

# Manejo de errores: Excepciones en Java

ELO329: Diseño y Programación Orientados a  
Objetos

# Excepciones (o errores)

- Los lenguajes orientados a objeto han buscado la forma de facilitar la programación de las situaciones de error en un programa.
- Muchas cosas pueden generar excepciones (o errores): Errores de hardware (falla de disco), de programa (acceso fuera de rango en arreglo), apertura de archivo inexistente, ingreso de un depósito negativo, probabilidad mayor que 1, etc.
- En lugar de mezclar el código asociado a la lógica principal del programa con el de tratamiento de excepciones dificultando la claridad de la tarea principal del programa, los lenguajes orientados a objeto como **Java y C++ disponen un mecanismo de excepciones que separa la parte fundamental del código (mayor % de los casos) de las situaciones de error.**
- Una **excepción es un evento que ocurre durante la ejecución de un programa que rompe el flujo normal de ejecución.** Cuando se habla de excepciones nos referimos a un evento excepcional.

# Excepciones (o errores) Cont.

- Cuando se produce una excepción dentro de un método, se crea un objeto que contiene información sobre la excepción y retorna en forma inusual al código llamador con la información de la excepción.
- La rutina receptora de la excepción es responsable de reaccionar a tal evento inesperado.
- **Cuando creamos un objeto para la excepción y lo pasamos al código llamador decimos que lanzamos una excepción (Throw an exception)**
- Si el método llamador maneja la excepción se busca hacia atrás en la pila de llamados anidados hasta encontrarlo.
- **Decimos que el manejador atrapa la excepción (palabra reservada “catch”)**

# Ventajas de usar excepciones: Separar código normal de casos de error

- Supongamos que queremos leer un archivo completo a memoria:

```
readFile {  
    abrir un archivo;  
    determinar su tamaño;  
    localizar esa cantidad de memoria;  
    leer el archivo en memoria;  
    cerrar el archivo;  
}
```

# Ventajas de usar excepciones: Separar código de casos de error

- Sin excepciones podríamos encontrar algo así:

```
errorCodeType readFile {
    initialize errorCode = 0;
    Abrir el archivo;
    if (theFileIsOpen) {
        Determinar el largo del archivo;
        if (gotTheFileLength) {
            Localizar esa cantidad de memoria;
            if (gotEnoughMemory) {
                Leer el archivo en memoria;
                if (readFailed)    errorCode = -1;
            } else    errorCode = -2;
        } else    errorCode = -3;
        Cerrar el archivo;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else    errorCode = errorCode and -4;
    } else    errorCode = -5;
    return errorCode;
}
```

# Ventajas de usar excepciones: Separar código de casos de error

- Con excepciones:

```
readFile {  
  try {  
    abrir un archivo;  
    determinar su tamaño;  
    localizar esa cantidad de memoria;  
    leer el archivo en memoria;  
    cerrar el archivo;  
  } catch (fileOpenFailed) {  
    doSomething;  
  } catch (sizeDeterminationFailed) {  
    doSomething;  
  } catch (memoryAllocationFailed) {  
    doSomething;  
  } catch (readFailed) {  
    doSomething;  
  } catch (fileCloseFailed) {  
    doSomething;  
  }  
}
```

Quando el código lanza una excepción, se detiene el procesamiento del código restante en el try y se continúa en el catch correspondiente o se retorna del método si no tenemos try (esto es relanzar la excepción).

# Aspectos Operativos

- Dos partes: Quien genera la condición de error (excepción) debe informarla. Quien es informado debe hacer algo al respecto.
- Para lo primero, el método que genera la excepción la informa lanzando un objeto que porta antecedentes sobre el error.
- En Java los objetos lanzados deben ser instancias de clases derivadas de Throwable.

Ej.

```
Throwable e = new IllegalArgumentException("Stack underflow");  
throw e;
```

O alternativamente:

```
throw new IllegalArgumentException("Stack underflow");
```

- Al invocar métodos que puedan lanzar error se usa la sentencia `try ... catch`.

# Captura de Excepciones (completo)

- El manejo de excepciones se logra con el bloque try

Forma general:

```
try {  
    //sentencias normales  
} catch (e-clase1 e ) {  
    // sentencias tratamiento error e-clase1  
} catch (e-clase2 e ) {  
    // sentencias tratamiento error e-clase2  
} ...  
finally { // esta parte es opcional  
    //sentencias  
}
```

- La cláusula finally es ejecutada con posterioridad cualquiera sea la condición de término del try (sin o con error, return, break, continue). Esta sección permite dejar las cosas consistentes antes del término del bloque try.



# Captura de Excepciones: Ejemplos

- ```
try {  
    // código  
} catch (StackError e )  
{  
    // código que se hace cargo del error reportado en e  
}
```
- El bloque try puede manejar múltiples excepciones:  

```
try {  
    // código  
} catch (StackError e )  
{  
    // código para manejar el error de stack  
} catch (MathError me)  
{  
    // código para manejar el error matemático indicado en me.  
}
```

# Captura de Excepciones: Ejemplo 1

```
public static void doio (InputStream in, OutputStream out) {  
    int c;  
    try {  
        while (( c=in.read()) >=0 ) {  
            c= Character.toLowerCase( (char) c);  
            out.write( c );  
        }  
    } catch ( IOException e ) { // read y write pueden lanzar error  
        System.err.println("doio: I/O Problem");  
        System.exit(1);  
    }  
}
```

# Captura de Excepciones: Ejemplo 2

.....

```
try { FileInputStream infile = new FileInputStream(argv[0]);  
    File tmp_file = new File(tmp_name);  
  
    ....  
} catch (FileNotFoundException e) {  
    System.err.println("Can't open input file "+ argv[0]);  
    error = true;  
} catch (IOException e ) {  
    System.err.println("Can't open temporary file "+tmp_name);  
    error = true;  
}finally {  
    if ( infile != null) infile.close();  
    if (tmp_file != null) tmp_file.close();  
    if (error) System.exit();  
}
```

- El código de la sección finally es ejecutado no importando si el bloque try terminó normalmente, por excepción, por return, o break.

# Ventajas de las Excepciones

- Claridad y simplicidad de la tarea a realizar más frecuentemente.
- Propaga los errores hacia atrás hasta el punto donde se puede tomar una acción.
- Se agrupan los errores según su naturaleza.
- Ej:
  - Si hay más de un archivo que se abre, basta con un código para capturar tal caso.
  - Si se lanzan excepciones que son todas subclases de una base, basta con capturar la base para manejar cualquiera de sus instancias derivadas.

# Tipos de Excepciones

- Las hay de dos tipos
  - Aquellas generadas por el lenguaje Java. Éstas se generan cuando hay errores de ejecución, como al tratar de acceder a métodos de una referencia no asignada a un objeto, división por cero, etc. En este caso no se exige el bloque try.
  - Aquellas incluidas por el programador vía paquetes o sus propias clases.
- El compilador chequea por la captura de las excepciones lanzadas por los objetos usados en el código.
- Si una excepción no es capturada, debe ser relanzada.

# Reenviando Excepciones: dos formas

```
■ public static void doio (InputStream in, OutputStream out) throws IOException {  
    int c; // en caso de más de una excepción throws exp1, exp2  
    while (( c=in.read()) >=0 ) {  
        c= Character.toLowerCase( (char) c);  
        out.write( c );  
    }  
}
```

*Si la excepción no es manejada con try, debe ser reenviada*

■ Alternativamente:

```
■ public static void doio (InputStream in, OutputStream out) throws IOException {  
    int c;  
    try {  
        while (( c=in.read()) >=0 ) {  
            c= Character.toLowerCase( (char) c);  
            out.write( c );  
        }  
    } catch ( IOException t )  
        throw t;  
}
```

*En este caso el método envía una excepción - que aquí corresponde al mismo objeto capturado -por lo tanto se debe declara en la cláusula throws.*

■ !!! Si el método usa la sentencia throw debe indicarlo en su declaración con la cláusula throws.

■ En este caso es responsabilidad de quien llame a doio atrapar la excepción o relanzarla. Así esto sube hasta posiblemente llegar al método main.

# Creación de tus propias excepciones

- Siempre es posible lanzar alguna excepción de las ya definidas en Java (IOException por ejemplo).
- También se puede definir nuevas excepciones creando clases derivadas de las clases Error o Exception.

```
class ZeroDenominatorException extends Exception
{
    private int n;
    public ZeroDenominatorException () {}
    public ZeroDenominatorException(String s) {
        super(s);
    }
    public setNumerator(int _n) { n = _n;}
    // otros métodos de interés
}
```

- Luego la podemos usar como en este constructor:

```
....
public Fraction (int n, int d) throws ZeroDenominatorException {
    if (d == 0) {
        ZeroDenominatorException myExc = new
            ZeroDenominatorException("Fraction: Fraction with 0 denominator?");
        myExc.setNumerator(n);
        throw (myExc);
    }
    ....
}
```

# Creación de tus propias excepciones

```
class ProbabilidadException extends Exception {  
    private float p;  
    public ProbabilidadException(float p, String s) {  
        super(s);  
        this.p=p;  
    }  
    public float getProbabilidad() { return p;}  
}
```

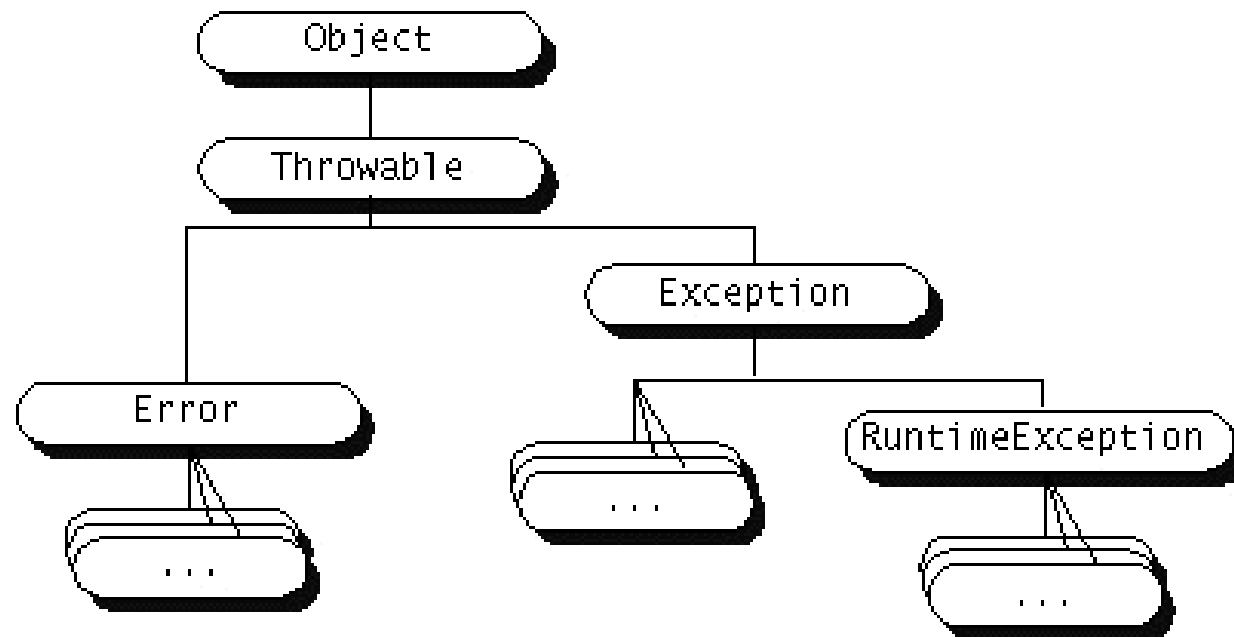
■ Luego:

```
public class MyClass {  
    public void setProbabilidad (float nueva_p) throws ProbabilidadException {  
        if (nueva_p > 1.0) {  
            throw(new ProbabilidadException(nueva_p,"Probabilidad Mayor  
            que uno"));  
        }  
        probabilidad = nueva_p;  
    }  
    // .... otras declaraciones....  
    private float probabilidad;  
}
```



# Jerarquía de Excepciones

- Cuando creamos nuestras excepciones, serán subclases de Exception.
- Java prohíbe crear subclases de Throwable.
- Java no obliga a manejar o reenviar RuntimeException.



# ¿Cuándo no podemos relanzar una excepción?

- Hay situaciones en que estamos obligados a manejar una excepción. Consideremos por ejemplo:

```
class MyApplet extends Applet {
    public void paint (Graphics g) {
        FileInputStream in = new FileInputStream("input.dat");    //ERROR
        ....
    }
}
```

- La intención es redefinir un método de la clase Applet - método paint- el cual no genera ninguna excepción. Si un método no genera excepciones la función que lo redefina no puede lanzar excepciones.
- Lo previo obliga a que debemos hacernos cargo de la excepción.

```
class MyApplet extends Applet {
    public void paint (Graphics g) {
        try {
            FileInputStream in = new FileInputStream("input.dat");    //Posible error
            ....
        } catch (Exception e) {///.....}
    }
}
```

# Cosas a tomar en cuenta

- Las excepciones consumen tiempo, no usarlas cuando hay alternativas mejores, ejemplo `ExceptionalTest.java`
- Agrupar el manejo de varias excepciones en un único `try...`
- En cada caso evaluar si es mejor atrapar la excepción o reenviarla a código llamador.

- Ejemplo:

```
public void readStuff(String name) Throws IOException {  
    FileInputStream in= new FileInputStream(name);  
    ....  
}
```

El código que llama al método puede manejar de mejor forma la excepción que aquí.