Universidad Complutense de Madrid
Facultad de Informática
Grado en Ingeniería Informática, Grupo en Inglés

Tecnología de la Programación

Tratamiento de excepciones

Simon Pickin,
Yolanda García, Alberto Díaz, Marco Antonio Gómez, Puri Arenas,

Excepciones

- Una excepción
  - es un evento producido durante la ejecución de un programa
    - debido a la existencia de un error
  - el cual interrumpe el flujo de ejecución natural
  - por ejemplo, cuando se accede a una posición no existente de un array
- El mecanismo de excepciones
  - trata los errores de ejecución que suceden en un programa
- En general, las excepciones sirven para:
  - indicar que un error ha ocurrido
  - transferir el control a un código de tratamiento de errores específico

Ejemplo: exception lanzada por el sistema

```
public class Overflow {

   public static void main(String[] args){
      int values[] = {1, 2, 3};
      for (int i = 0; i <= 3; i++)
         System.out.println(values[i]);
   }
}
```

```
1
2
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
at Overflow.main(Overflow.java:6)
```


Tipos de errores: Fatal y Recoverable

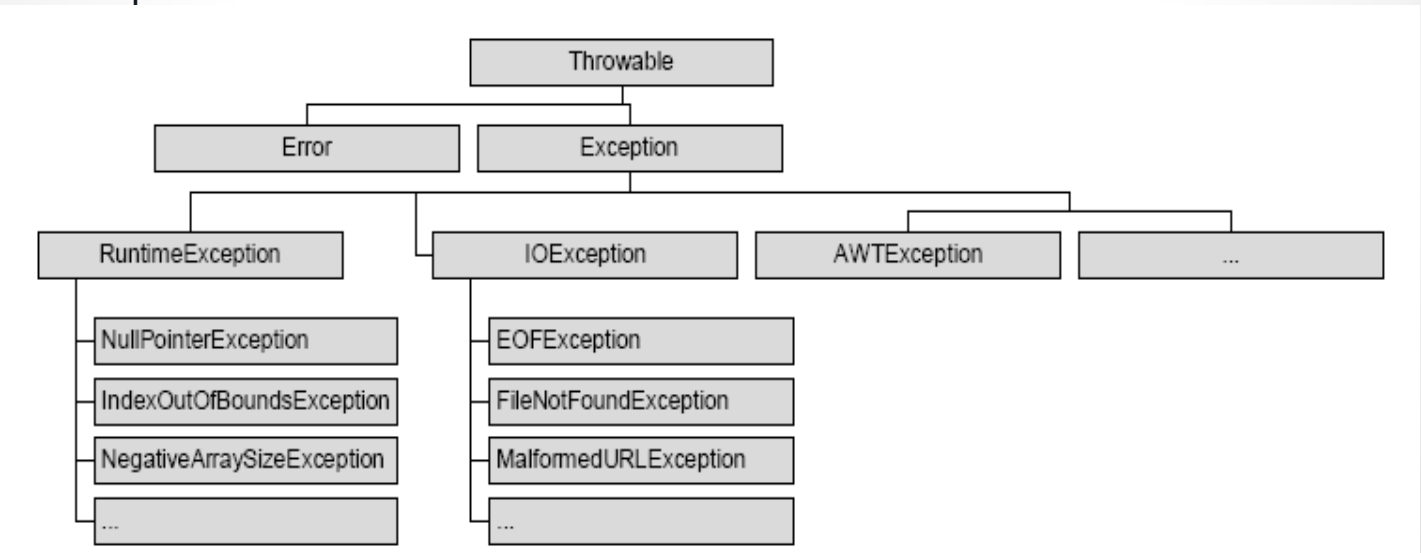
- Algunos errores son de tipo *Fatal*
  - la ejecución del programa debe parar
  - la terminación debe llevarse a cabo de forma controlada
  - un mensaje debe ser mostrado en la salida de error estándar
    - conteniendo información sobre el error ocurrido
- Otros errores son *Recoverable*
  - si ocurren, el programa debe tomar las medidas de recuperación oportunas, de forma que no es necesario parar el programa
  - errores de entrada: la recuperación generalmente incluye dar al usuario la oportunidad de corregir el error. Por ejemplo, si el usuario:
    - introduce el nombre de un fichero que no existe: pedirle otro nombre
    - introduce un carácter no numérico cuando se esperaba un `int`: pedir otra vez

Creación y tratamiento de excepciones

- Cuando ocurre un error, se crea un objeto que representa la excepción
  - El cual contiene información sobre el tipo de error y estado del programa.
- Si se crea por el programa en lugar de por el sistema subyacente, el objeto de la excepción debe ser entregado al propio sistema
  - Esta acción se conoce como lanzamiento (*throwing*) inicial de la excepción
- Al lanzarse una excepción, el sistema busca código que pueda tratarla
  - Comprueba, en orden inverso, cada uno de los métodos en la cadena de invocación, desde el `main` hasta el método donde se produce el error
  - Si una excepción es lanzada desde el `main`, el programa finaliza

Jerarquía de excepciones en Java

- La figura muestra parte de la jerarquía de clases Exception en Java
  - Las distintas clases se corresponden con los diferentes tipos de errores posibles.



Clases `Exception` en Java

- Clase `Throwable`
  - Clase de la cual derivan todas las excepciones
- Clase `Error`
  - Representa errores importantes externos a la aplicación
    - JVM, SO, hardware,…
  - Generalmente, los programas no los tratan
- Clase `Exception`
  - Define las excepciones que los programas suelen tratar
  - `IOException`, `ArithmeticException`, ...
  - `Exception` dividida en: `RunTimeException` and "*checked exceptions*"
    - la clase `IOException` es una "checked exception"
    - la clase `NullPointerException` es una "*unchecked exception*"

Clases `Exception` en Java

- Las clases en la jerarquía pertenecen a distintos paquetes
  - en `java.lang`
    - `Throwable`, `Exception`, `RuntimeException`, ...
  - en `java.io`
    - `EOFException`, `FileNotFoundException`, ...
  - ...
- Todas las excepciones heredan los siguientes métodos de `Throwable`
  - `String getMessage()`
    - Extrae el mensaje asociado a la excepción o `null`
  - `void printStackTrace()`
    - Imprime información de cada elemento en la pila de llamadas (*call stack*)
    - Ayuda a ver dónde se ha producido la excepción

Excepciones *Run-Time*

- Excepciones que surgen principalmente de errores de programación
  - Son lanzadas por el sistema
  - Generalmente no se tratan en el programa
    - Usualmente se corrige el propio error
    - Por ejemplo, referencia a null o índice fuera de rango
  - En ocasiones nos interesa atrapar estas excepciones
    - Por ejemplo, división por cero (`ArithmeticException`)
- El programador no está obligado a tratar las excepciones *run-time*
  - Así se evita que el código sea excesivamente complicado
    - elevado número de posibles excepciones run-time
    - cantidad de sitios en el código donde estas excepciones pueden suceder

Lanzamiento de excepciones

- Un método debe declarar las excepciones que pueden generarse
  - en su cuerpo
  - en el cuerpo de los métodos que se invocan en él

  a menos que contenga un bloque (`catch`) que trate la excepción.
- Esto se declara con la palabra reservada `throws` en su cabecera
- Ejemplo:

```
public void readFile(String file)
            throws EOFException, FileNotFoundException { ... }
```

o, ya que `EOFException`, `FileNotFoundException` hereda de `IOException`

```
public void readFile(String file) throws IOException { ... }
```

Captura (`catch`) de excepciones Java

- Si una excepción se genera pero no se trata en un método m_0
  - El compilador obliga a m_0 a declarar que lanza esa excepción
  - El compilador obliga a cualquier método m_1 que invoca m_0 a tratarla (`catch`) o a lanzarla (`throws`)
  - Si m_1 declara que lanza la excepción, el compilador obliga a cualquier método m_2, que invoque m_1 a tratarla (`catch`) o a lanzarla (`throws`), etc. sucesivamente
- El compilador no obliga a ninguna acción una vez la excepción es capturada
  - El bloque `catch` puede estar vacío (no se aconseja)
- Al menos se debe imprimir un mensaje con información que describa la excepción capturada

Los bloques `try-catch` en Java

- Las excepciones a procesar deben generarse en bloques `try`
  - Un bloque `try` contiene código donde una excepción puede ser lanzada
- Las excepciones son tratadas en los bloques `catch`
  - Un bloque `catch` contiene el código que se ejecuta al capturar la excepción
  - Se pueden utilizar varios bloques `catch` en el mismo bloque `try`
- Si se lanza una excepción en un bloque `try`
  - Si se captura en un bloque `catch`
    - Se para la ejecución del boque `try`
    - El control pasa al bloque `catch` y luego al código después del bloque catch
  - Si no se captura en un bloque `catch (*)`
    - El método lanza la excepción (*undeclared checked exception: compiler error*)
    - La ejecución del método finaliza sin devolver un valor de retorno
    - El control pasa al método que invocó la llamada

(*) o es lanzada fuera de un bloque `try`

Los bloques `try-catch` en Java

- Un bloque catch puede capturar una excepción específica
  - o un grupo de excepciones que deriven de la misma superclase
  - o un grupo de excepciones indicadas explícitamente (desde Java 7)
- Cuando existen varios bloques `catch`, el primero cuya excepción encaje con la excepción lanzada (la misma o su superclase) es utilizado
- Ejemplo de un bloque `try` con múltiples bloques `catch`

```
try {
      ...
} catch (IndexOutOfBoundsException e) {
   System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
   System.err.println("Caught IOException: " + e.getMessage());
}
```


Ejemplo: excepción sin `try-catch`

```
public class EjemploExcep  {

  public static void main(String[] args) {
          int a = args.length;
          System.out.println("a = " + a);
          int b = 42 / a;
          System.out.println("b = " + b);
          }
}
```

- Si a = 0, se produce un error y la ejecución se interrumpe

```
a = 0
java.lang.ArithmeticException: / by zero at
EjemploExcep.main(EjemploExcep.java:6)
Exception in thread "main" Process Exit...
```

Ejemplo: excepción con `try-catch`

```
public class EjemploExcep  {

    public static void main(String[] args) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            System.out.println("b = " + b);
        }
        catch (ArithmeticException e) {
            System.out.println("No dividas por 0 (" + e + ")");
        }
        System.out.println("La ejecución sigue ...");
    }
}
```

- Si a = 0, se produce un error pero la excepción no se interrumpe

```
a = 0
No dividas por 0 (java.lang.ArithmeticException: / by zero)
La ejecución sigue ...
```


Sentencias `try-catch-finally` en Java

- Algunas partes del código deben ser ejecutadas siempre
  - tanto si ocurre la excepción, como si no
  - tanto si se captura la excepción, como si no
  - por ejemplo: cerrar un fichero
  - por ejemplo: liberar un cerrojo en la programación concurrente
- Esto se consigue con el bloque `finally` que se puede colocar
  - después del bloque `try` y los bloques `catch`
  - después del bloque `try`, (sin que haya bloques `catch`)
- Un bloque `finally` contiene el código que debe ejecutarse antes de finalizar la ejecución del método.
  - Se ejecuta incluso si el bloque `try` contiene `continue`, `break` o `return`

Ejemplo: excepción con `try-catch-finally`

```java
public void metodo2() throws IOException {

        // Código que puede lanzar las excepciones IOException

} // Fin del metodo2
```

```java
public void metodo1(){
    try {
        metodo2();
    }
    catch (IOException e) {
        // Se ocupa de IOException simplemente dando aviso
        System.out.println(e.getMessage());
    }
    finally { // Sentencias que se ejecutarán en cualquier caso
    }
}
```

- Si `metodo2` lanza `IOException`, ésta será capturada por `metodo1`

Lanzar excepciones explícitamente en Java

- Lanzamiento implícito: creadas y lanzadas por la JVM
- Lanzamiento explícito: creadas con `new` y lanzadas con `throw`
  - Ejemplo:

```java
if (size == 0) {
    throw new EmptyStackException();
}
```

- Después de que una excepción sea lanzada,
  - tanto implícita como explícitamente, sin ser capturada en ese mismo método
  - finaliza la ejecución del método & el control pasa al método “invocante”
  - Si el método “invocante” no captura la excepción
    - la lanza, la ejecución finaliza, sin devolver un valor de retorno y el control pasa al método “invocante”
    - así, sucesivamente hasta que la excepción se captura o hasta que se lanza por el `main`, en cuyo caso la ejecución del programa finaliza.

Crear y lanzar nuevas excepciones en Java

- Muchos lenguajes permiten la creación de nuevas excepciones
  - pueden incluir los parámetros que se crean necesarios
- Para crear una excepción en Java,
  - La nueva clase debe heredar de `Exception` o una subclase de `Exception`
    - Se elige el supertipo más adeucada de la nueva clase
  - Se considera buena práctica incluir cinco constructores
    - Generalmente, cada uno simplemente invoca al constructor de la superclase.
- Es una mala práctica derivar excepciones de `RunTimeException` (*)
  - Programadores lo hacen para evitar declarar `throws` en las cabeceras de los métodos
  - A pesar de que conlleva un código mucho menos transparente
  - Sin embargo, esta falta de transparencia es buena para preguntas de examen (¡aviso!)

(*) Algunos autores tales como Bruce Eckel no están de acuerdo con esta posición estándar

Constructores estándar para las clases `Exception`

- Código incluido al crear una nueva excepción

```
public myException extends Exception {
    public myException() { super(); }
    public myException(String message){ super(message) }
    public myException(String message, Throwable cause){
        super(message, cause);
    }
    public myException(Throwable cause){ super(cause); }
    Exception(String message, Throwable cause,
              boolean enableSuppression, boolean writableStackTrace){
        super(message, cause, enableSuppression, writeableStackTrace)
    }
}
```

- Ver la documentación del API de Java para más detalles

Ejemplo: crear y lanzar nuevas excepciones

```
public class MiExcepcion extends Exception {

    public MiExcepcion(){
        super("Error malísimo...");
    }
}
```

// No sigue la práctica estándar
// de incluir 5 constructores

```
public class UnaClase {

    public void metodo() throws MiExcepcion {
        System.out.println("Lanzo mi excepción desde aquí.");
        throw new MiExcepcion();
    }
}
```

- Define un nuevo tipo de excepción llamado `MiExcepcion`
- Define un método que crea y lanza excepciones del nuevo tipo

Ejemplo: crear y lanzar nuevas excepciones

```
public class EjemploExcep  {

    public static void main(String[] args) {

        UnaClase c = new UnaClase();
        try {
            c.metodo(); // Invoco al método que eleva la excepción
        }
        catch(MiExcepcion e) {
            System.out.println("La tengo! " + e);
        }
        System.out.println("... y sigo.");
    }
```

- Captura de excepciones en el `main`:

```
Lanzo mi excepción desde aquí.
La tengo! MiExcepcion: error malísimo...
... y sigo.
```

Captura y re-lanzamiento de excepciones

- En algunos casos se requiere capturar y re-lanzar una excepción

```
catch(Exception e1) {
  // do something with exception e1
  throw e2;  // e2 may or may not be the same exception as e1
}
```

- Casos en los que puede realizarse esto:
  - Añadir información específica a la excepción: depuración.
  - Llevar a cabo el tratamiento que no queremos situar en el mismo método que el propio tratamiento del error (e.g. logging).
  - Para "envolver" una excepción de bajo nivel en otra de alto nivel más apropiada.
    - E.g. `SQLException` debe ser envuelta en una excepción de alto nivel, o simplemente en una `RunTimeException`

Sentencia `try-with-resources`

- Ejemplo de la lectura de un flujo de bytes y la impresión del resultado en la salida estándar, pre-Java 7:

```
private static void printFile() throws IOException {
  InputStream input = null;
  try {
    input = new FileInputStream("file.txt");
    int data = input.read();              // may throw IOException
    while(data != -1){
      System.out.print((char) data);
      data = input.read();                // may throw IOException
    }
  } finally {
    if(input != null){ input.close(); } // may throw IOException
  }
}
```

Sentencia `try-with-resources`

- ¿Qué pasa si excepción lanzada en el `try` y también en el `finally`?
  - Cuál de la dos excepciones se propaga por la pila de llamadas?
  - *Respuesta*: se lanza la excepción producida en el bloque `finally`
    - La excepción lanzada en el bloque `try` se "suprime"
- Java 7 introdujo la sentencia `try-with-resources`
  - Recursos creados y asignados a variables como parte de la sentencia `try`
  - Los recursos deben implementar la interfaz: `java.lang.AutoCloseable`
- Significado de la sentencia `try-with-resources`
  - Los recursos se cierran automáticamente cuando el bloque `try` finaliza
  - Si se lanza una excepción en el `try` y también en el `finally`
    - La excepción del `try` se propaga por la pila de llamadas, el de `finally` se suprime
    - Las excepciones suprimidas se pueden obtener mediante `getSuppressed`

Sentencia `try-with-resources`

- Ejemplo anterior utilizando `try-with-resources`:

```
private static void printFile() throws IOException {
  try(FileInputStream input = new FileInputStream("file.txt")) {
    int data = input.read();
    while(data != -1){
      System.out.print((char) data);
      data = input.read();
    }
  }
}
```

Sentencia `try-with-resources`

- Ejemplo con varios recursos en el bloque `try-with-resources`:

```
private static void printFile() throws IOException {
  try( FileInputStream input = new FileInputStream("file.txt");
       BufferedInputStream bufferedInput = new BufferedInputStream(input)
     ) {
    int data = bufferedInput.read();
    while(data != -1){
      System.out.print((char) data);
      data = bufferedInput.read();
    }
  }
}
```


Declaración de excepciones en interfaces

- Un interface contiene la declaración de métodos que incluyen la cláusula throw
  - El método correspondiente en la clase que implemente el interface, no está obligado a declarar que lanza la misma excepción
- ¿Por qué no?
  - Un excepción es un tipo especial de salida
  - En una relación de subtipo, los tipos de salida de los métodos son covariantes
    - Por ejemplo: Se permite que el tipo de salida de un método (subtipo) sea más restrictivo que el tipo de salida del método (supertipo) correspondiente.