Projecte de Programació

Excepciones en Java

Prueba de Programas: Java

Java admite un tratamiento de errores y excepciones mediante las clases que heredan de la clase **`Throwable`**, que es la clase encargada de gestionar las situaciones excepcionales que se producen durante la ejecución de un programa

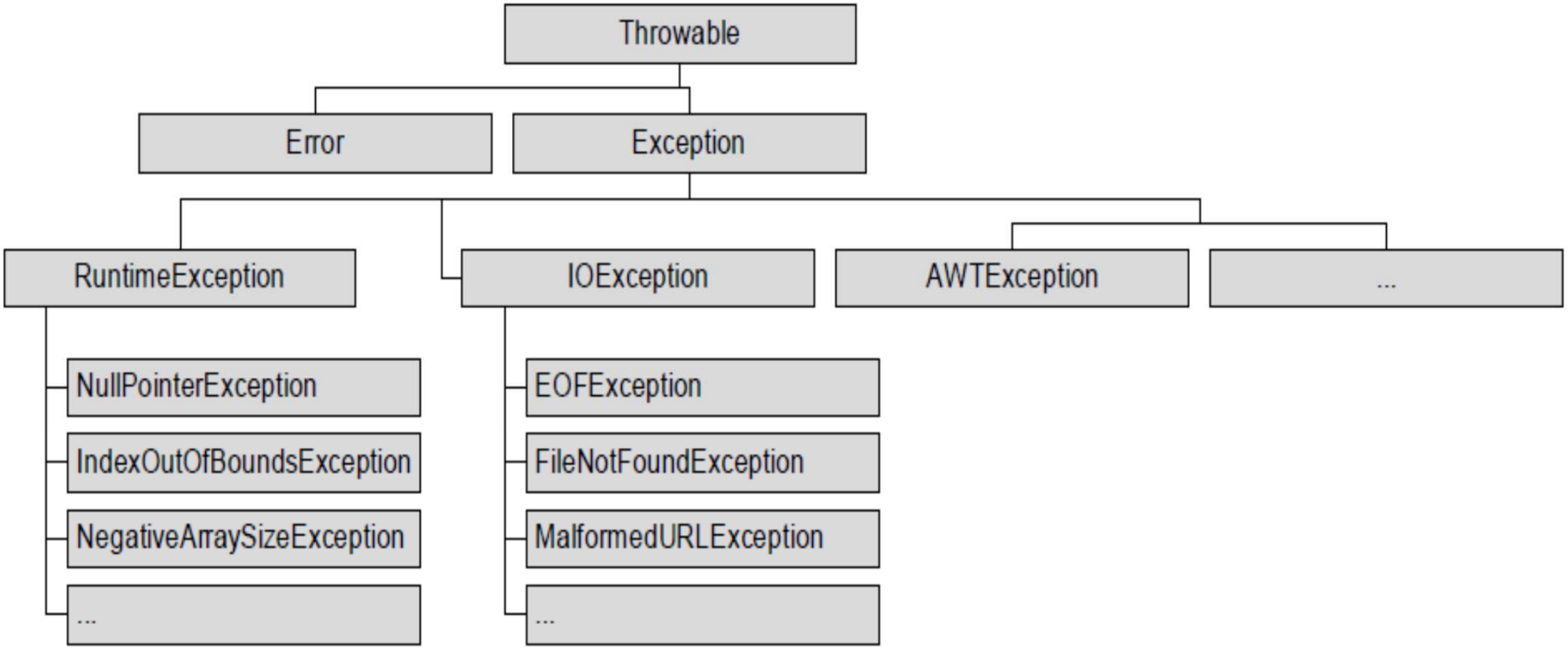
Hay dos tipos de situaciones excepcionales:

- Errores
  (**`java.lang.Error`**, subclase de **`java.lang.Throwable`**)
- Excepciones
  (**`java.lang.Exception`**, subclase de **`java.lang.Throwable`**)

Un buen programa debe gestionar correctamente todas o la mayor parte de estas situaciones que se puedan producir

Prueba de Programas: Java

Jerarquía de clases que heredan de la clase Throwable



El mecanismo de excepciones permite que las condiciones erróneas que un método puede señalar sean parte específica del contrato del método

Prueba de Programas: Java

- Errores
  (**`java.lang.Error`**, subclase de **`java.lang.Throwable`**)

En Java, los errores se refieren a situaciones que no deberían pasar nunca y que el programa ha de abortar: errores de compilación, del sistema o de la JVM

Ejemplos:

- La JVM no puede continuar por falta de recursos
- Un **`.class`** tiene problemas en su contenido
- Problemas de incompatibilidades entre clases dependientes
- Problemas *serios* de entrada/salida
- Un *thread* muere sin razón

Estos errores son irrecuperables, no dependen del programador y no tiene sentido tratarlos. Normalmente el programa se ha de parar

Prueba de Programas: Java

- Excepciones
  (**`java.lang.Exception`**, subclase de **`java.lang.Throwable`**)

En Java, las excepciones se refieren a situaciones extraordinarias que hay que tratar de manera especial. 2 tipos:

1) Excepciones implícitas: son las **`RunTimeException`**, relacionadas con errores de programación, p.ej. **`NullPointerException`**, **`IndexOutOfBoundsException`**, **`ArithmeticException`**, **`ClassCastException`**, etc.

El propio Java durante la ejecución chequea y lanza automáticamente las excepciones que derivan de **`RunTimeException`**

Sería posible capturar estos tipos de errores, pero el código se complicaría excesivamente si se necesitara chequearlos continuamente.

Errores y excepciones implícitas se denominan *unchecked exceptions* (no tienen que estar declaradas en los **`throws`** de los métodos)

Prueba de Programas: Java

- Excepciones
  (**`java.lang.Exception`**, subclase de **`java.lang.Throwable`**)

2) Excepciones explícitas: Son todas las demás clases derivadas de **`Exception`**. Estos errores tiene sentido tratarlos (capturarlos) y continuar el programa

Ejemplos:

- **`IOException`**: Errores en operaciones de entrada/salida
- **`NoSuchFieldException`**: No se encuentra un atributo
- **`NoSuchMethodException`**: No se encuentra un método
- Hay muchísimos más...

P.ej., una **`FileNotFoundException`**, no encontrar un fichero en el que leer o escribir algo debería ser recuperable. El programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado)

Prueba de Programas: Java

- Excepciones

Una excepción en un programa Java ha de ser *considerada* de alguna forma. De hecho, el compilador nos obliga: se denominan *checked exceptions*

Imaginemos que estamos programando un método `m()` que invoca otro método, y éste ejecuta una operación susceptible de generar una excepción de nombre *NomExcepción*

Entonces el método que estamos programando, `m()`, puede tratar o no la excepción

Prueba de Programas: Java

- Excepciones

Supongamos que **`m()`** **NO** trata la excepción *NomExcepción*

- En este caso, en la cabecera de **`m()`** hay que poner **`throws NomExcepción`** o el nombre de cualquiera de las superclases de *NomExcepción*, por ej:

```
Public void leerFichero (String fich) throws EOFException,
FilenotFound Exception
```

```
Public void leerFichero (String fich) throws IOException
```

- *No tratarla* significa que *si se produce una excepción con este nombre (o cualquiera de sus subclases), la excepción se pasa al método que la ha llamado*

Si no se quiere tratar NINGUNA excepción, basta con poner la cláusula **`throws Exception`** (la clase padre de todas las excepciones)

Si la excepción llega al **`main`** el programa aborta

Prueba de Programas: Java

- Excepciones

Supongamos que **`m()`** **SÍ** trata la excepción *NomExcepción*

- En este caso, no hay que poner nada en la cabecera de **`m()`**

- Hay que poner las instrucciones susceptibles de generar la excepción dentro de un bloque **`try`** con un **`catch`** para tratar las excepciones. Eventualmente se puede poner un bloque **`finally`**

A grandes rasgos: El código dentro del **`try`** está *vigilado*. Si se produce una situación anormal y se genera una excepción, el control pasa al bloque **`catch`** correspondiente a la excepción generada. Podemos poner tantos bloques **`catch`** como queramos. Finalmente, si está, se ejecutará el bloque **`finally`**, tanto si se ha generado una excepción como si no (es decir, *siempre*).

`finally` es necesario en los casos en que se necesita recuperar o devolver a su situación original algunos elementos

Prueba de Programas: Java

- Excepciones

Ejemplo de NO tratamiento de excepciones:

```
public void metodoExcepcion() throws Exception {
 // FileNotFoundException
 BufferedReader in = new
     BufferedReader(new FileReader("noexiste"));
     ...
}
```

[o alternativamente:

```
   throws IOException
   throws FileNotFoundException
]
```

Prueba de Programas: Java

- Excepciones: Ejemplo de tratamiento de excepciones:

```
public void metodoExcepcion() {
    try {
       int[] a = new int[3];
       // IndexOutOfBoundsException
       a[5]=10;
       // FileNotFoundException
       BufferedReader in = new BufferedReader(new FileReader("noexiste.txt"));
       ...
    }
    catch (IndexOutOfBoundsException e) {
        System.out.println(" *** IndexOutOfBoundsException... ");
    }
    catch (FileNotFoundException e) {
         System.out.println(" *** getMessage(): " + e.getMessage());
         System.out.println(" *** toString(): " + e.toString());
         e.printStackTrace();
   }
   finally {
         System.out.println(" *** finally... ");
    }
}
```

Prueba de Programas: Java

- Excepciones. Algunas consideraciones:

- No tiene mucho sentido tratar algunas **`RunTimeExceptions`**, como **`NullPointerException`** o **`IndexOutOfBoundsException`**, cada vez que accedemos a un objeto o trabajamos con un *Array*

- Hay algunos mensajes que podemos utilizar en un bloque **`catch`**:

  - **`String getMessage()`**: Retorna el mensaje asociado a la excepción

  - **`String toString()`**: Retorna la representación textual de la excepción

  - **`printStackTrace()`**: Retorna la representación textual de la pila de ejecución en el momento de la excepción

Prueba de Programas: Java

- Excepciones. Creación de excepciones propias

Hay que hacer una clase que herede de **`Exception`** (o de una de sus clases derivadas, la que se adapte mejor)

En ella suele haber una constructora sin argumentos y una constructora con una **`String`** de argumento (el mensaje explicativo de la excepción). Ambas constructoras han de invocar a la constructora de la clase padre

```
class myException extends Exception {
    public myException() {
      super();
    }

    public myException(String s) {
      super(s);
    }
}
```

Prueba de Programas: Java

- Excepciones. Creación de excepciones propias

Ejemplo: crear una excepción propia para proporcionar más información

```
class ExcepFichero extends Exception {
   private String filename;

   public ExcepFichero(String s) {
     super("Error en fichero " + s + ", formato
incorrecto");
     filename = s;
   }

   public String toString() {
     return "Excepción Fichero " + filename;
   }
}
```

Prueba de Programas: Java

- Excepciones. Creación de excepciones propias

Crearemos un objeto de la clase propia haciendo:

```
MyException me = new MyException("--Mensaje de error")
ExcepFichero mef = new ExcepFichero(nomFichero)
```

Si se quiere generar la excepción: `throw me`

```
public boolean leeDatos(File fichero1)throws
ExcepFichero {
      …
      //el formato correcto espera línea con contenido
      if (linea == null) throw new
ExcepFichero(fichero1.getName());
      …
   }
```

Al generar la excepción, el método acaba immediatamente sin retornar ningún valor

Prueba de Programas: Java

- Excepciones y Herencia

Si un método redefine otro de una superclase que utiliza **`throws`**, no tiene por qué lanzar las mismas excepciones de la clase padre. Puede lanzar las mismas o menos, pero NO puede lanzar nuevas excepciones ni excepciones de una clase más general

Imprescindible para que se mantenga la regla de compatibilidad de tipos del polimorfismo de subtipo

Prueba de Programas: Java

- Chequeo de errores en PROP:

  - La idea es compatibilizar el chequeo de errores convencional (ciertos métodos devuelven códigos de error que se chequean y gestionan en el entorno que llama al método) y el uso de excepciones

  - El uso de excepciones es OBLIGATORIO en PROP

  - Cuándo usarlas? Se suelen usar más para "errores inesperados y/o difícilmente controlables" (en general cuando intervienen elementos "externos" al sistema: E/S, interficie de usuario, …), mientras que para el resto de casos se usará chequeo convencional

Prueba de Programas: Java

- La instrucción **`assert`**

Se puede utilizar **`assert`** para depurar programas Java

Hay que insertar en el código:

```
assert ExpresionBooleana;

assert ExpresionBooleana1 : Expresion2;
```

Cuando se ejecuta la instrucción, se evalúa **`ExpresionBooleana`**. Si el resultado es falso, el sistema genera un **`AssertionError`** con un mensaje que contendrá el resultado de haber evaluado **`Expresion2`** (si la hay)

Por defecto la JVM NO ejecuta los **`assert`**, hay que invocarla con la opción **`-ea`** o **`--enableassertions`**