



EXCEPCIONES

EXCEPCIONES

- una forma de reportar errores
 - no usan el resultado del método
 - informa al código cliente de lo que ha ido mal, si algo va mal
 - no pasa nada si todo va bien
- la filosofía es
 - mientras todo va bien, aquí no pasa nada
 - se programa de buena fe”
 - si algo va mal, el servidor grita
 - pero hay que tener oído para detectar problemas

ERRORES EN EJECUCIÓN

java.lang.ArithmeticException: / by zero
at Errores.divisionPorCero(Errores.java:10)

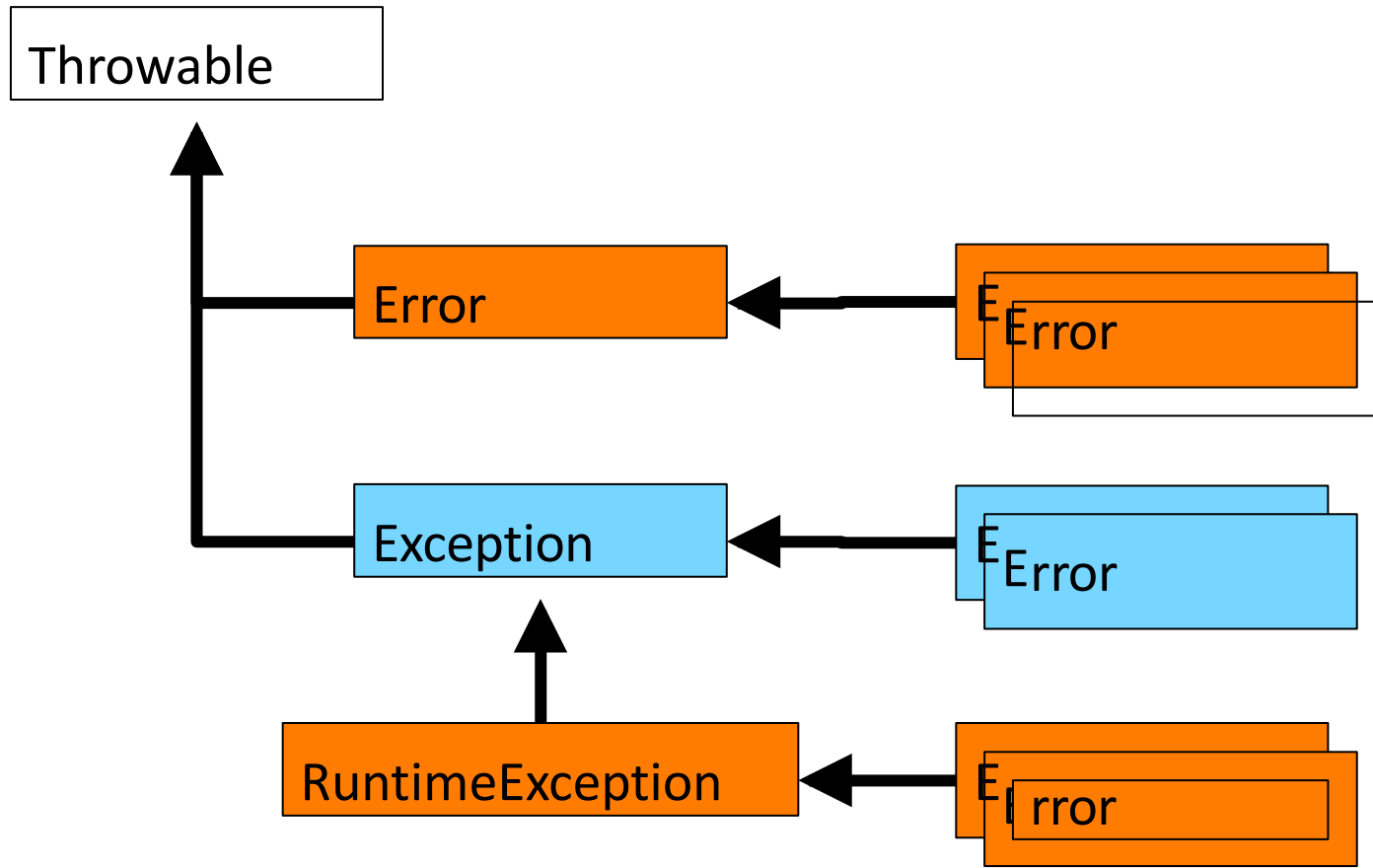
java.lang.NullPointerException
at Errores.objetoInexistente(Errores.java:16)

java.lang.NullPointerException
at Errores.arraySinCrear(Errores.java:21)

java.lang.NegativeArraySizeException
at Errores.arrayImposible(Errores.java:25)

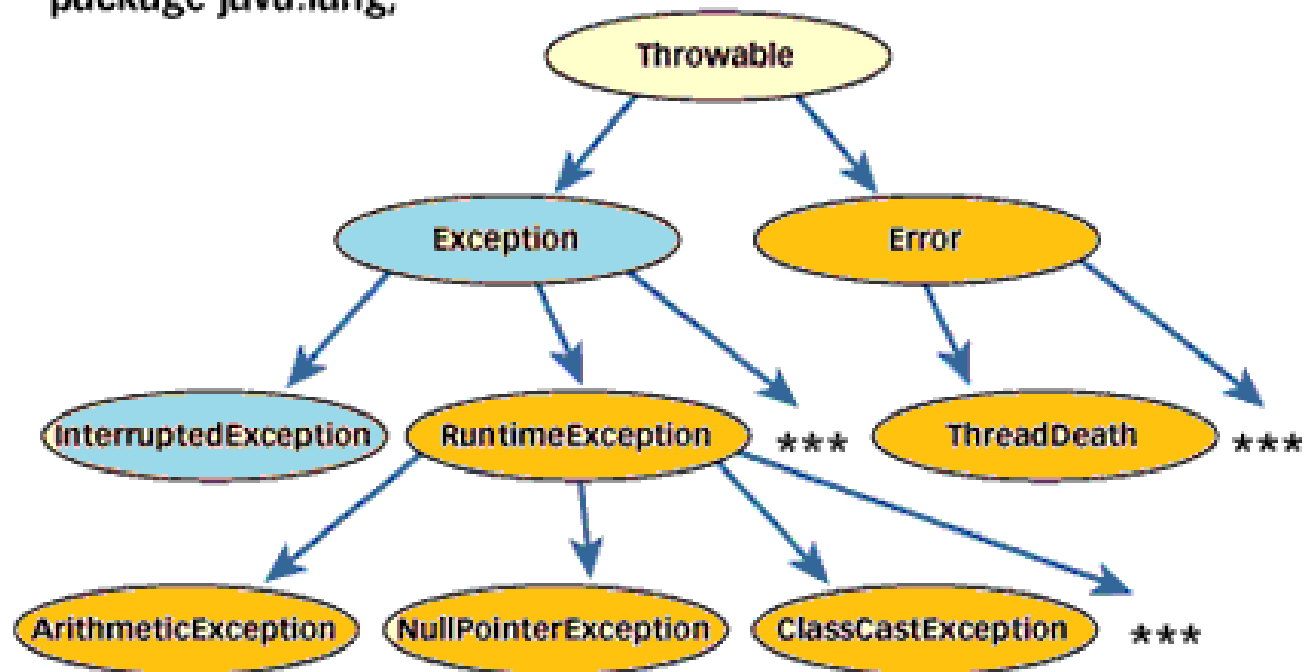
java.lang.ArrayIndexOutOfBoundsException: 1
at Errores.indiceDesbordado(Errores.java:30)

JERARQUÍA DE EXCEPCIONES



JERARQUÍA DE EXCEPCIONES

package java.lang;



TIPICAS

- Error
 - `CoderMalfunctionError`
 - ...
- Exception
 - `IOException`
 - `RuntimeException`
 - otras definidas por nosotros ...
- `RuntimeException`
 - `ArithmeticException`
 - `ClassCastException`
 - `IllegalArgumentException`
 - `IndexOutOfBoundsException`
 - `NullPointerException`
 - ...

OBJETOS EXCEPTION

<http://java.sun.com/javase/6/docs/api/java/lang/Exception.html>

- Constructores
 - Exception()
 - Exception(String message)
- Métodos
 - String toString()
 - String getMessage()
 - void printStackTrace();

EJERCICIOS

- chequear el argumento de elimina(Contacto)
 - lanzando IllegalArgumentException
 - lanzando Exception

```
public void elimina(String clave) {  
    Contacto contacto = map.get(clave);  
    map.remove(contacto.getNombre());  
    map.remove(contacto.getTelefono());  
    numero--;  
}
```


EJERCICIOS

- chequear el argumento de elimina(Contacto)
 - lanzando IllegalArgumentException

```
public void elimina(String clave) {  
    if (clave == null || clave.length() == 0)  
        throw new IllegalArgumentException("clave nula");  
    Contacto contacto = map.get(clave);  
    if (contacto == null)  
        throw new IllegalArgumentException("clave inexistente");  
    map.remove(contacto.getNombre());  
    map.remove(contacto.getTelefono());  
    numero----;  
}
```

EJERCICIOS

- chequear el argumento de elimina(Contacto)
 - lanzando Exception

```
public void elimina(String clave) throws Exception {  
    if (clave == null || clave.length() == 0) throw new Exception("clave nula");  
    Contacto contacto = map.get(clave);  
    if (contacto == null)  
        throw new Exception("clave inexistente");  
    map.remove(contacto.getNombre());  
    map.remove(contacto.getTelefono()); numero--;  
}
```

MÉTODOS Y EXCEPCIONES

- Un método puede, en su cuerpo, lanzar una excepción
- Debe indicar en la cabecera que puede lanzar una excepción
 - nótese que puede lanzarla o no
 - **es una opción, no una obligación**

```
metodo (argumentos) throws Exception {  
    ...  
    throw new Exception();  
    ...  
}
```

EJERCICIOS

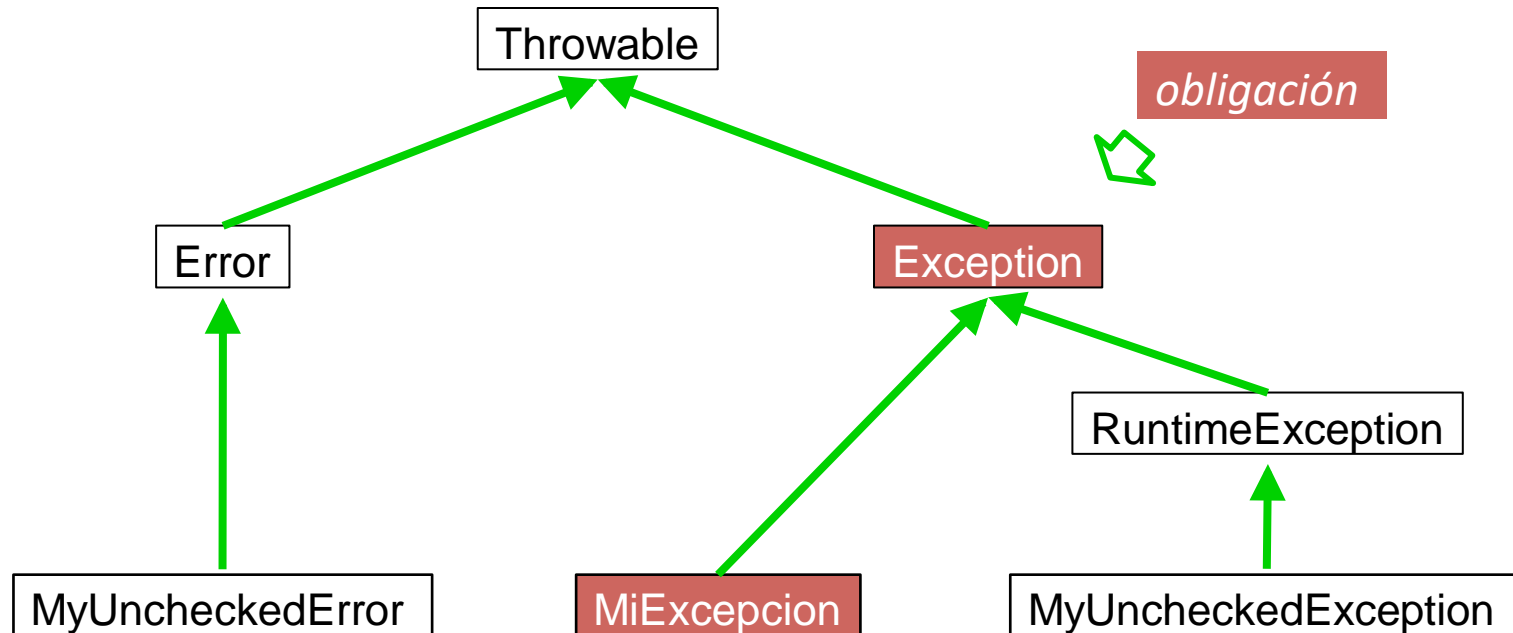
- las excepciones se pueden propagar
“pasando la pelota al siguiente”

```
public void cambia(String claveAntigua, Contacto contacto) throws Exception {  
    elimina(claveAntigua);  
    mete(contacto);  
}
```

OBLIGACIÓN DE DECLARAR

- Las excepciones de clase Exception
 - deben declararse en la cabecera de los métodos que las puedan lanzar
 - o ser capturadas (catch) internamente
- Error
 - no hay que declararlas en las cabeceras
 - no suelen usarse en los programas
 - son errores mayúsculos que detienen el programa
- RuntimeError
 - no hay que declararlas en las cabeceras
 - pueden capturarse en un catch
 - se usan para detectar errores de uso

OBLIGACIÓN DE DECLARAR



NOTA: aún si no hay obligación, se recomienda declarar lo que se programa

EJERCICIOS

- chequear los argumentos del constructor
 - lanzando IllegalArgumentException
 - lanzando Exception

```
public Contacto(String nombre, String telefono, String direccion) {  
    // se eliminan blancos al principio o al final  
    this.nombre = nombre.trim();  
    this.telefono = telefono.trim();  
    this.direccion = direccion.trim();  
}
```

EJERCICIOS

- chequear los argumentos del constructor
 - lanzando `IllegalArgumentException`

```
public Contacto(String nombre, String telefono, String direccion) {  
    if (nombre == null || nombre.length() == 0)  
        throw new IllegalArgumentException("nombre nulo");  
    if (telefono == null || telefono.length() == 0)  
        throw new IllegalArgumentException("telefono nulo");  
    if (direccion == null || direccion.length() == 0)  
        throw new IllegalArgumentException("direccion nulo");  
    // se eliminan blancos al principio o al final  
    this.nombre = nombre.trim();  
    this.telefono = telefono.trim();  
    this.direccion = direccion.trim();  
}
```


EJERCICIOS

- chequear los argumentos del constructor
 - lanzando Exception

```
public Contacto(String nombre, String telefono, String direccion) throws  
Exception {
```

```
    if (nombre == null || nombre.length() == 0) throw  
        new Exception("nombre nulo");
```

```
    if (telefono == null || telefono.length() == 0) throw  
        new Exception("telefono nulo");
```

```
    if (direccion == null || direccion.length() == 0) throw  
        new Exception("direccion nulo");
```

```
    // se eliminan blancos al principio o al final
```

```
    ...
```

```
}
```

EXCEPCIÓN PROPIA

- recurre a los constructores de super

```
public class MiExcepcion extends Exception {  
    public MiExcepcion() {  
        }  
  
    public MiExcepcion(String mensaje) { super(mensaje);  
    }  
}
```

EJERCICIOS

- chequear los argumentos del constructor
 - lanzando MiExcepcion

```
public Contacto(String nombre, String telefono, String direccion) throws MiException {  
    if (nombre == null || nombre.length() == 0) throw new  
        MiExcepcion("nombre nulo");  
    if (telefono == null || telefono.length() == 0) throw new  
        MiExcepcion("telefono nulo");  
  
    if (direccion == null || direccion.length() == 0) throw new  
        MiExcepcion("direccion nulo");  
  
    // se eliminan blancos al principio o al final  
    ...  
}
```

EXCEPCIÓN PROPIA

- recurre a los constructores de super

```
public class MiExcepcion extends Exception {  
    private int codigo;  
  
    public MiExcepcion(int codigo) {  
        this.codigo = codigo;  
    }  
    public MiExcepcion(int codigo, String mensaje) {  
        super(mensaje);  
        this.codigo = codigo;  
    }  
    public int getCodigo() { return codigo; }  
}
```

¿A DÓNDE VAN LAS EXCEPCIONES LANZADAS?

- Su lanzamiento interrumpe la ejecución secuencial
- Se propagan hasta que son capturadas
 - interrumpe toda sentencia que no la capture
- Se capturan con un **catch**
 - se pueden atrapar al vuelo
 - **try { ... } catch**
 - **(Exception e) {**
System.err.println(e);
}

TRY {...} CATCH (ARGS) {...}

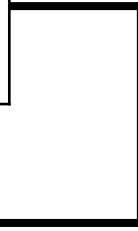
- Si no se lanza nada, el catch como si no existiera
- Dentro del try se lanza (throw) una excepción
 - Se olvida el resto del código
 - Se ejecuta lo que diga el catch

```
try {
```

```
...  
throw new Exception();  
...
```

```
} catch (Exception e) {
```

```
...  
}
```



¿QUÉ HACER CON LA PELOTA?

- o se pasa ... o se atrapa
- lo que no puede hacerse es ignorarla

```
public void actionPerformed(ActionEvent ev) {
    agenda.mete(new Contacto(nameField.getText(),
                             phoneField.getText(),
                             address.getText()));
}
```

¿QUÉ HACER CON LA PELOTA?

- puede no hacer nada

```
public void actionPerformed(ActionEvent ev) {  
    try {  
        agenda.mete(new Contacto(nameField.getText(),  
            phoneField.getText(),  
            address.getText()));  
    } catch (Exception e) {  
        // no hago nada  
    }  
}
```


¿QUÉ HACER CON LA PELOTA?

- o mejor se lo comenta al usuario

```
public void actionPerformed(ActionEvent ev) {  
    try {  
        agenda.mete(new Contacto(nameField.getText(),  
            phoneField.getText(),  
            address.getText()));  
    } catch (Exception e) {  
        JOptionPane.showMessageDialog(GUI.this, e.getMessage());  
    }  
}
```

MÉTODOS Y EXCEPCIONES

```
método_3() throws Exception {  
    throw new Exception();  
}
```

```
método_2() throws Exception {  
    método_3();  
}
```

```
método_1() {  
    try { método_2();  
    } catch (Exception e) { e...; }  
}
```

```
main(...) { método_1(); }
```

traza: llamada al método



MÉTODOS Y EXCEPCIONES

```
método_3() throws Exception {  
    throw new Exception();  
}
```

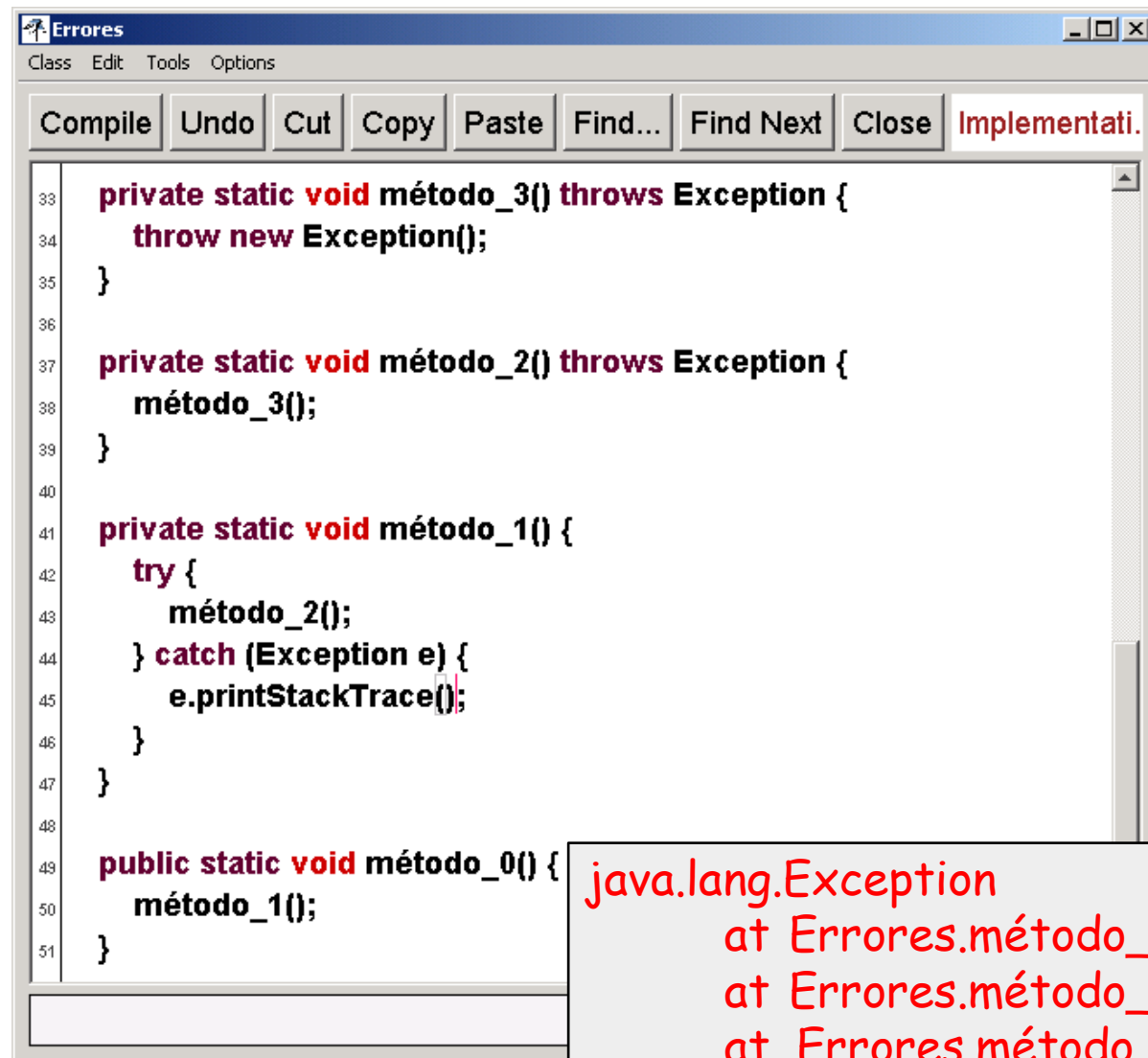
```
método_2() throws Exception {  
    método_3();  
}
```

```
método_1() {  
    try { método_2();  
    } catch (Exception e) { e...; }  
}
```

```
main(...) { método_1(); }
```

trasiego de la excepción

The diagram illustrates the flow of an exception through a series of method calls. It consists of four rectangular boxes arranged vertically, each containing a code snippet for a different method. The boxes are labeled 'método_3()', 'método_2()', 'método_1()', and 'main(...)'. Red arrows indicate the path of the exception: one arrow points from the right side of the 'método_3()' box to the right side of the 'método_2()' box, and another arrow points from the right side of the 'método_2()' box to the right side of the 'método_1()' box. A third arrow points from the right side of the 'método_1()' box to the right side of the 'main(...)' box. The text 'trasiego de la excepción' is written vertically in red to the right of the arrows, indicating the direction of the exception flow.



The screenshot shows an IDE window titled "Errores" with a menu bar (Class, Edit, Tools, Options) and a toolbar (Compile, Undo, Cut, Copy, Paste, Find..., Find Next, Close, Implementati.). The code editor displays the following Java code:

```
33 private static void método_3() throws Exception {
34     throw new Exception();
35 }
36
37 private static void método_2() throws Exception {
38     método_3();
39 }
40
41 private static void método_1() {
42     try {
43         método_2();
44     } catch (Exception e) {
45         e.printStackTrace();
46     }
47 }
48
49 public static void método_0() {
50     método_1();
51 }
```

A stack trace is visible at the bottom right of the editor, indicating the sequence of method calls that led to the exception.

java.lang.Exception
at Errores.método_3(Errores.java:34)
at Errores.método_2(Errores.java:38)
at Errores.método_1(Errores.java:43)
at Errores.método_0(Errores.java:50)

CATCH

```
try {  
    ... ..  
} catch (claseA ida) {  
    ... ..  
} catch (claseB idb) {  
    ... ..  
} catch (claseC idc) {  
    ... ..  
}
```

Los diferentes *catch* se intentan en el orden en que aparecen hasta que uno de ellos casa; después de casar con uno, los demás se olvidan

- Casar significa que la excepción a agarrar es de la clase indicada o de una clase extensión de ella (subtipo de ...)

- ida instanceof claseA
- idb instanceof claseB
- idc instanceof claseC

TRY ... CATCH ... finally

- Se puede añadir un trozo **finally** que se ejecuta
 - bien cuando acaba el código normal (**try**)
 - bien cuando acaba el código excepcional (**catch**)
 - ... es decir, SIEMPRE se ejecuta;
incluso si **try** lanza una excepción que no captura ningún **catch**

finally

```
try {
```

```
    .....  
    throw new Exception();  
    .....
```

```
}
```

```
catch (Exception e) {
```

```
    .....  
}
```

```
}
```

```
finally {
```

```
    .....  
}
```

```
}
```

```
try {
```

```
    .....  
    throw new Exception();  
    .....
```

```
}
```

```
catch (Exception e) {
```

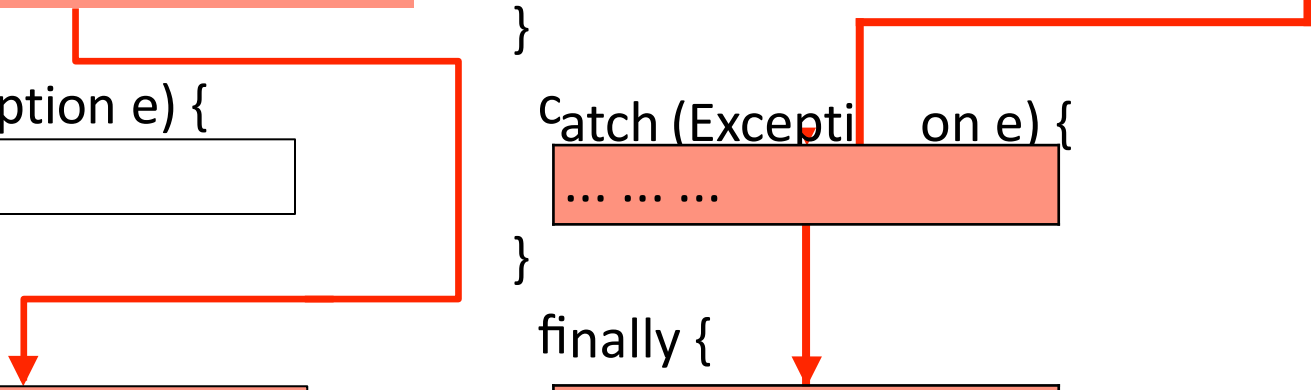
```
    .....  
}
```

```
}
```

```
finally {
```

```
    .....  
}
```

```
}
```



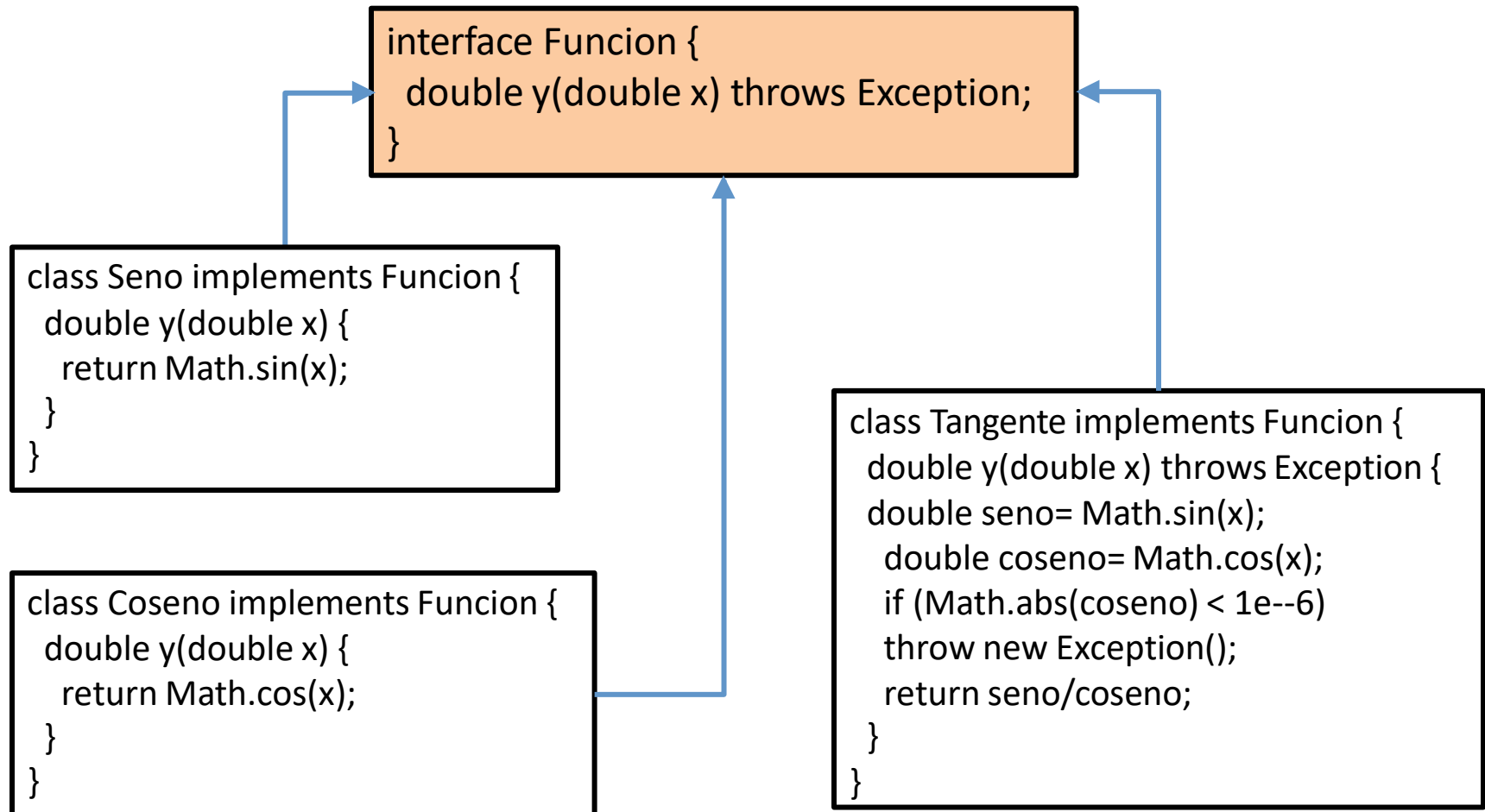
EJEMPLO

```
public class Inversos {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        int pruebas = 0;  
        while (true) {  
            double inverso = 0;  
            try {  
                String x = scanner.next();  
                if (x.equals("fin"))  
                    break;  
                int valor = Integer.parseInt(x);  
                inverso = 100 / valor;  
            } catch (Exception e) {  
                System.out.println("-> " + e);  
                inverso = 0;  
            } finally {  
                pruebas++;  
                System.out.println("≧" + pruebas + ": " + inverso);  
            }  
        }  
    }  
}
```


EXCEPTION & INTERFACE

- Si en la interface NO se permiten
 - la implementación NO puede permitir las
- Si en la interface SÍ se permiten
 - la implementación puede permitir las
 - o no

EJEMPLO



EJEMPLO

```
double raiz(Funcion f, double m, double n)
    throws Exception {
    double p = (m + n) / 2;
    if (Math.abs(f.y(p)) <
        ERROR) return p;
    if (Math.abs(m - n) < ERROR)
        throw new MiEx("no hay solución");
    if (f.y(m) * f.y(p) >
        0) return raiz(f,
        p, n);
    else
        return raiz(f, m, p);
}
```

```
public static void main(String[] args) throws Exception {
    // nadie captura la excepción
    long t0 = System.currentTimeMillis();
    try {
        Funcion f = new Test();
        System.out.println("raíz= " + raiz(f, 1, 3));
    } finally {
        long t2 = System.currentTimeMillis();
        System.out.println((t2 - t0) + "ms");
    }
}
```

EXCEPTION & EXTENDS

- La subclase puede redefinir
 - métodos de objetos
 - a base de definir métodos con la misma **signatura**
 - mismo nombre
 - mismo número y tipo de argumentos
 - **igual o menos excepciones**
 - igual o más visible
 - (paquete) → public
 - igual resultado

EXCEPCIONES

- Las excepciones ofrecen salidas de emergencia
 - de { sentencias; }
 - de llamadas a métodos
- Se pretende que el programador se centre en el comportamiento “normal” (si todo va bien)
 - y que la situación excepcional aborte
 - informando de dónde y por qué

¿CUÁNDO USAR EXCEPCIONES?

- Conviene usarlo cuando hay que programar una situación anómala
 - el que lo lea se pondrá en situación mental de analizar una situación excepcional

¿CUÁNDO EXTENDER?

- Programa mejor documentado
- Crear hijos para poder discriminar en el *catch*

¿CUANDO LANZAR EXCEPTION?

- Un método lanzará una Exception, que queda documentada en la cabecera, cuando
 - la responsabilidad de determinar si el método falla es del propio método, que informa cuando lo detecta
 - es responsabilidad del que llama gestionar la excepción cuando se produzca
 - son errores RECUPERABLES

PROGRAMACIÓN INFORMATIVA

¿... RUNTIMEEXCEPTION?

- Un método lanzará una RuntimeException, incluso NO documentada en la cabecera, cuando
 - es responsabilidad del que llama al método, saber si los argumentos son correctos o darían pie a un error
 - el que llama no prevé ninguna excepción, ni tratamiento explícito alguno
 - el llamado se protege de fallos del programador lanzando una excepción de ejecución
 - son errores IRRECUPERABLES

PROGRAMACIÓN DEFENSIVA