int x,int y) {
        int cociente;
        cociente=x/y;
        return cociente;
    }
}

```

Al reformar el código debes de obtener la siguiente salida



```

E:\programacion>java Unidad4
2
no se puede dividir por cero
3
E:\programacion>

```

Ejercicio U5_B8B_E2: En el siguiente ejercicio se aprecia con claridad el mecanismo de la propagación de excepciones. Tienes que entenderlo muy bien.

Al siguiente código, de un ejercicio antiguo

```

abstract class Figura {
    protected String color;

    public Figura(String color) {
        this.color = color;
    }

    abstract public double area();
}

class Triangulo extends Figura {
    private double base;
    private double altura;

    public Triangulo(double base, double altura, String color) {

```

```

        super(color);
        this.base = base;
        this.altura = altura;
    }

    @Override
    public double area() {
        return base * altura / 2;
    }
}

class Circulo extends Figura {

    private double radio;

    public Circulo(double radio, String color) {
        super(color);
        this.radio = radio;
    }

    @Override
    public double area() {
        return Math.PI * radio * radio;
    }
}

```

Añádele la capacidad de generar una excepción cuando se genere una figura de color blanco de forma que funcione el siguiente main.

```

class App {

    public static void main(String[] args) {
        try {
            Circulo c = new Circulo(2.0, "blanco");
            System.out.println("Area circulo " + c.area());
        } catch (Exception e) {
            //aquí podrían ir instrucciones para rectificar el color como corresponda
            //o simplemente avisamos que no puede crear el objeto...
            System.out.println("NO SE PUDO CREAR OBJETO: " + e.getMessage());
        }
        try {
            Triangulo t = new Triangulo(2.0, 3.0, "rojo");
            System.out.println("Area triangulo " + t.area());
        } catch (Exception e) {
            System.out.println("NO SE PUDO CREAR OBJETO" + e.getMessage());
        }
    }
}

```

Observa que para avisar de la excepción como no creamos una excepción personalizada podemos usar simplemente Exception, indicando el mensaje descriptivo en el constructor

```
throw new Exception("color blanco no válido")
```

Ejercicio U5_B8B_E3:

En el ejercicio anterior utilizamos la clase Exception. Es habitual utilizar directamente la clase Exception para casos simples como el anterior. Para complicadas jerarquías, es más habitual utilizar excepciones personalizadas, tal y como trata de hacer el siguiente ejemplo con FiguraException.

Para incluir un mensaje descriptivo en una excepción como FiguraException puedo usar uno de estos métodos:

1. Indicarlo en parámetro de super(), o sea, super("bla bla")
2. sobrescribir el método getMessage()

Ejercicio U5_B8B_E4:

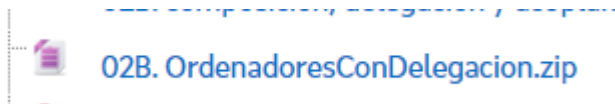
Escribe de nuevo el siguiente código, respetándose tal cual, pero añadiendo la capacidad de trabajar con excepciones (clase Exception) de forma que cuando se crea un rectángulo cuyo origen es un punto con alguna coordenada negativa se lanza una Exception.

```
class Punto {
    int x = 0;
    int y = 0;
    Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
class Rectangulo {
    Punto origen;
    int ancho ;
    int alto ;
    Rectangulo(int x, int y, int w, int h) {
        origen = new Punto(x,y);
        ancho = w;
        alto = h;
    }
}
class App{
    public static void main(String[] args){
        Rectangulo miRectangulo=new Rectangulo(-2,3,4,5);
    }
}
```

AL VOLVER A ESCRIBIR EL CÓDIGO ANTERIOR TEN EN CUENTA:

- El control del valor de las coordenadas debe ser realizado **obligatoriamente** en el constructor Punto(int x, int y) que será el encargado de generar una excepción de la clase Exception() avisando que hay una coordenada negativa.
- Debes propagar convenientemente la excepción y tratarla en el main(). imprimiendo la descripción de la excepción con getMessage()

Ejercicio U5_B8B_E5: Usaremos el proyecto Ordenadores que está en la carpeta soluciones



SE PIDE:

- El constructor de Ordenador se encarga de lanzar una excepción de tipo OrdenadorException cuando se intenta crear un Ordenador en las siguientes situaciones:
 - se intenta configurar un ordenador con procesador modelo I7 sin disco tipo SSD
 - el número de serie comienza por HP y tiene menos de 4gb de ram. (Busca en API método de clase String que devuelve comienzo de String)
- Un objeto OrdenadorException recibe en su constructor un número de serie y un mensaje y reescribe con esta info getMessage()
- Funciona el siguiente main

```
//Principal.java
package miaplicacion;
import ordenador.Ordenador;
```



```

import ordenador.OrdenadorException;

class Principal{
    public static void main(String[] args) {

        try{
            new Ordenador("DELL122",8,"DDR2",533,(float)2.0,"MAGNETICO","i7",(float)3.3,400);
            System.out.println("ORDENADOR DELL122 OK");
        }catch(OrdenadorException e){
            System.out.println(e.getMessage());
        }
        try{
            new Ordenador("CLONIC900",8,"DDR2",533,(float)2.0,"SSD","i7",(float)3.3,400);
            System.out.println("ORDENADOR CLONIC900 OK");
        }catch(OrdenadorException e){
            System.out.println(e.getMessage());
        }
        try{
            new Ordenador("HP511",2,"DDR2",533,(float)2.0,"SATA","i5",(float)3.3,400);
            System.out.println("ORDENADOR HP511 OK");
        }catch(OrdenadorException e){
            System.out.println(e.getMessage());
        }
    }
}

```

Que genera la salida:

```

DELL122: i7 sin SSD no se monta
ORDENADOR CLONIC900 OK
HP511: Serie HP no puede tener menos de 4gb de memoria

```

Ejercicio U5_B8B_E6: Añade en el ejercicio anterior una subclase *Portatil* que extienda a *Ordenador*. Un portatil tiene un atributo peso. Si se crea un objeto *Portatil* cuyo peso sea mayor de 10.0 kg se genera una excepción que indica que se rebasó el límite de peso. Funciona el siguiente main

```

//Principal.java
package miAplicacion;
import ordenador.Portatil;
class Principal{
    public static void main(String[] args) {
        try{
            //primer portatil no genera excepcion
            new Portatil("msi999",8,"DDR2",533,(float)2.0,"SSD","i7",(float)3.3,400,(float)7.5);
            System.out.println("ORDENADOR msi999 OK");

            //genera excepción por peso
            new Portatil("DELL122",8,"DDR2",533,(float)2.0,"SSD","i7",(float)3.3,400,(float)11.0);
            System.out.println("ORDENADOR DELL122 OK");

        }catch(ordenador.OrdenadorException e){
            System.out.println(e.getMessage());
        }
    }
}

```

run:
ORDENADOR msi999 OK
DELL122: portatil de más de 10 kg no se monta
BUILD SUCCESSFUL (total time: 0 seconds)

Ejercicio U5_B8B_E7:

En España el DNI fue creado por Franco, así que como buen dictador se puso a sí mismo el número 1, y a su mujer(Carmen Polo) el número 2, ...

Inicialmente el DNI era un número. Hoy en día consiste en 8 dígitos y una letra. Si quisieramos controlar NIFs y NIEs habría más combinaciones que no contemplamos en este ejercicio.

Fijate como es el algoritmo de validación de DNI

<https://www.interior.gob.es/opencms/ca/servicios-al-ciudadano/tramites-y-gestiones/dni/calculo-del-digito-de-control-del-nif-nie/>

SE PIDE: Modificar el siguiente ejemplo para que en lugar de Exception **utilice dni.DNIException**

```
//DNI.java
package dni;
public class DNI {
    private String dni;//8 digitos + letra
    private static final String LETRAS_DNI = "TRWAGMYFPDXBNJZSQVHLCKE";

    public DNI(String dni) throws Exception {
        //suponemos que un dni ES CORRECTO SI tiene 8 digitos + una letra. Sin guiones.
        //comprobamos si llega String null
        if (dni == null) {
            throw new Exception("dni con valor nulo");
        }
        //comprobamos si la longitud del string no es exactamente 9
        if (dni.length() != 9) {
            throw new Exception("la longitud de "+ dni +" no es de 9 caracteres");
        }

        //comprobamos que último caracter es letra
        //y la pasamos a mayúscula si no estuviera para simplificar ifs
        char letraDni = dni.charAt(dni.length() - 1);
        letraDni=Character.toUpperCase(letraDni);

        if (!Character.isLetter(letraDni)) {
            throw new Exception("el último caracter de "+ dni +" no es letra ");
        }

        //comprobamos que los 8 primeros caracteres son números
        String parteNumero = dni.substring(0, dni.length() - 1);
        if (!esNumero(parteNumero)) {
            throw new Exception("los 8 primeros caracteres de "+ dni +" no son todos números");
        }

        //compruebo el algoritmo de la letra que detecta si hay error tanto en letra como en número
        char letraCorrecta = calcularLetra(parteNumero);
        if (letraDni != letraCorrecta) {
            throw new Exception(" el dni "+ dni +" no cumple algoritmo de validación módulo 23");
        }
        this.dni = dni;
    }

    //algún posible método que podría interesar
    public String getDNI() {
        return this.dni;
    }

    public String getDNISoloNumero() {
        return this.dni.substring(0, this.dni.length() - 1);
    }

    public char getDNISoloLetra() {
        return this.dni.charAt(dni.length() - 1);
    }

    //aunque ahora todos los métodos son públicos las code conventions no obligan a que estén juntos
    //se pueden intercalar con los públicos sin problemas

    private boolean esNumero(String s) {
        for (char c : s.toCharArray()) {
            if (!Character.isDigit(c)) {
                return false;
            }
        }
        return true;
    }
}
```

```

    }
    //con excepciones es menos eficiente
    private boolean esNumero2(String s){
        int numero = 0;
        try {
            numero = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            return false;
        }
        return true;
    }

    private char calcularLetra(String parteNumero) {
        int numero = Integer.parseInt(parteNumero);
        return LETRAS_DNI.charAt(numero % 23);
    }
}

//App.java

public class App{
    public static void main(String[] args) {
        //metemos tb. un string vacio ""
        String[] dnis ={"12345678A","123456789","123456789","0000001R","00000001R","", "123abv11a"};
        for(String dni:dnis){
            try{
                System.out.print("\nintentando crear "+ dni+": ");
                new dni.DNI(dni);
                System.out.print(dni+ " icreado con exito!");
            }catch(Exception e){
                System.out.print(e.getMessage());
            }
        }
    }
}

```

Ejercicio U5_B8B_E8: Creamos una clase *personas.Empleado* con los campos:

- DNI (tipo DNI)
- nombre(tipo String)
- sueldo(tipo Double).

Y desde main() de App intentamos crear un empleado como en el ejemplo

```

DNI: 44444444H
NOMBRE: YO
SUELDO: 1000
DNI inválido: 44444444H => el dni no cumple algortimo de validación módulo 23
DNI: 44444444A
NOMBRE: YO
SUELDO: 1000
empleado YO creado correctamente

```