Unidad 6: Tratamiento de errores en tiempo de ejecución

Metodología de la Programación

Curso 2021-2022

© *Candi Luengo Díez , Francisco Ortín Soler y José Manuel Redondo López, Alberto M. Fernandez Álvarez*

Bibliografía

- **Programación orientada a objetos con Java. 6th Edición**
  David Barnes, Michael Kölling.
  Pearson Education. 2017

  Capítulo 14: Tratamiento de errores

En esta unidad …

- Conoceremos que sucede **cuando se detecta un error** en el programa.
- Aprenderemos a **manejar los errores de ejecución.**
- Conoceremos las **características** (básicas y avanzadas) del **manejo de errores en Java.**
- Aprenderemos como usar los **assertos.**

Principales conceptos

- Programación defensiva
  - Prever que las cosas pueden salir mal
- Lanzamiento de excepciones
- Manejo de excepciones
- Informe de errores

Sobre los defectos en el software

- Los defectos del software no son simples "molestias"
  - Un producto con defectos deja de usarse.
  - Puede producir enormes pérdidas (incluso de vidas humanas).
  - Producen retrasos en las entregas, enormes dificultades técnicas... y son responsables de muchas horas extraordinarias.
- Hay que evitar producir programas defectuosos
  - Fundamentos del desarrollo (análisis, diseño, buena codificación).
  - Depuración.
  - Buen tratamiento de errores.
  - Pruebas y documentación.

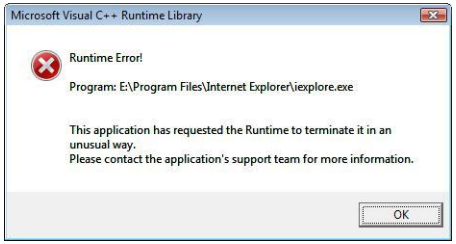
Algunas causas de situaciones de error

- Implementación incorrecta.
  - No cumple con la especificación.
- Solicitar a un objeto que haga algo que es incapaz de hacer.
  - Invocar al método `get` de una colección con un índice no válido.
  - Invocar a un método con un objeto null, o acceder a una posición inexistente de un `array`
- Acciones erróneas del usuario
  - Introducir una letra al teclear un número.

Algunas causas de situaciones de error

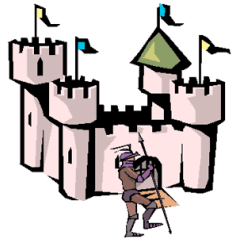
- Dejar a un objeto en un estado inconsistente.
  - Por ejemplo, cuando se reutiliza una clase (probablemente mediante la herencia).
- Violación del contrato de una clase o método
  - *Ejemplos*: invocar `pop()` sobre una pila vacía, invocar a `raizCuadrada()` pasando un número negativo.
- Condiciones de ejecución que imposibilitan el cómputo
  - *Ejemplos*: no queda memoria o espacio en disco, falla el hardware, pérdida de ficheros, interrupciones de red…

Resumiendo…



- El compilador de Java detecta muchos errores estáticamente (en tiempo de compilación).
  - Por ejemplo, los errores de sintaxis y los errores de tipo.
- Pero, hay errores que se detectan **en tiempo de ejecución**, debido a su carácter dinámico.
  - Pasar un mensaje a una variable objeto que vale null.
  - División por cero.
  - Acceso a un Array con un índice fuera de los límites.
  - Valores de argumentos inválidos (p.e., factorial de un número negativo).
- Es necesario disponer de un mecanismo para manejar los errores de ejecución.

Programación defensiva

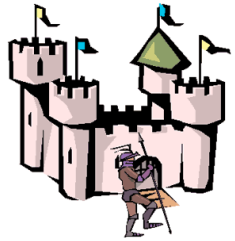


- La programación defensiva tiene por objeto garantizar el funcionamiento permanente de un trozo de software, a pesar del uso imprevisible.
  - La aplicación debe comportarse de una manera consistente y predecible incluso en el caso de condiciones inesperadas.
    - NO debe finalizar “buscamente” si se produce un error .

```
Exception in thread "main" java.lang.RuntimeException: The file does not exist
        at uo.mp.exception.FileReader.<init>(FileReader.java:16)
        at uo.mp.exception.Accounter.readLinesOnFile(Accounter.java:12)
        at uo.mp.exception.Accounter.processFile(Accounter.java:7)
        at uo.mp.exception.Main.run(Main.java:15)
        at uo.mp.exception.Main.main(Main.java:8)
```

  - La programación defensiva debe tener en cuenta los errores de ejecución.

Programación defensiva



- Un mecanismo para controlar los errores de ejecución debe ser:
  - **Robusto**: El mecanismo debe obligar al programador a escribir código para manejar los errores de ejecución (p. ej., ¿qué sucede si un archivo no se puede abrir?).
  - **Reusable**: Podemos reutilizar el código, implementando diferentes manejadores de error en diferentes escenarios.
  - **Extensible**: Permita añadir nuevos tipos de errores y extender los ya existentes.
    - ¿Podemos dar suficientes detalles sobre el error?
    - ¿Podemos precisar la naturaleza del error?

Mecanismos existentes

- Código de cliente y código de proveedor.
  - Un método (el cliente) llama a otro método (el proveedor).
  - Cómo está siendo notificado el "cliente" sobre un error en el “proveedor”.

```
void m() {
    list.remove( 3 );
}
public class LinkedList {
  public Object remove(int index) {
     ...
  }
}
```

← ¿falla o funciona ?

Mecanismos existentes

- Analizamos los mecanismos clásicos* utilizados para detectar los errores de ejecución.
  - **Valores de retorno especiales**
  - **No notificar el error**
  - **Finalizar la ejecución del programa**
  - **Comprobación a priori**
  - **Comprobación a posteriori**
    - **Ignorar el error**

* clásicos → antes de que aparecieran las excepciones

Mecanismos existentes

- **Valores de retorno especiales** : Un valor especial es retornado.

```
public class LinkedList {
  public Object remove(int index) {
   if (isEmpty())
         return null;
   // ...
  }
}
```

  - Es reusable y extensible pero…
  - En primer lugar, no es robusto:

```
((Person)list.remove(0)).getAge();
```

Valores de retorno especiales

- **Valores de retorno especiales**
  - En segundo lugar, no siempre es posible encontrar un valor especial.

```
public class IntStack {
  public int pop() {
   if (isEmpty())
         return ?;
   // ...
  }
}
```

Si coleccionamos int ¿Qué valor se devuelve?
- **0?**
- **1?**
- **-1?**
- **NaN?**
- **…**

¡Ninguno es posible realmente! ☹

No notificar el error

- **No notificar el error**: La operación solicitada no se realiza y no se notifica al cliente.

```
public class ArrayList {
  public void set(int index, Object element) {
   if (index >=0 && index < size())
         elements[index]=element;
  }
  // ...
}
```

  - No es robusto porque el cliente piensa que no hay errores.
  - No es reusable ni extensible porque el error no se notifica.

Finalización de la ejecución

- **Terminación del programa**: La ejecución se para cuando se detecta un error de ejecución.

```
public class ArrayList {
  public void set(int index, Object element) {
   if (index < 0 || index> = size())
         System.exit(-1);
   elements[index]=element;
  }
  // ...
}
```

  - No es reusable y tampoco es extensible.

Comprobación a priori

- **Comprobación a priori** : Permite comprobar el estado antes de llamar al método.
  - Esta condición debe usar, por lo tanto, parte de la interfaz de la clase (p.e., `isEmpty()`) (depende del diseño)

```
if (!stack.isEmpty()) {
    stack.pop();
}
```

  - Sí es reusable y extensible.
  - No es robusto, porque el cliente no está obligado a comprobar el estado antes de la invocación del método.

Comprobación a posteriori

- **Comprobación a posteriori**: Comprueba si ocurrió un error en la llamada del método anterior.
  - Esta condición debe usar, por lo tanto, parte de la interfaz de la clase (p.e, `fail()`) (depende del diseño)

```
int a;
do
  a = console.readInt();
while (console.fail());
```

  - Sí es reusable y extensible.
  - No es robusto, porque el cliente no está obligado a comprobar el estado después de la invocación del método.

Comparación

Técnica	Robusto	Reusable	Extensible	Aplicable siempre
Valores de retorno	✗	✓	✓	✗
No notificar el error	✗	✗	✗	✓
Terminación del programa	n/a	✗	✗	✓
Comprobación a priori	✗	✓	✓	✗
Comprobación a posteriori	✗	✓	✓	✗
Excepciones	✓	✓	✓	✓

Excepciones

- Debido a la falta de mecanismos para la gestión de los errores de ejecución, Java incorpora el **manejo de excepciones en tiempo de ejecución.**
- El código de manejo de excepciones se separa
  - Se detecta el error y se "lanza" una excepción (**proveedor**).
  - Se maneja el error del modo que más interese (**cliente**).

  ⇒ Implica una **alta reusabilidad y extensibilidad**
- El manejo de excepciones fuerza el tratamiento de los errores.
  - En caso contrario el programa termina anormalmente

  ⇒ Implica **gran robustez.**

Excepciones

- ¿Porqué son reusables?
  - El programador añade el control de excepciones donde sea necesario.
- ¿Porqué es extensible?
  - El programador puede añadir nuevos tipos de excepciones.
- ¿Porqué es robusto?
  - Excepciones comprobadas: El compilador no permite que el programador ignore la excepción y le obliga a añadir código para manejar la excepción.
  - Excepciones no comprobadas: El programa “rompe” si no hay código para manejar la excepción. Este error aparece en la etapa de desarrollo.

Lanzar excepciones

- Cuando se detecta un error en tiempo de ejecución, se debe lanzar una excepción.
- Las **excepciones** son **objetos** derivados de la clase **`Exception`**

```
new Exception("Ha ocurrido un error")
```

- Java ofrece diferentes excepciones para ser utilizadas por el programador.
  - Se pueden añadir nuevos tipos de excepciones (clases).
- Las excepciones son lanzadas con la palabra reservada **`throw`**

```
throw new Exception("Ha ocurrido un error")
```

Lanzar excepciones


```
public class ArrayList {
  public Object get(int index){
    if (index < 0 || index> = size())
      throw new IllegalArgumentException(
                    "Indice fuera de los límites");
    return elements[index];
  }
  public static void main(String[] args) {
    ArrayList v=new ArrayList();
    int a = (int)v.get(-1);
    System.out.println("Número de elementos: " +
                    v.size() );
  }
}
```

¿Estoy fuera del array?

¿Se notifica el problema al que llama?

¿Cuál es el comportamiento en tiempo de ejecución si se lanza una excepción?

Excepciones no capturadas

- La ejecución anterior produce una excepción no capturada (no hay código para manejar la excepción).
- Bajo estas circunstancias, el programa **termina anormalmente**…
- … y la siguiente **información se muestra en la salida estándar de errores** (`System.err`):
  - La cadena de texto que representa a la excepción (`toString`)
  - La traza de la pila cuando la excepción fue lanzada

Exception in thread "main" java.lang.IllegalArgumentException: Index out of bounds

at exceptions.ArrayList.get(ArrayList.java:25)

at exceptions.ArrayList.main(ArrayList.java:86)

Manejo de excepciones

- Las excepciones en Java son manejadas con bloques **try** y **catch**
- Bloque **Try**: Código que puede lanzar una excepción
- Bloque **Catch**: Código que maneja la excepción

```
int index =0;
try {
  ArrayList vi=new ArrayList();
  index = askForInt();
  int value = (int)vi.get(index);
  System.out.println("Valor de" +index+ " elemento: "+value);
}
catch(IllegalArgumentException e) {
  System.err.println(index+" está fuera de los límites.");
}
System.out.println(index);
```

Resumiendo …

- Las excepciones son un mecanismo de tratamiento de errores, disponible en diversos lenguajes OO (Java, C++, C#, Smalltalk...)
- El lanzamiento de una excepción es la manera más efectiva que tiene un objeto servidor para indicar que no puede completar la solicitud del cliente.
  - Usa un tipo especial de retorno que el cliente no puede ignorar (el cliente no puede ignorar los errores)
    - El flujo normal de ejecución es interrumpido
  - Se ponen en marcha los mecanismos específicos de recuperación de errores.

Manejo de excepciones

- Ejecución sin excepciones

```
int index =0;
try {
  ArrayList vi=new ArrayList();
  index = askForInt();
  int value = (int)vi.get(index);
  System.out.println("Valor de " + index +
                     " elemento: " + value);
}
catch(IllegalArgumentException e) {
  System.err.println(index+ " fuera de los límites");
}
System.out.println(index);
```

No se lanza la excepción

La ejecución continúa

Manejo de excepciones

- Ejecución lanzando la excepción `IllegaArgument`

```
int index =0;
try {
   ArrayList vi=new ArrayList();
   index = askForInt();
   int value = (int)vi.get(index);
   System.out.println("Valor de " + index +
                      " elemento: " + value);
}
catch(IllegalArgumentException e) {
   System.err.println(index+ "fuera de los límites.");
}
System.out.println(index);
```

Se lanza la excepción `IllegalArgument`

Se maneja la excepción!

La ejecución continúa

Manejo de excepciones

- Ejecución lanzando otra excepción (diferente a `IllegaArgument`)

```
int index =0;
try {
  ArrayList vi=new ArrayList();
  index = askForInt();
  int value = (int)vi.get(index);
  System.out.println("Valor de " + index +
                     " elemento: " + value);
}
catch(IllegalArgumentException e) {
  System.err.println(index+"fuera de los límites.");
}
System.out.println(index);
...
}
```

Se lanza otra excepción

No se maneja la excepción!

La excepción no capturada es, a su vez, lanzada por este código!

Múltiples bloques Catch

- Un bloque try puede lanzar diferentes tipos de excepciones.
- Podemos estar interesados en manejar diferentes errores de distintas maneras.
- Los bloques catch **se comprueban en el orden en que se escriben hasta que se encuentra una coincidencia**
- De lo contrario (si no se encuentran coincidencias) la excepción se propaga

```
try { Código del algoritmo
    }
catch(IllegalArgumentException e) {
       Código para manejar IllegalArgumentException    }
catch(NullPointerException e) {
       Código para manejar NullPointerException            }
catch(IOException e) {
       Código para manejar IOException                  }
La ejecución continúa
```

Múltiples bloques Catch repetidos

```
try { Código del algoritmo

}
catch(IllegalArgumentException e) {
    Código repetido para manejar la excepción
}
catch(IllegalStateException e) {
    Código repetido para manejar la excepción
}
La ejecución continúa
```

Evitar la repetición

```
try { Código del algoritmo
}
catch(IllegalArgumentException | IllegalStateException e {
    Código para manejar la excepción
}
La ejecución continúa
```

Usando Polimorfismo

- El polimorfismo puede ser usado para **generalizar un grupo de excepciones que se desea capturar**
- Pregunta: ¿Qué excepciones serán capturadas en cada bloque `catch`?

```
try {                Código del algoritmo                }
catch(IllegalArgumentException e) {
    Código para manejar IllegalArgumentException
}
catch(RuntimeException e) {
    Código para manejar RuntimeException
}
catch(Exception e) {
    Código para manejar Exception
}
La ejecución continúa
```

`catch(Throwable t)` captura cualquier error ¿Por qué?

Usando Polimorfismo

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                java.lang.NullPointerException
                java.lang.AritmeticException
                java.lang.IndexOutOfBoundsException
                    java.lang.ArrayIndexOutOfBoundsException
            java.io.IOException
                java.io.EOFException
                java.io.FileNotFoundException
```

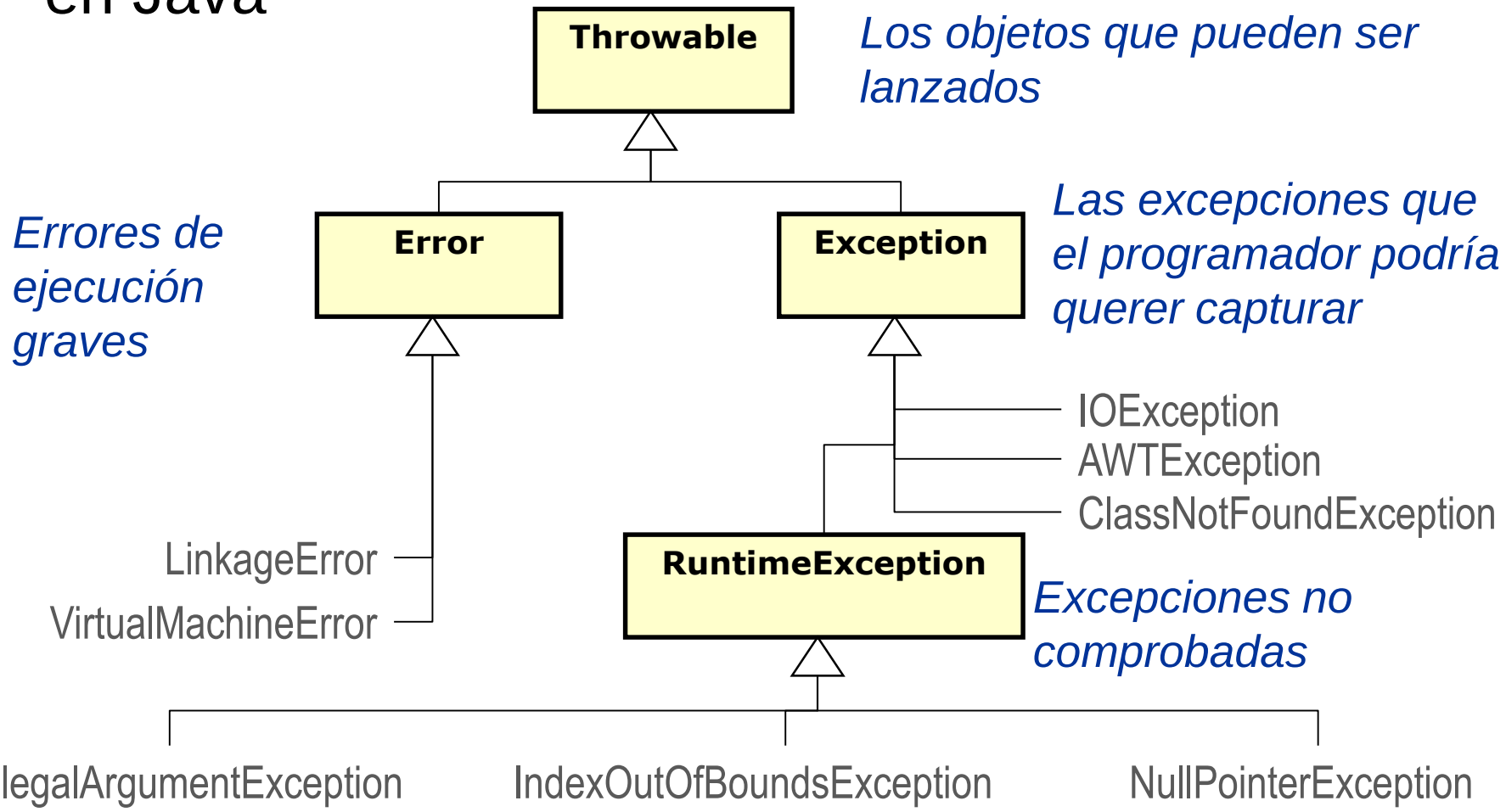
Usando Polimorfismo

- Pregunta: ¿Ahora, qué excepciones serán capturadas en cada bloque `catch`?

```
try {                  Código del algoritmo
}
catch(Exception e) {
    Código para manejar Exception
}
catch(RuntimeException e) {
    Código para manejar RuntimeException
}
catch(IlleaglArgumentException e) {
    Código para manejar IlleaglArgumentException

}
La ejecución continúa
```

Clases de excepciones

- Esta es la jerarquía de clases de excepciones en Java



Tipos de excepciones

- En Java se pueden lanzar los siguientes objetos:
  - Errores
  - Excepciones no comprobadas
  - Excepciones comprobadas

  - **Errores** (`Error`): errores en tiempo de ejecución graves que normalmente el programador no controla (p.e., `virtual machine error`)

Tipos de excepciones

- **Excepciones no comprobadas** (`RuntimeException`): Excepciones que son el resultado de un problema de programación o un error del sistema.
  - Es posible que el código del cliente **no** se pueda recuperar de ellos o los maneje de alguna manera.
  - Ejemplos: `IllegalArgumentException` o `NullPointerException`
- **Excepciones comprobadas** (`Exception`, no derivado de `RuntimeException`). Se espera que el código del cliente se pueda recuperar de un problema en tiempo de ejecución. Se puede lanzar una `IOException`, representando:
  - Que no hay suficiente espacio en disco
  - Que no tenemos permisos para crear ese fichero

Excepciones comprobadas

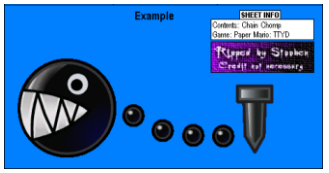
- El siguiente método

```
private static File createTemporaryFile(String nameOfFile,
                    String extension, String directory) {
  File folder = new File(directory);
  return File.createTempFile(nameOfFile, extension,
                                    folder);
}
```

- Es rechazado por el compilador, mostrando el mensaje:

  Unhandled exception type IOException

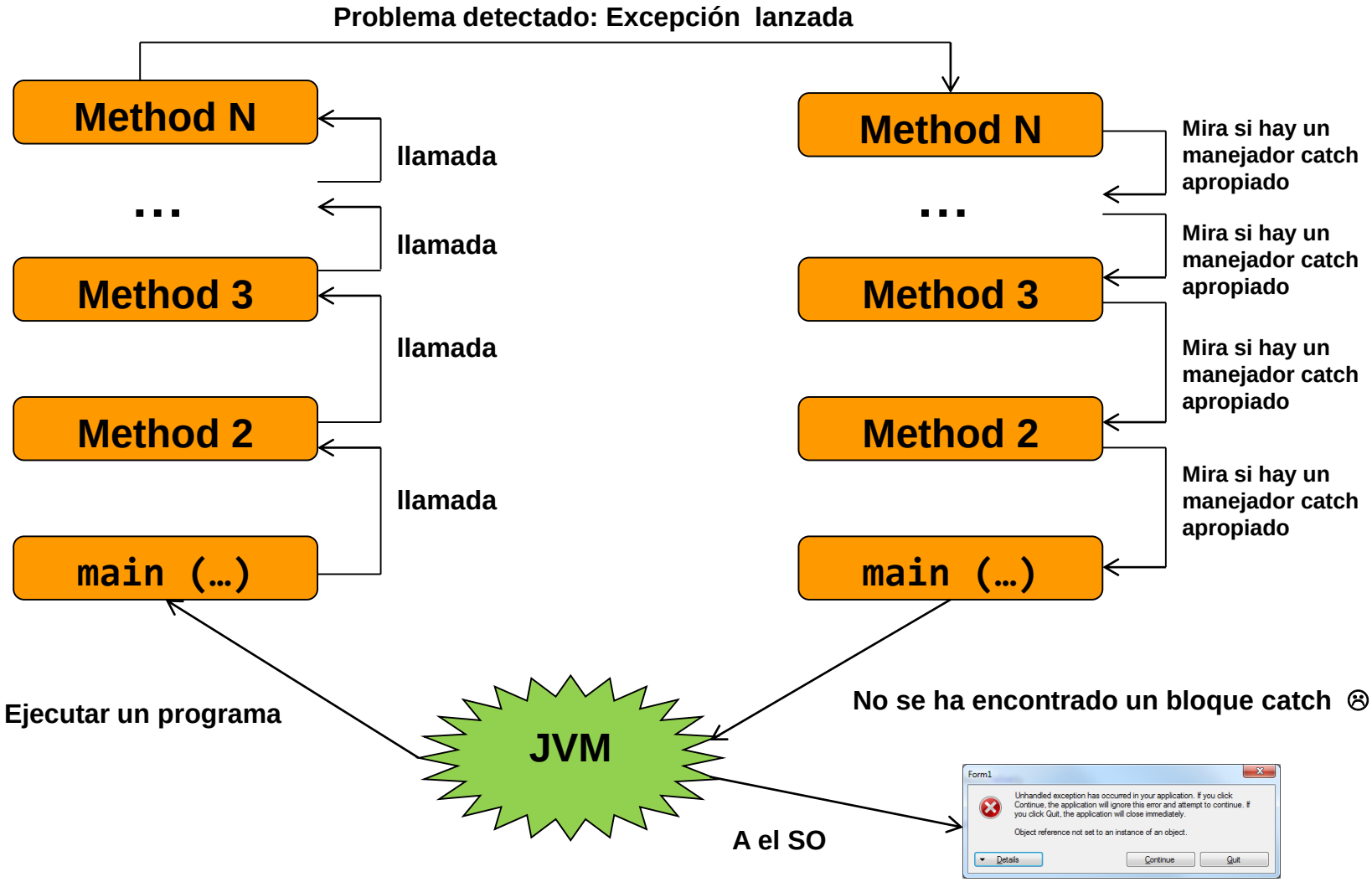
ThrowS



- Cuando un método lanza una excepción comprobada, es **obligatorio declararla** mediante la cláusula `throws`

```
private static File createTemporaryFile(String nameOfFile,
                String extension, String directory)
                    throws IOException {
  File folder = new File(directory);
  return File.createTempFile(nameOfFile, extension,
                                folder);
}
```

Resumen: Manejo excepciones

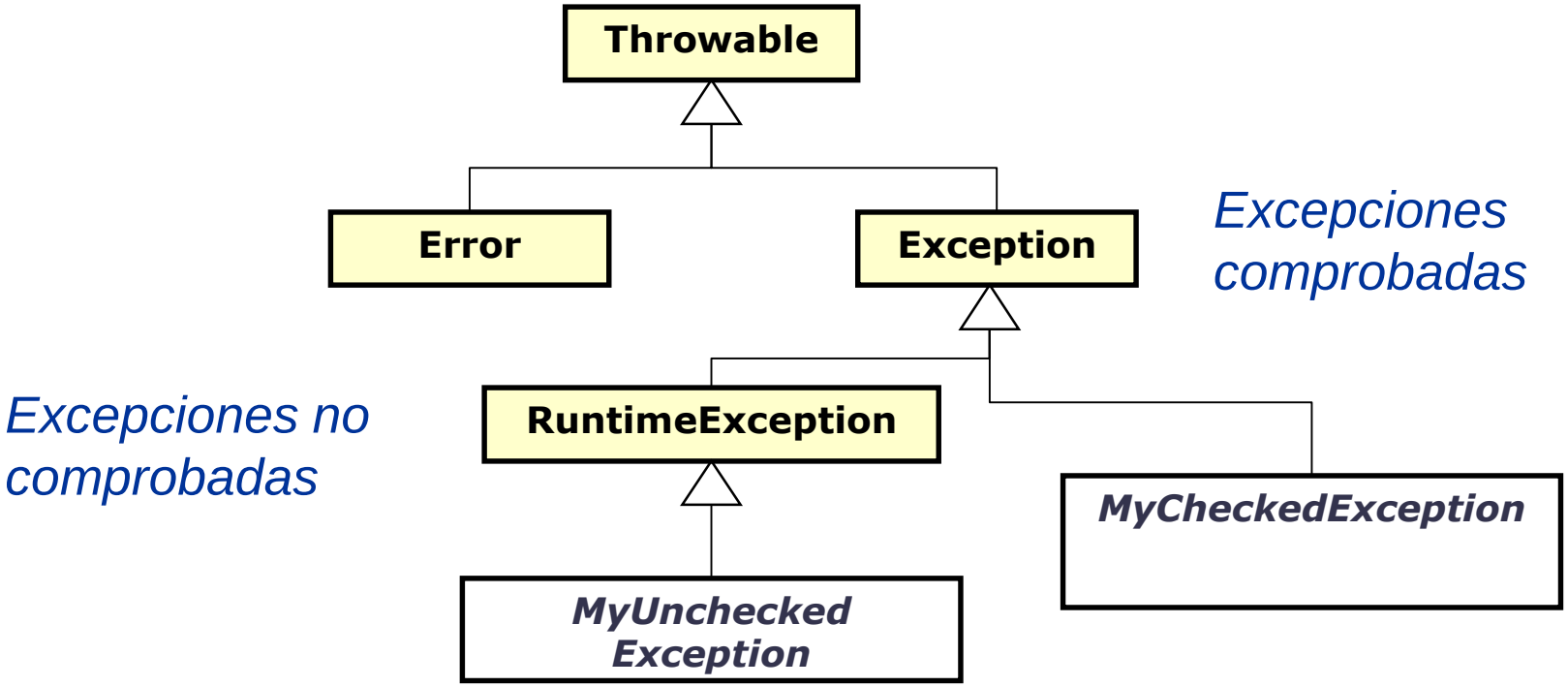


Definiendo nuevas excepciones

- Los objetos Exception permiten:
  1. Cambiar el flujo de ejecución.
  2. Enviar la información del error al manejador de la excepción.
- Podría haber escenarios en los que las excepciones existentes no describen la naturaleza del problema
- Entonces, podría ser interesante poder **definir nuevos** tipos (clases) de **excepciones**
- En estas clases, se podría añadir nueva información (con nuevos atributos)
- El método **`toString`** debe estar redefinido (se utiliza para imprimir las excepciones no capturadas)

Comprobadas o No comprobadas

- Cuando definimos nuevas excepciones, hay que decidir si son comprobadas o no comprobadas
  - No comprobadas serán subclases de **`RuntimeException`**
  - Comprobadas serán subclases de **`Exception`**



Comprobada o no comprobada

- Depende del diseño, hay que pensar…
- Usaremos generalmente excepciones no comprobadas.
  - Señalan un error que impide que el programa continúe la ejecución:
    - La red está “caída”, el servidor de la base de datos está “caído”, el archivo de configuración está dañado, etc.
    - Por lo general, son errores que tienen que ser resueltos por el personal técnico, no por el usuario del programa.
    - Errores del Sistema o de programación.
- Usaremos excepciones comprobadas cuando el programa o el usuario pueden reaccionar ante el error.
  - Por lo tanto, el programa puede continuar su ejecución.

Ejercicio: crear una nueva excepción

- Java ofrece una excepción no comprobada `IndexOutOfBoundsException` que utiliza en sus colecciones, incluidos los arrays.
- Sin embargo, su información solo es un mensaje
- Vamos a añadir una nueva excepción para nuestro `ArrayList`, que incluya la siguiente información
  1. Un mensaje de error
  2. El índice usado para acceder al `ArrayList`
  3. Una referencia al `ArrayList`

Crear una nueva excepción

```
public class ArrayListIndexOutOfBoundsException
                              extends RuntimeException {
  private String message;
  private int index;
  private ArrayList arrayList;

  public ArrayListIndexOutOfBoundsException(
                    String message, int index,
                    ArrayList arrayList) {
    this.message = message;
    this.index = index;
    this.arrayList = arrayList;
  }
```

Crear una nueva excepción

```
  public String getMessage() {
    return message;
  }
  public int getIndex() {
    return index;
  }
  public ArrayList getArrayList() {
    return arrayList;
  }
  @Override
  public String toString() {
    return getMessage() + getIndex() +
           getArrayList() ;
  }
```

Manejo de recursos…
y duplicación de código

- Analiza el siguiente (pseudo)código
- ¿Hay algo mal? ¿Se podría mejorar?

```
// Se abre un fichero de texto para escritura
FileWriter file = new FileWriter( … );
try {
   file.write( … ); // Escribe en el fichero
} catch(IOException e) {
   exception handling
   file.close();    // Cierra el fichero
} catch(Exception e) {
   exception handling
   file.close(); // Cierra el fichero
}
file.close(); // Cierra el fichero
```

Un recurso es…

- Algo que sigue este patrón de uso:
  - Create
  - Use
  - Close

```
Resource r = new Resource(...);
    r.doSomething();
r.close();
```

- La última llamada es obligatoria (close)
  - El programador debe asegurarse de que el recurso quede cerrado en cualquier circunstancia.
- Ejemplos de recursos:
  - Ficheros, conexiones de red, streams, etc.

Recursos y excepciones

- Los recursos son comúnmente asignados y liberados
  - En el código anterior, el fichero es el recurso
  - Con ficheros, Asignar = Open, Liberar = Close
- Se debe liberar el recurso
  - Si en el bloque try se lanza una excepción
  - Si en el bloque try no se lanza ninguna excepción
- Esta es la razón por la cual, la sentencia `file.close()` se duplica en el código anterior
- Este es un problema común cuando se trata de recursos y excepciones
- Así, Java añadió **`finally`** al lenguaje

Finally

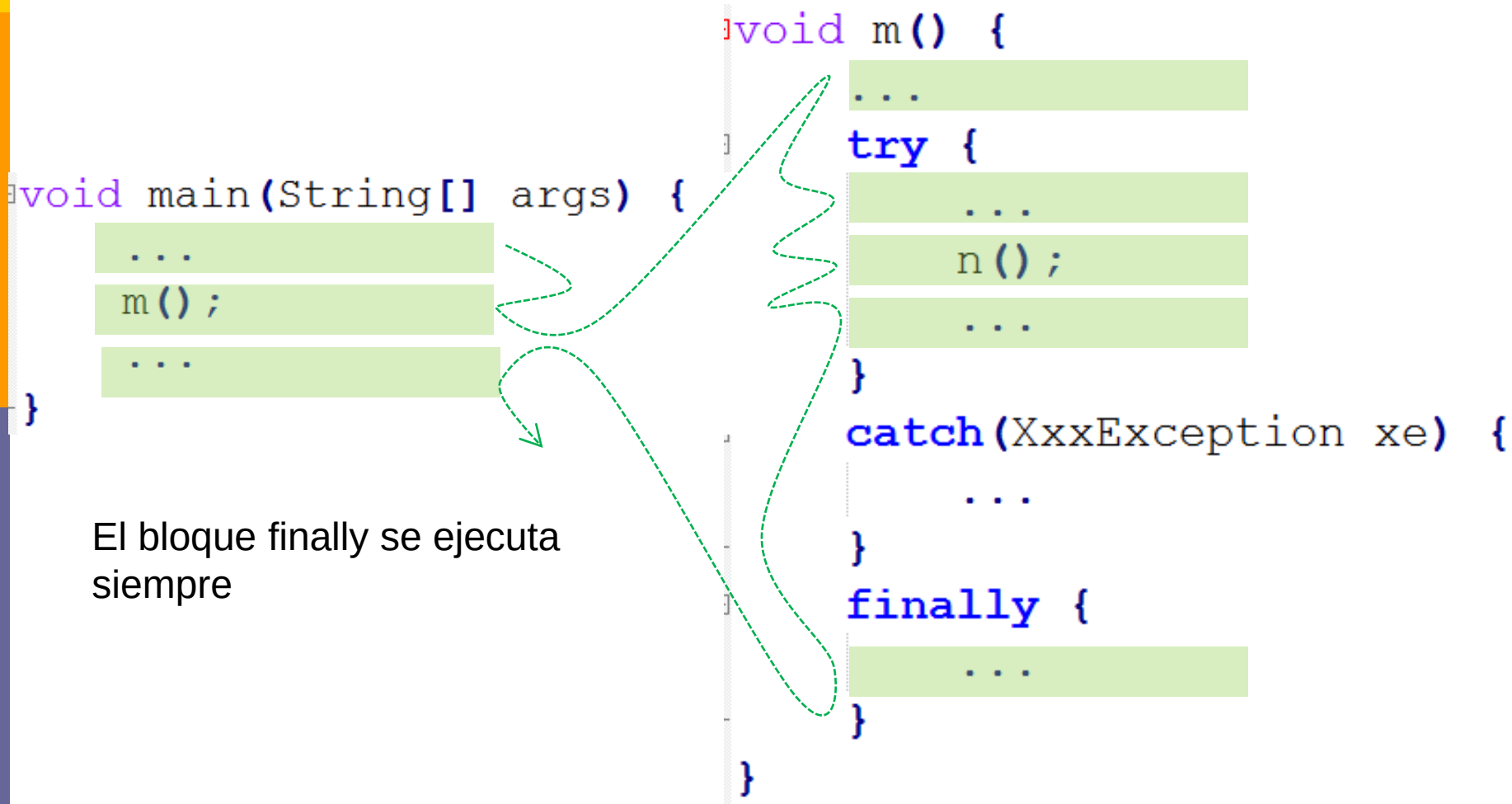
- Un bloque finally se ejecuta siempre
  - Después del bloque try si no hay excepciones
  - Después del bloque catch correspondiente si la excepción es manejada
  - Después del bloque try si no hay un bloque catch que maneje la excepción y se propaga

```
FileWriter file = new FileWriter( … );
try {
    file.write( … );
} catch(IOException e) {
    exception handling
} catch(Exception e) {
    exception handling
} finally {
  file.close();
}
```

Ahora, el código que libera el recurso no está duplicado

Flujo de ejecución con finally

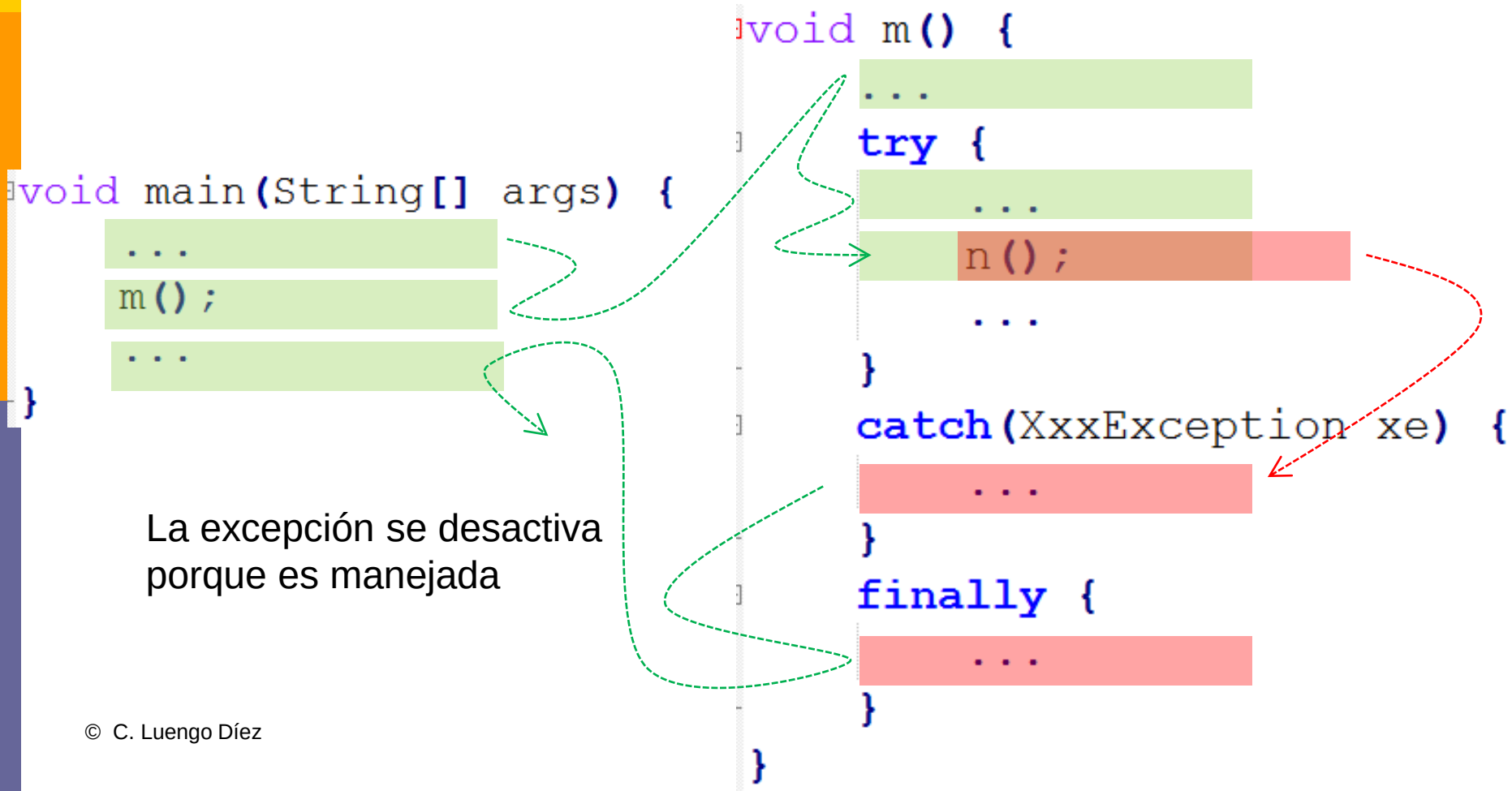
- Si no hay excepciones



El bloque finally se ejecuta siempre

Flujo de ejecución con finally

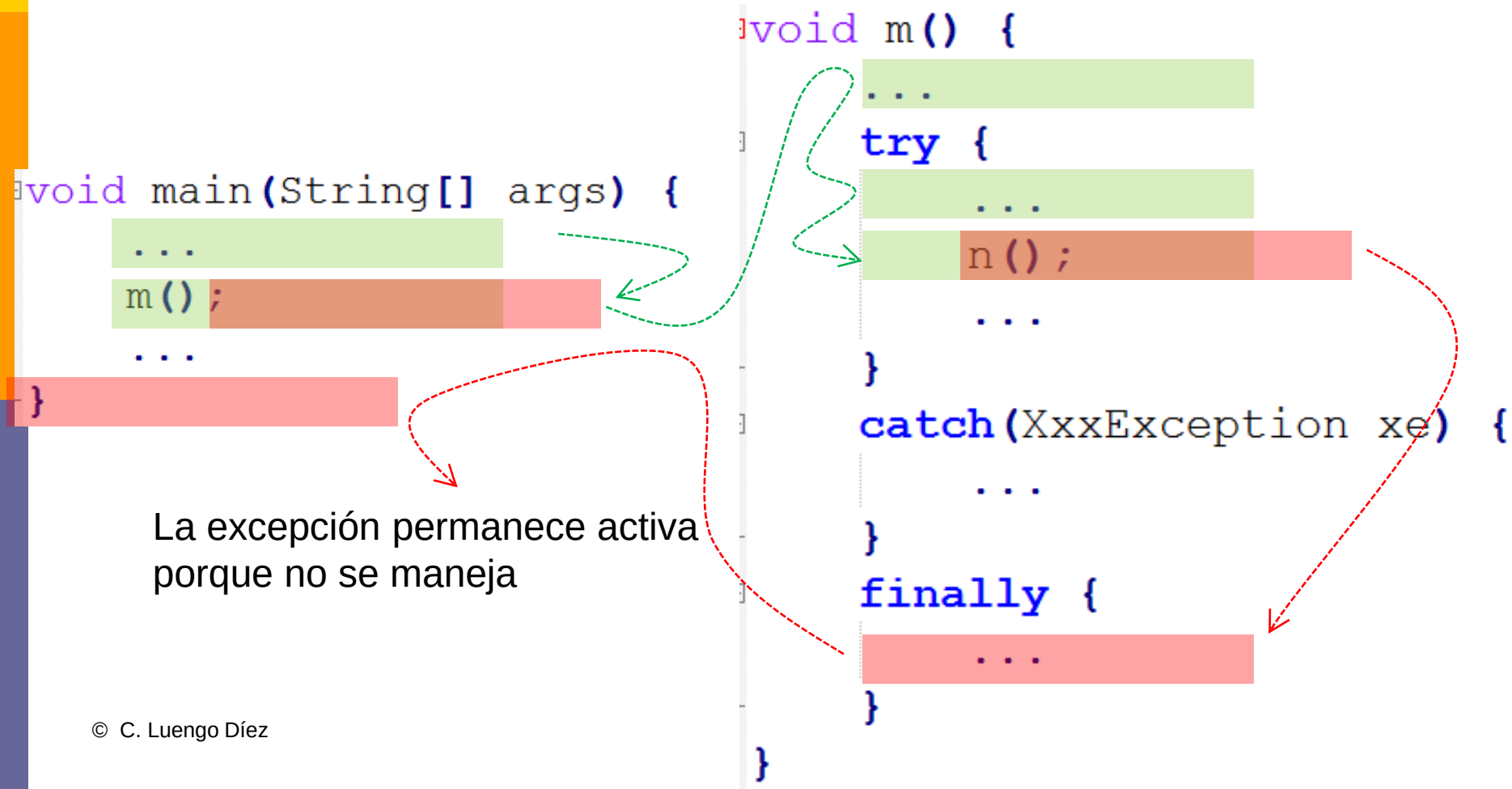
- Si hay una excepción manejada



La excepción se desactiva porque es manejada

Flujo de ejecución con finally

- Si hay una excepción pero no se maneja



¿Donde se pone el bloque catch para una excepción?

- Nuevamente es necesario pensar…
- Si m() invoca a n() que llama a p(), y p() puede lanzar una excepción, entonces…
  - Comienza con n() y piensa que puede hacer n() para manejar la excepción?…
  - … Si nada, entonces haz que la excepción llegue a m (), y piensa de nuevo que puede hacer m () para manejar la excepción?...
  - … Si nada…
- Pero no permitas que el programa “rompa” debido a una excepción no controlada, esto sería un bug

¿Donde se pone el bloque catch para una excepción?

- Normalmente, las excepciones no comprobadas se manejan en un bloque que:
  - Informa al usuario del error técnico.
  - Registra la traza y la información de la excepción.
  - Y el programa acaba.
- Normalmente, las excepciones comprobadas se manejan muy cerca del método que lanzó la excepción y
  - Podría informar al usuario sobre el error, si el usuario tiene que tomar una decisión, o
  - Podría ejecutar alguna acción alternativa

Precondiciones

- Otra **técnica de programación defensiva**

- Una **precondición** es una condición (predicado) que siempre debe ser verdadera antes de la ejecución de alguna sección de código.

- En el paradigma Orientado a Objetos, se aplica a la condición que debe cumplirse antes de la ejecución de un método **público.**

Precondiciones en Java

- Hay dos escenarios:
  1. Los valores de los argumentos no son apropiados (p.e., un índice fuera de los límites)
  2. El método invocado no puede ser ejecutado en el estado actual del objeto (`this`) (p.e., extraer un elemento de una cola vacía)

- Java ofrece dos excepciones no comprobadas para esas dos condiciones
  1. **`IllegalArgumentException`** → Los valores de los argumentos no son apropiados.
  2. **`IllegalStateException`** → El método invocado no puede ser ejecutado en el estado actual del objeto (`this`)

IllegalArgumentException

```
public class ArrayList {

   public Object get(int index){
     if (index <0 || index>=this.size()){
       throw new IllegalArgumentException(
            "Índice fuera de los límites");
     }

    return elements[index];
   }
   ...
}
```

IllegalStateException

- El método dequeue de la clase Queue...

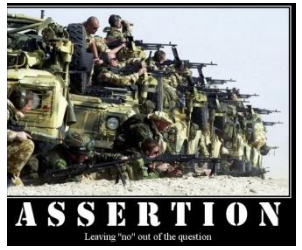
```
public Object dequeue() {
  if (isEmpty())
    throw new IllegalStateException(
              "No se puede extraer un elemento
  + "de una cola vacía");
  // ...
}
```

IllegalStateException

- El método dequeue de la clase Queue podría ser implementado de dos formas (poll() y remove())

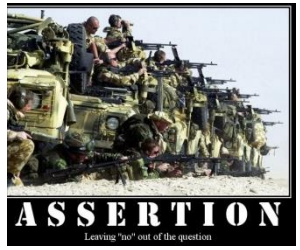
```
public Object dequeue() {          (poll)
  if (isEmpty()) return null;
  // ...
}

public Object dequeue() {          (remove)
  if (isEmpty()) {
    throw new IllegalStateException(
              " No se puede extraer un elemento
  + "de una cola vacía ");
  // ...
}
```

Aserciones



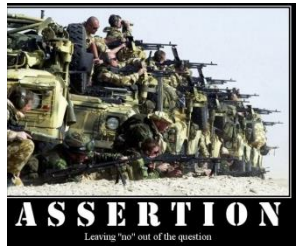
- Las aserciones son condiciones (predicados) que deben ser verdaderas para la correcta ejecución de un programa.
  - Si una condición no se cumple, el programa termina de forma anormal.
- Puesto que las aserciones pueden ocasionar la terminación del programa,
  - No deben utilizarse para detectar excepciones en tiempo de ejecución (errores).
  - No son reutilizables, ni extensibles.

Usando aserciones



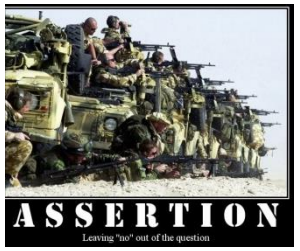
- **¿Cuando debemos usar las aserciones?**
  - Para comprobar las condiciones que siempre deben ser true.
  - Si no son ciertas, significa que **hay un error de programación.**
    - Están dirigidas a detectar errores de programación.
  - Sólo se utilizan en el ciclo de desarrollo
  - Se deben eliminar de forma transparente en la versión de lanzamiento (despliegue).
    - No deben suponer un coste para el rendimiento en tiempo de ejecución en el despliegue.

Aserciones en Java



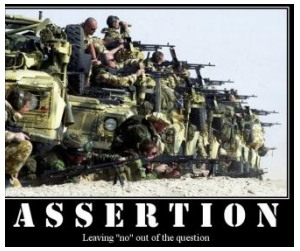
- En Java 1.4+ la sentancia `assert` fué incluida
  - `assert` *`condition`*
  - `assert` *`condition`* `:` *`message`*
- ***`Condition`*** es una expresión booleana:
  - Debe ser true
  - En caso contrario, el programa termina de forma anormal
- ***`Message`*** (opcional) es un objeto:
  - Cuando la aserción (*`condition`*) no se cumple, la representación del objeto mediante la llamada a toString, se muestra en la salida de error estándar

Usando aserciones en Java




```
public class Stack {
  private LinkedList list = new LinkedList();

  public void push(Object element) {
    if (element == null)
       throw new IllegalArgumentException(
                "No se pueden apilar elementos null");

    assert this.size() == list.size() : "Estado de los objetos incoherente";

    list.addFirst(element);
    assert !this.isEmpty() : "Error en push, la pila está vacía";
  } }
```

Habilitar aserciones en Java

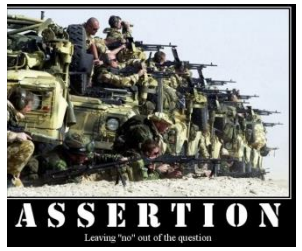


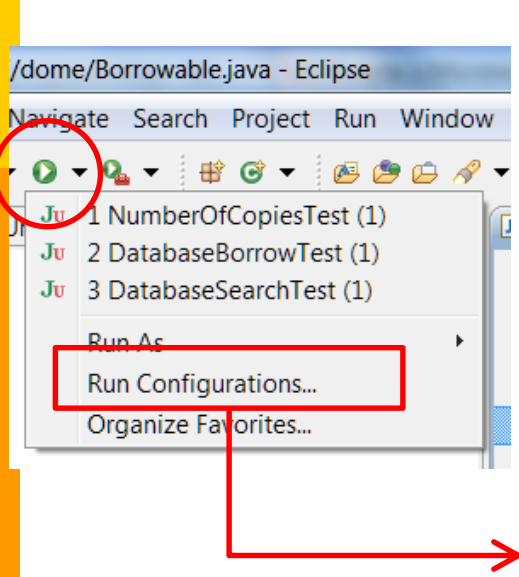
- En Java, las aserciones están deshabilitadas por defecto! ☹
- Para poder habilitarlas, debemos indicar la opción `-ea` (*enable assertions*) a la máquina virtual:

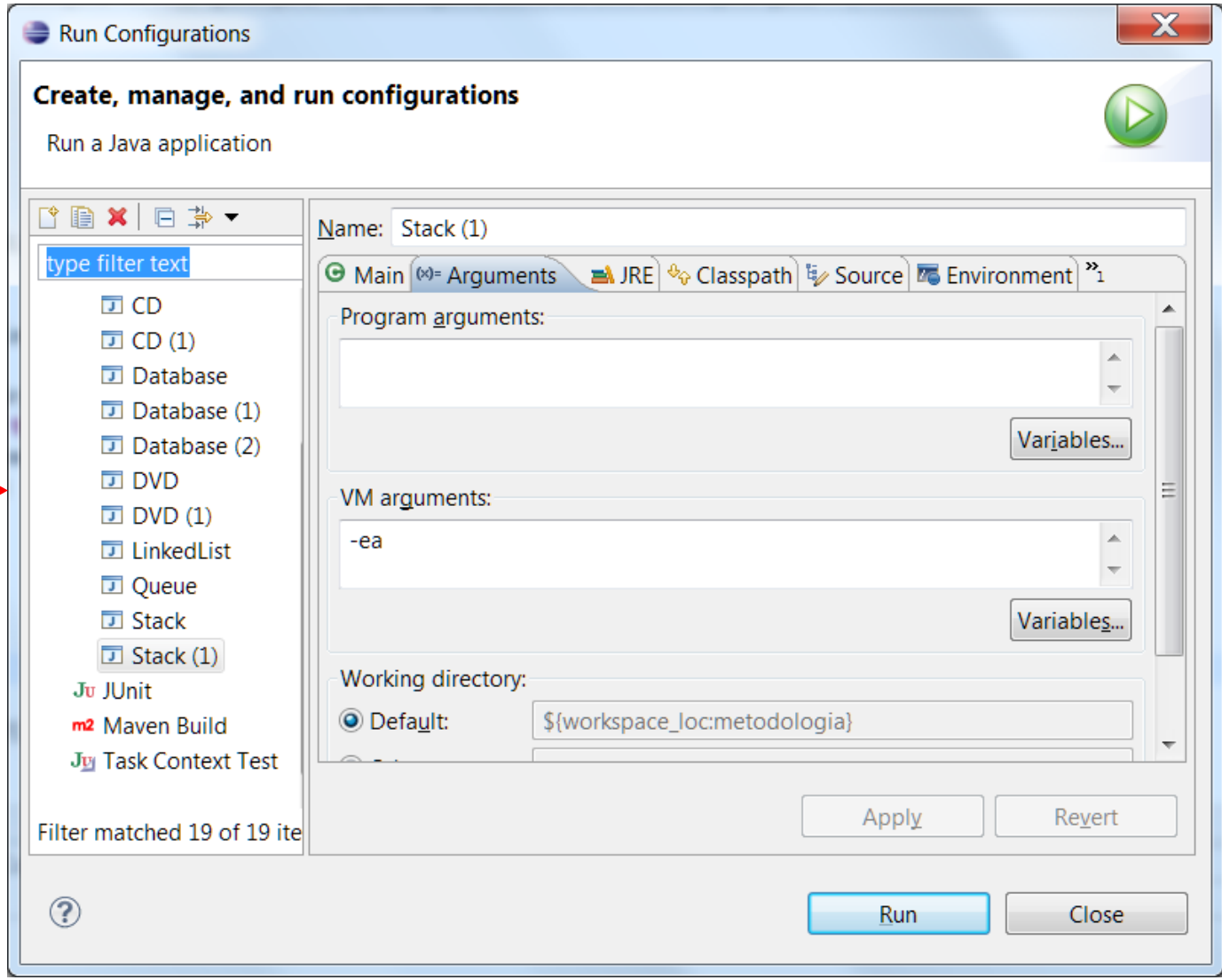
```
java -ea MyClass applicationArguments
```

- La opción `-da` (*disable assertions*) deshabilita las aserciones (y están por defecto)

Habilitar aserciones en Java







Conclusiones

- El manejo de excepciones es un mecanismo robusto, reusable y extensible para gestionar los errores de ejecución.
- Cuando se detecta un error se lanza una excepción con throw
- Donde se pueda manejar la excepción
  - Múltiples bloques catch son posibles
  - Usa polimorfismo para la generalización
- Cuando se manejan recursos hay que utilizar la claúsula finally
- Hay que pensar dónde se maneja cada exception

Conclusiones

- Las excepciones no comprobadas no requieren comprobaciones del compilador, mientras que las excepciones comprobadas sí.
- Se pueden crear nuevos tipos de excepciones
  - Comprobadas por problemas relacionados con el dominio
  - No comprobadas para la programación o errores del sistema
- Las precondiciones se deben comprobar siempre
  - En Java, se usan las excepciones
  - `IllegalArgumentException` y `IllegalStateExceptions`
- Las aserciones se usan para detectar errores de programación y se deshabilitan en despliegue.