

# Programación

## UD 9: Gestión de excepciones

# Gestión de excepciones

---

## 1.- Fallos de ejecución y su modelo Java

### 1.1.- La jerarquía Throwable

### 1.2.- Ampliación de la jerarquía Throwable con excepciones de usuario

## 2.- Tratamiento de excepciones

### 2.1.- Captura de excepciones: try / catch / finally

### 2.2.- Propagación de excepciones: throw versus throws

---

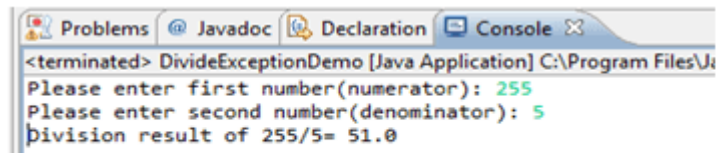
# 1.- Fallos de ejecución y su modelo Java

# 1.- Fallos de ejecución y su modelo Java

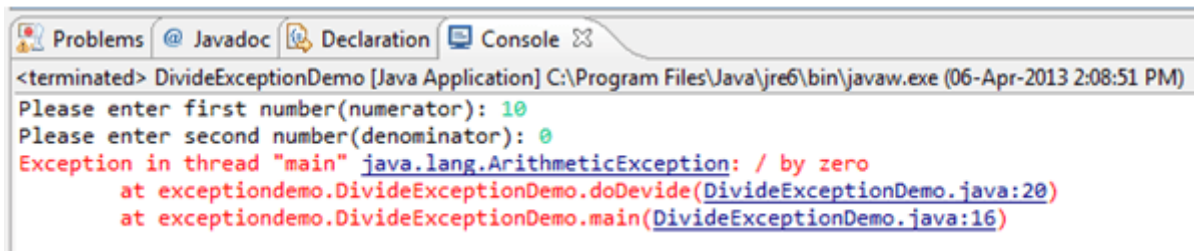
Cuando se aprende a programar, es muy habitual cometer fallos de programación como dividir por cero, calcular la raíz cuadrada de un valor negativo y acceder a una posición inexistente de un array o a un objeto null.

A los fallos o condiciones anormales que ocurren durante la ejecución de un segmento de código se las denomina **excepciones**.

Al ocurrir una excepción, ésta debe ser **tratada** porque, de lo contrario, la ejecución de la secuencia de código se detendrá dando un **error en tiempo de ejecución**.



```
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (06-Apr-2013 2:08:51 PM)
Please enter first number(numerator): 255
Please enter second number(denominator): 5
division result of 255/5= 51.0
```



```
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (06-Apr-2013 2:08:51 PM)
Please enter first number(numerator): 10
Please enter second number(denominator): 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at exceptiondemo.DivideExceptionDemo.doDevide(DivideExceptionDemo.java:20)
    at exceptiondemo.DivideExceptionDemo.main(DivideExceptionDemo.java:16)
```

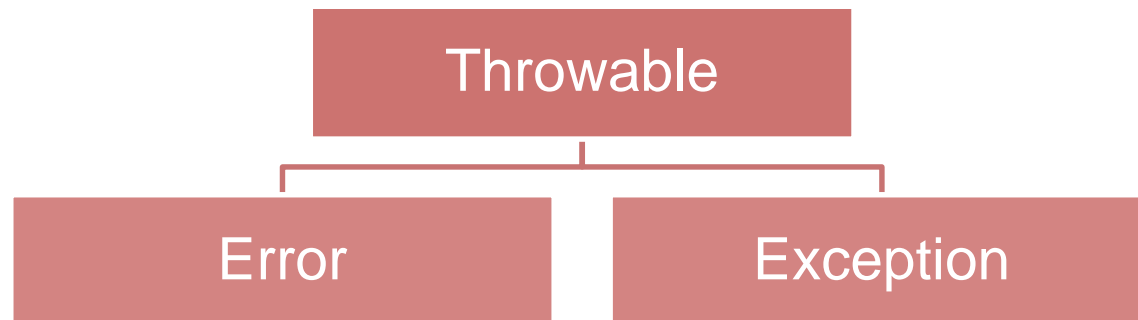
# 1.- Fallos de ejecución y su modelo Java

---

En el lenguaje Java, las excepciones son objetos que describen una condición excepcional que se produce durante la ejecución de un fragmento de código.

Dicho objeto se envía al causante de la excepción, que puede tratar la aparición de la misma (mediante instrucciones explícitas de tratamiento), o no tratarla dejando que sea el sistema (dando un error) el encargado de ello.

Los fallos de ejecución descritos se clasifican como **excepciones** porque resulta factible o razonable evitar sus consecuencias y reconducir la ejecución de la aplicación una vez se producen; justo lo contrario sucede con los denominados **errores**, fallos de ejecución que por ser muy severos o por involucrar al soporte hardware de la aplicación (por ejemplo, agotar la memoria o acceder indebidamente a una de sus zonas) resultan de difícil o imposible solución y, por ello, irrecuperables.



---

## 1.1.- La jerarquía Throwable

# 1.1.- La jerarquía Throwable

---

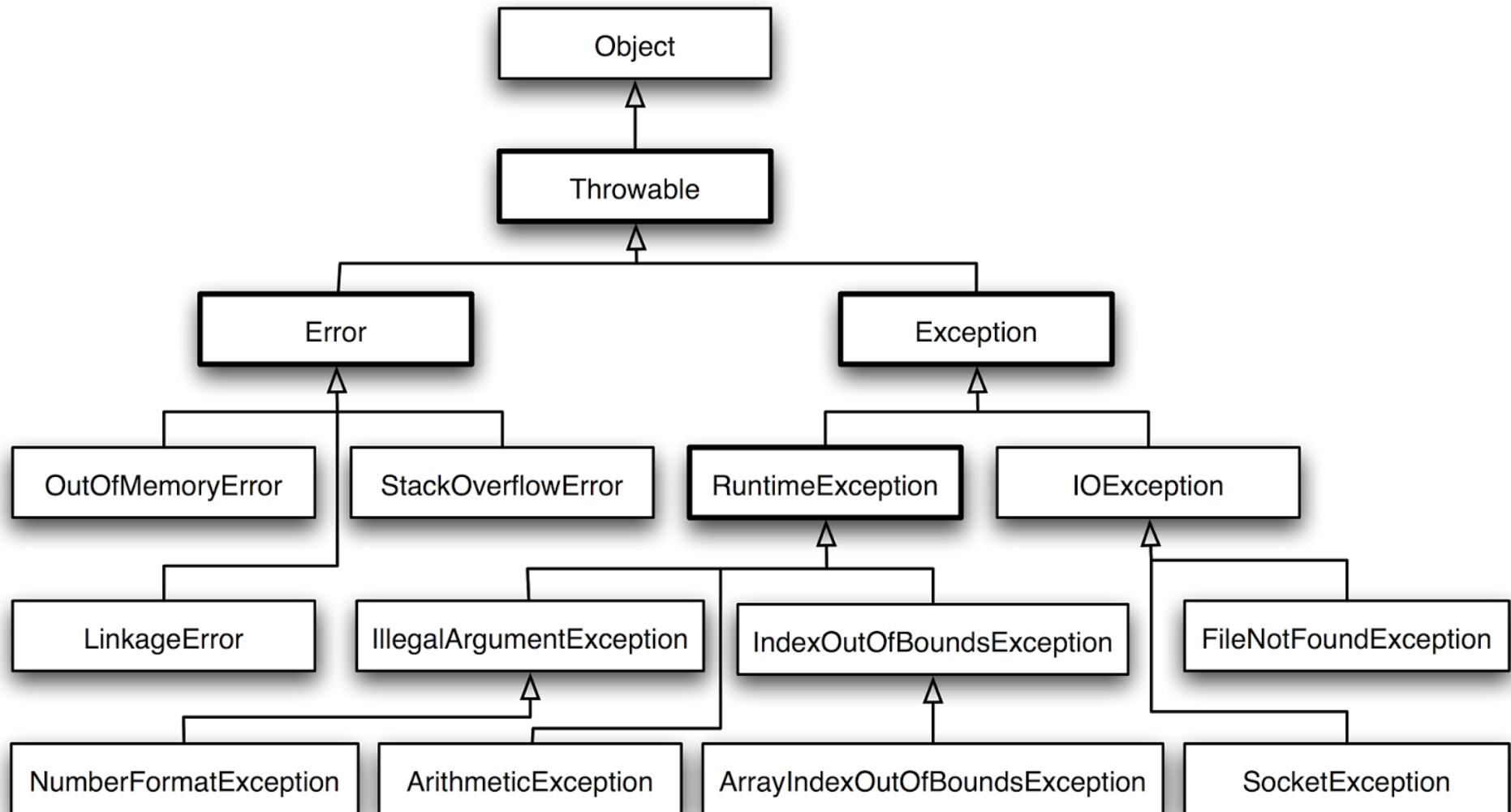
El lenguaje Java proporciona, en su paquete estándar `java.lang`, la clase **Throwable**, derivada de `Object`, que **representa cualquier fallo de ejecución independientemente de su tipo**.

Visto que una excepción es un fallo de ejecución -recuperable- y que también un error es un fallo de ejecución -aunque irrecuperable-, las clases **Error** y **Exception** se diseñan como derivadas directas de `Throwable`, la base de la jerarquía Java de errores y excepciones.

También es importante destacar que `Exception` y `Error` son además las raíces de sendas jerarquías donde coexisten clases que representan tipos concretos de excepciones y errores junto con subárboles de clases que representan subtipos de éstos; así, por ejemplo, de `Exception` se derivan tanto **`ClassNotFoundException`** que representa la excepción que se produce al cargar una clase empleando un nombre incorrecto como **`RuntimeException`** que es la clase raíz de la jerarquía que contiene todos los fallos de programación.

Se puede ver la **jerarquía completa** en la siguiente diapositiva:

# 1.1.- La jerarquía Throwable





---

## 1.2 Ampliación de la jerarquía Throwable con excepciones de usuario

# 1.2.- Excepciones de usuario

---

Como ya se ha comentado previamente, la jerarquía Throwable del estándar de Java **no contempla la representación de los fallos en la lógica de una aplicación** precisamente por su carácter *ad hoc*.

Una excepción de este tipo se produce al ignorar el resultado atípico que obtiene en ciertas circunstancias alguno de los métodos que usa una aplicación dada.

Además, existe otro tipo de errores que tampoco serían advertidos por el compilador ni se considerarían fallos de programación.

Para poder ser tratado en Java, cualquier fallo en la lógica de una aplicación debe ser **representado mediante una nueva clase de la jerarquía Throwable**, en concreto, como una nueva clase derivada de Exception.

A estas nuevas clases, para distinguirlas de las que ya figuran en la jerarquía Throwable del estándar de Java, se las denomina **excepciones de usuario** (en inglés, user-defined exceptions) aunque, en principio, poseen la misma funcionalidad básica y características que cualquier otra con los métodos que heredan de Exception.

# 1.2.- Excepciones de usuario

---

## Ejemplo (1/2):

Lanzar una excepción cuando un número entero no esté en un rango determinado.

1.- Definir la clase *ExcepcionIntervalo* derivada de la clase base *Exception*.

```
public class ExcepcionIntervalo extends Exception {  
    public ExcepcionIntervalo(String msg) {  
        super(msg);  
    }  
}
```

2.- Lanzar la excepción cuando el número esté fuera de rango.

```
public static void rango(int num) throws ExcepcionIntervalo{  
    if ((num > 100) || (num < 0)){  
        throw new ExcepcionIntervalo("Número fuera del intervalo");  
    }  
}
```

---

```
Exception in thread "main" ExcepcionIntervalo: Número fuera del intervalo  
    at Principal.rango(Principal.java:15)  
    at Principal.main(Principal.java:6)
```

# 1.2.- Excepciones de usuario

Ejemplo (2/2):

3.- Capturar la excepción.

```
...  
try{  
    numerador    = Integer.parseInt(str1);  
    denominador  = Integer.parseInt(str2);  
  
    rango( numerador );  
    rango( denominador );  
  
    cociente     = numerador/denominador;  
    respuesta    = String.valueOf(cociente);  
  
}catch(NumberFormatException ex){  
    respuesta = "Se han introducido caracteres no numéricos";  
  
}catch(ArithmeticException ex){  
    respuesta = "División entre cero";  
  
}catch(ExcepcionIntervalo ex){  
    respuesta = ex.getMessage();  
}  
  
System.out.println( respuesta );
```

El programa no saca las líneas en rojo de la excepción, sino que lo hace a través de un *System.out* y continúa con el flujo normal del programa.

---

## 2.- Tratamiento de excepciones

## 2.- Tratamiento de excepciones

---

Se ha visto que una excepción provoca la terminación abrupta del programa. Sin embargo, una excepción puede tratarse para evitar dicha terminación abrupta.

Existen dos maneras de tratar una excepción:

- ✓ Capturar la excepción
- ✓ Propagar la excepción

---

## 2.1 Captura de excepciones: try / catch / finally

## 2.1 Captura de excepciones: try / catch / finally

---

```
try {  
    ... // instrucciones que pueden provocar alguna excepción  
  
}  
catch ( NombreDeExcepcion1 nomE1 ) {  
    ... // instrucciones a realizar si se produce  
        // la excepción NombreDeExcepcion1  
  
}  
... // puede haber varios catch por cada try  
catch ( NombreDeExcepcionX nomEX ) {  
    ... // instrucciones a realizar si se produce  
        // la excepción NombreDeExcepcionX  
  
}  
finally {  
    ... // bloque opcional, instrucciones a realizar tanto  
        // si se ha producido una excepción como si no  
}
```



## 2.1 Captura de excepciones: try / catch / finally

---

Básicamente, el significado es como sigue:

1. Se intenta (**try**) ejecutar un bloque de código en el que pueden ocurrir errores que se representan con alguna de las excepciones que se explicitan en los bloques catch.
2. Si se produce un error, el sistema lanza una excepción (**throws**) que puede ser capturada (**catch**) en base al tipo de excepción, ejecutándose las instrucciones correspondientes.
3. Finalmente, tanto si se ha producido o no una excepción y si ésta ha sido o no tratada, se ejecutan las instrucciones asociadas a la cláusula **finally**.
4. Al finalizar todo el bloque, la ejecución se reanuda del modo habitual.

Siempre que aparece una cláusula try, debe existir al menos una cláusula catch o finally.

Nótese que, **para una única cláusula try, pueden existir tantos catch como sean necesarios** para tratar las excepciones que se puedan producir en el bloque de código del try.

Cuando en el bloque try se lanza una excepción, **los bloques catch se examinan en orden**, y el primero que se ejecuta es aquel cuyo tipo sea compatible con el de la excepción lanzada.

## 2.1 Captura de excepciones: try / catch / finally

---

Así pues, el orden de los bloques catch es importante.

Por ejemplo, el orden en los bloques catch que siguen no sería adecuado:

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

Con este orden, el segundo bloque catch nunca se alcanzaría, puesto que todas las excepciones serían capturadas por el primero, ya que la excepción `ExcepcionNumeroNegativo` se deriva de la clase `Exception`.

Afortunadamente, el compilador advierte sobre esto. El orden correcto consiste en invertir los bloques catch para que la excepción más específica aparezca antes que cualquier excepción de una clase antecesora.

## 2.1 Captura de excepciones: try / catch / finally

---

El bloque precedido por *finally es opcional* si hay al menos un catch.

Contiene un grupo de instrucciones que se ejecutarán tanto si el try tiene éxito como si no.

Aparentemente funciona igual si se sitúa una instrucción después del último catch o dentro del finally, como en el ejemplo que sigue:

```
try {  
    ... // instrucciones que pueden generar una excepción  
    System.out.println("Intento logrado");  
}  
catch (NombreExcepcion e) {  
    System.out.println("Capturada!");  
}  
finally {  
    System.out.println("Y el finally!");  
}
```

## 2.1 Captura de excepciones: try / catch / finally

---

En el ejemplo, los puntos suspensivos representan las instrucciones que pueden producir la excepción `NombreExcepcion` capturada en el `catch`. Tanto en un caso como en otro, el final de la ejecución del fragmento sería la escritura del texto “Y el finally!”.

Sin embargo, ¿qué ocurriría si durante la ejecución de los puntos suspensivos se produjese una excepción diferente a `NombreExcepcion`, por ejemplo `OtraExcepcion`?

La respuesta es que aún así, antes de la terminación abrupta del código, se efectuaría el `finally`.

(sin Excepcion):

Intento logrado

Y el finally!

(con `NombreExcepcion`):

Capturada!

Y el finally!

(con `OtraExcepcion`):

Y el finally!

Esto puede quedar más claro si se piensa que en el `finally` pueden ir instrucciones como guardar datos en ficheros o bases de datos, etc. Es decir, aquellas instrucciones que salvaguardan la integridad de la información o de la ejecución.

---

## 2.2 Propagación de excepciones: throw versus throws

## 2.2 Propagación de excepciones

---

Si la excepción se produce dentro de un método, puede elegirse dónde se trata.

Si se trata dentro del método se utilizan las cláusulas ya vistas try/catch/finally pero si no, puede decidirse su propagación hacia el punto del programa desde donde se invocó el método.

Para hacer esto es necesario que el método tenga en su cabecera la cláusula **throws** y el nombre de la clase de excepción (o clases, separadas por comas) que se propagarán desde dicho método.

De este modo, si se produce una excepción dentro del cuerpo del método se abortará la ejecución del mismo y se lanzará la excepción hacia el punto desde donde se invocó al método.

**throws NombreExcepcion:** se sitúa en la cabecera del método e indica que, si se produce una excepción de ese tipo, el método la propagará al punto de invocación del mismo;

**throw NombreExcepcion:** indica que se lanza la excepción que se escribe a continuación.

## 2.2 Propagación de excepciones

---

Ejemplo (1/3):

```
static int leerEnteroPositivo(Scanner t) throws ExcepcionIntervalo {  
    int leido;  
    do {  
        System.out.print("Introduce un entero positivo: ");  
        leido = t.nextInt();  
  
        if (leido < 0)  
            throw new ExcepcionIntervalo("Error: no es positivo");  
    } while (leido < 0);  
    return leido;  
}
```

## 2.2 Propagación de excepciones

---

### Ejemplo (2/3):

Si la cabecera del método no incluyese la cláusula `throws`, toda la ejecución del programa acabaría en la lectura desde teclado del `int`, independientemente de desde dónde se hubiese invocado al método `leerEnteroPositivo()`.

Sin embargo, pese a que no se puedan encontrar en este fragmento de código las cláusulas `try/catch/finally`, la ejecución no termina de forma abrupta en este punto. Lo que sí que sucede es que el método termina sin llegar al `return`, es decir, el método sí termina pero el programa no.

La excepción se propaga porque en la cabecera del método aparece la cláusula `throws` de manera que, en otra parte del programa, podría encontrarse lo siguiente:



## 2.2 Propagación de excepciones

---

### Ejemplo (3/3):

```
Scanner tec = new Scanner(System.in).useLocale(Locale.US);

boolean salir = false;
int resp;

while (!salir) {
    try {
        resp = LeerEnteroPositivo(tec);
        System.out.println("Se ha leído el número " + resp);
        salir = true;
    }
    catch (ExcepcionIntervalo e) {
        System.out.println(e.getMessage());
    }
    finally {
        tec.nextLine();
    }
}
```