Tema 6 – Corrección y Robustez
Excepciones en Java

Programación Orientada a Objetos

Curso 2019/2020

DIS
Departamento de
Informática y Sistemas

UNIVERSIDAD DE
MURCIA

Contenido

- Excepciones.
- Tratamiento de excepciones: try-catch.
- Jerarquías de excepciones y compatibilidad.
- Lanzamiento de excepciones.
- Excepciones y redefinición.
- Excepciones no tratadas.
- Errores de programación.
- Excepciones runtime.
- Diseño por contrato.

Corrección vs Robustez

- **Robustez**:
  - Es la capacidad de los productos software de reaccionar adecuadamente ante **situaciones excepcionales**.
  - Estas situaciones se notifican con excepciones que deben ser tratadas con el fin de recuperar los errores.
- **Corrección**:
  - Es la capacidad del software de realizar con exactitud su tarea (**cumplir su especificación**).
  - El incumplimiento de los **requisitos** de uso de las operaciones implica la finalización de la ejecución.

Motivación

- En el mundo real se producen fallos:
  - Sacamos el pendrive sin “extraerlo de forma segura”.
  - El vecino apaga la wifi …
  - Las obras del AVE rompen el cable de la fibra óptica.

- Una aplicación es **robusta** si es capaz de reaccionar adecuadamente ante **situaciones excepcionales**.

Caso de estudio

- **Caso de estudio**:
  - Queremos guardar en disco una página web.

- Conocemos los tipos de datos que nos ofrecen la funcionalidad requerida:
  - **`java.net.URL`**: representa el localizador de un recurso en internet, como por ejemplo, una página web.
  - Ofrece un método para obtener el flujo de lectura del recurso (**`openStream`**).
  - **`java.util.Scanner`**: permite leer fuentes de datos como un fichero o un recurso en internet.
  - **`java.util.PrintWriter`**: nos permite escribir un fichero en disco.

Caso de estudio

- El código que implementa la funcionalidad que buscamos es el siguiente:

```
URL url = new URL("http://dis.um.es/docencia/poo");

InputStream inputStream = url.openStream();
Scanner scanner = new Scanner(inputStream);
PrintWriter writer = new PrintWriter("poo.html");

while (scanner.hasNextLine()) {
    String linea = scanner.nextLine();
    writer.println(linea);
}

// Los flujos de lectura/escritura deben cerrarse
scanner.close();
writer.close();
```

Excepciones

- El fragmento de código anterior no compila debido a que **algunas operaciones pueden producir errores**:
  - Constructor de **`URL`**: el localizador del recurso no está bien formado.
  - Método **`openStream`**: no es posible localizar o leer el recurso especificado en internet.
  - Constructor **`PrintWriter`**: el nombre del fichero especificado no es correcto en el sistema de ficheros (por ejemplo, no se puede escribir).

- En Java, una operación indica que puede producir un error declarando que lanza una **excepción**.

Excepciones

- Por ejemplo, el método **`openStream`** se declara del siguiente modo:

```
public InputStream openStream() throws IOException {
```

- El método está indicando que *es posible* que falle por motivo de un error de entrada/salida (`IOException`)

- `IOException` es una **clase** que representa el error. Los objetos de esta clase se utilizan para notificar el error.

Tratamiento de excepciones

- El lenguaje Java exige al programador que no ignore los errores que puedan suceder.
- Así pues, el programador del código anterior debe dar tratamiento a las excepciones que puedan producirse.
- Java ofrece la construcción **try-catch** para tratar las excepciones que puedan producirse en el código.

Construcción try-catch

- La construcción **`try-catch`** está formada por:
  - **Bloque `try`**: bloque que encierra código que puede lanzar excepciones.
  - **Bloques `catch`** o **manejadores**: uno o varios bloques encargados de dar tratamiento a las excepciones.
  - **Bloque `finally`**: bloque que siempre se ejecuta, se produzca o no excepción (bloque opcional).

Construcción try-catch

- **Ejemplo** de tratamiento de error:

```
InputStream inputStream = null;

try {
   inputStream = url.openStream();
}
catch (IOException e) {

   System.out.println("Error al leer el recurso");
   return; // error crítico: finaliza la operación
}

Scanner scanner = new Scanner(inputStream);
// ...
```

Construcción try-catch

- La construcción **`try-catch`** se interpreta del siguiente modo:
  - *Intenta* ejecutar el fragmento de código envuelto por el bloque **`try`**.
  - Si la ejecución no produce errores, el código continúa después del bloque **`catch`**. En el ejemplo, sigue por la construcción del `Scanner`.
  - Si se produce algún error: se revisan en orden los manejadores (`catch`) y se ejecuta el **primero** que sea **compatible** con el error.

Bloque `finally`

- En el tratamiento de excepciones el bloque **`finally`** es opcional. Se reserva para aquellas acciones que deben realizarse, sucedan o no excepciones.

```
Scanner scanner = null;
PrintWriter writer = null;
try {
   // ...
} catch (IOException e) {
   // ...
}
finally {
   if (scanner != null)
      scanner.close();
   if (writer != null)
      writer.close();
}
```

Los flujos de E/S siempre hay que cerrarlos

Caso de estudio

- Generalizamos el fragmento de código anterior y definimos un **método** que reciba como parámetros la cadena con la URL del recurso y el nombre del fichero donde almacenarlo.

```
public static void guardarWeb(String urlRecurso,
                               String nombreFichero) {
    URL url = new URL(urlRecurso);
    InputStream inputStream = url.openStream();
    Scanner scanner = new Scanner(inputStream);
    PrintWriter writer = new PrintWriter(nombreFichero);

    while (scanner.hasNextLine()) {
        String linea = scanner.nextLine();
        writer.println(linea);
    }
    scanner.close();
    writer.close();
}
```

Caso de estudio

- En el método anterior, las instrucciones que aparecen subrayadas no compilan debido a que producen excepciones que no son tratadas.

- El constructor de `URL` declara lanzar la excepción **`URLMalformedException`** (clase descendiente de `IOException`).
- El método `openStream()` puede producir la excepción **`IOException`**.
- El constructor de `PrintWriter` declara la excepción **`FileNotFoundException`** (clase descendiente de `IOException`).

Dejar escapar una excepción

- En muchas ocasiones es difícil saber qué tratamiento dar a una excepción.
- Para el compilador de Java, un tratamiento “aceptable” de una excepción es dejarla escapar para que “otros” la traten.
- Dejar escapar una excepción equivale a declarar en la operación en la que está el código que lo produce que puede lanzar el error:

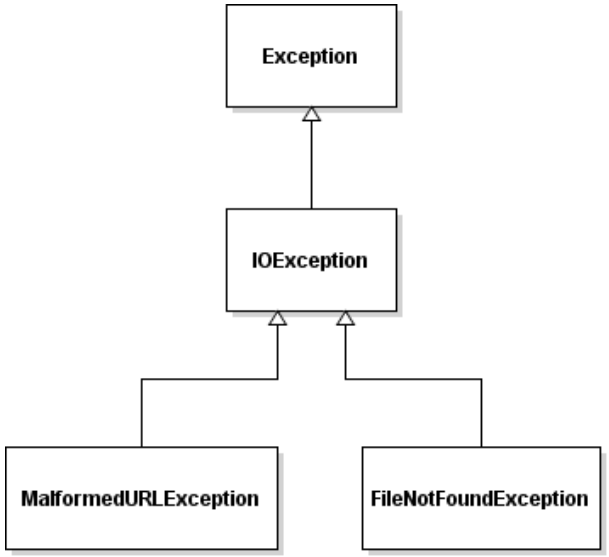
```
public static void guardarWeb(String urlRecurso,
    String nombreFichero) throws MalformedURLException {

    // Dejamos pasar la excepción MalformedURLException
    URL url = new URL(urlRecurso);

    // ...
```

Jerarquías de excepciones

- Las excepciones se organizan en **jerarquías de herencia**.
- La clase **`Exception`** es la raíz de todas las excepciones en Java.
- La clase `IOException` es la clase padre de `MalformedURLException` y `FileNotFoundException`.



Compatibilidad de excepciones

- La organización de las excepciones en jerarquías permite aprovechar el sistema de **compatibilidad de tipos** que ofrece la Programación Orientada a Objetos, tanto para dar tratamiento como para declarar las excepciones que escapan.
- **Ejemplo**: un único *catch* para todas las excepciones

```
public static void guardarWeb(String urlRecurso,
                              String nombreFichero) {
   try {
      URL url = new URL(urlRecurso);
      // ...
      writer.close();
   } catch (IOException e) {
      System.out.println("Error de Entrada/Salida");
      // ...
   }
}
```

Compatibilidad de excepciones

- En el ejemplo anterior, el bloque **`catch(IOException e)`** da tratamiento a los tres posibles errores:
  - `MalformedURLException` es compatible con el tipo del manejeador (`IOException`)
  - `IOException` corresponde con el tipo del manejador.
  - `FileNotFoundException` es compatible con el tipo del manejador.

- La **variable** **`IOException e`** es una referencia al objeto que contiene la notificación de error. Gracias al polimorfismo permite referenciar a cualquier objeto cuyo tipo sea descendiente de `IOException`.

Compatibilidad de excepciones

- La compatibilidad de tipos también es aplicable a la declaración de excepciones:

```
public static void guardarWeb(String urlRecurso,
        String nombreFichero)  throws IOException {

    URL url = new URL(urlRecurso);
    InputStream inputStream = url.openStream();
    Scanner scanner = new Scanner(inputStream);
    PrintWriter writer = new PrintWriter(nombreFichero);

    while (scanner.hasNextLine()) {
        String linea = scanner.nextLine();
        writer.println(linea);
    }
    scanner.close();
    writer.close();
}
```

Compatibilidad de excepciones

- El método anterior declara que puede lanzar excepciones compatibles con `IOException`.

- Todos los errores que puede producir el código del método quedan abarcados por la excepción `IOException`:
  - Constructor de `URL`: `MalformedURLException`
  - Método `openStream()`: `IOException`
  - Constructor de `PrintWriter`: `FileNotFoundException`

Clase `Exception`

- La clase **`Exception`** es la raíz de las excepciones.
- Gracias a la compatibilidad de tipos:
  - Declarar un manejador `catch(Exception e)` daría tratamiento a cualquier excepción.
  - Declarar en un método `throws Exception` indica que puede producir cualquier excepción.
- Las excepciones heredan de `Exception`:
  - La posibilidad de establecer un mensaje de error en el constructor y de recuperarlo durante el tratamiento a través del método `getMessage()`
  - Imprimir la pila de llamadas del punto en el que se ha producido el error: `printStackTrace()`

Lanzamiento de excepciones

- Una excepción se lanza construyendo un objeto de la clase que representa la excepción y notificándolo con **`throw`**.
- **Ejemplo**:
  - El método `checkError()` de `PrintWriter` comprueba si se produce un error en las operaciones de escritura.
  - En caso afirmativo, lanzamos una excepción `IOException`.

```
// ...
while (scanner.hasNextLine()) {
   String linea = scanner.nextLine();
   writer.println(linea);
   if (writer.checkError())
     throw new IOException("Error de escritura");
}
// ...
```

Lanzamiento de excepciones

- El lanzamiento de una excepción implica la finalización de la ejecución de la operación en ese punto del código.
- El compilador de Java obliga a que las operaciones declaren las excepciones que lanzan:

```
public static void guardarWeb(String urlRecurso,
                              String nombreFichero)  throws IOException {
   // ...
   while (scanner.hasNextLine()) {
       String linea = scanner.nextLine();
       writer.println(linea);
       if (writer.checkError())
          throw new IOException("Error de escritura");
   }
   // ...
```

Lanzamiento de excepciones

- En el ejemplo anterior se ha reutilizado la excepción `IOException` incluida en la librería de Java.
- Sin embargo, se podría haber creado una **nueva excepción**.
- Las clases que representan excepciones heredan de `Exception` y pueden organizarse en jerarquías.

```
public class EscrituraException extends Exception {

    public EscrituraException() {
        super();
    }
    public EscrituraException(String msg) {
        super(msg);
    }
}
```

Excepciones y redefinición

- Al redefinir un método heredado **podemos modificar la declaración de las excepciones** (**`throws`**).
- Podemos mantener o **reducir** las excepciones que lanza.
- Reducir las excepciones significa no declarar algunas que declara el método en la clase padre.
- También se considera reducir excepciones indicar una excepción más específica:
  - En la clase padre el método lanza `IOException` y la redefinición del método en la clase hija indica `FileNotFoundException` que es un subtipo.

Control de excepciones - Resumen

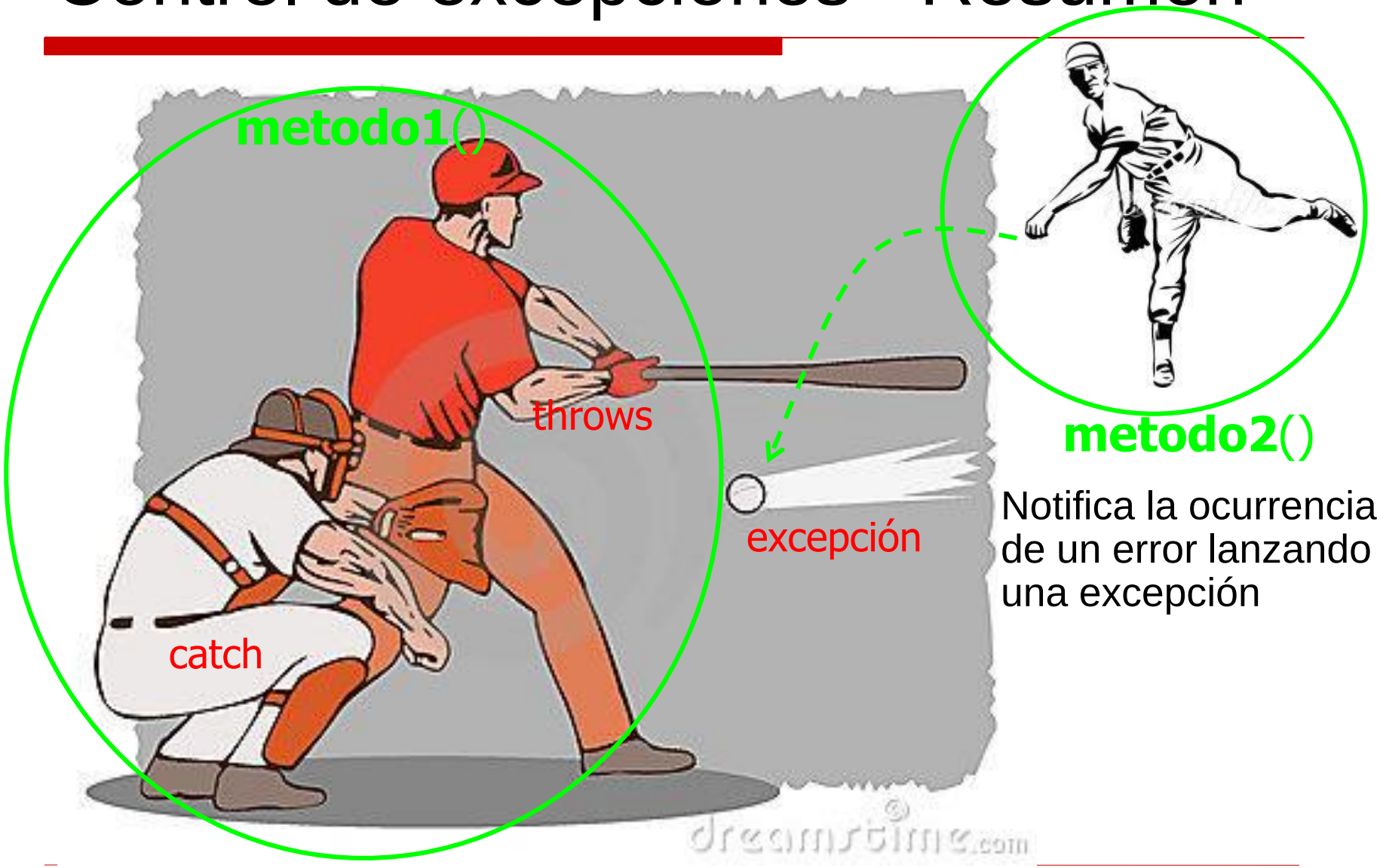
- El compilador realiza un **control del tratamiento de las excepciones**.

- Si un método utiliza una operación que puede lanzar una excepción, el compilador permite sólo dos **opciones**:

  - Dar tratamiento al error en un bloque *try-catch*.

  - Declarar que el método puede producir ese error (*throws*).

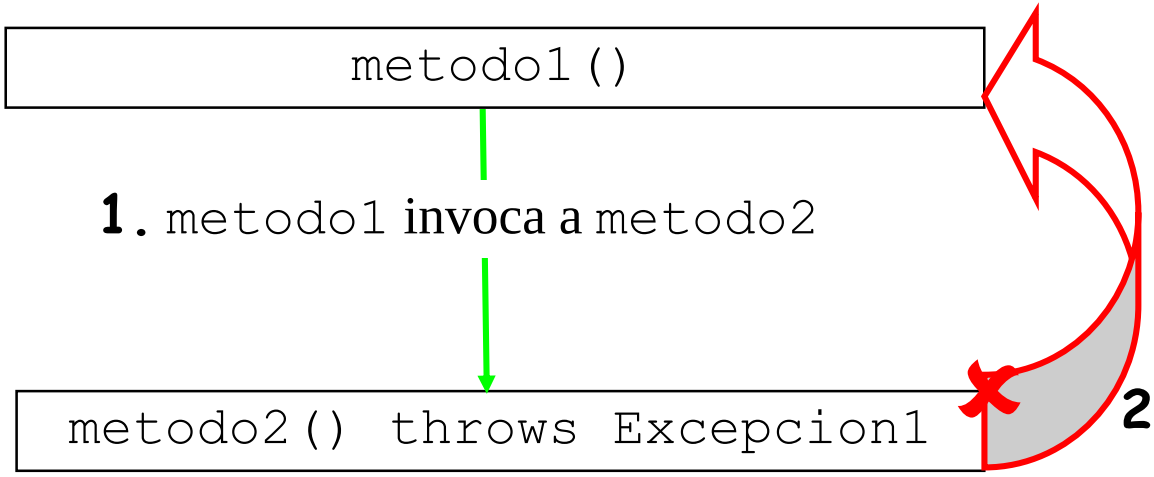
Control de excepciones - Resumen



Notifica la ocurrencia de un error lanzando una excepción

Control de excepciones - Resumen

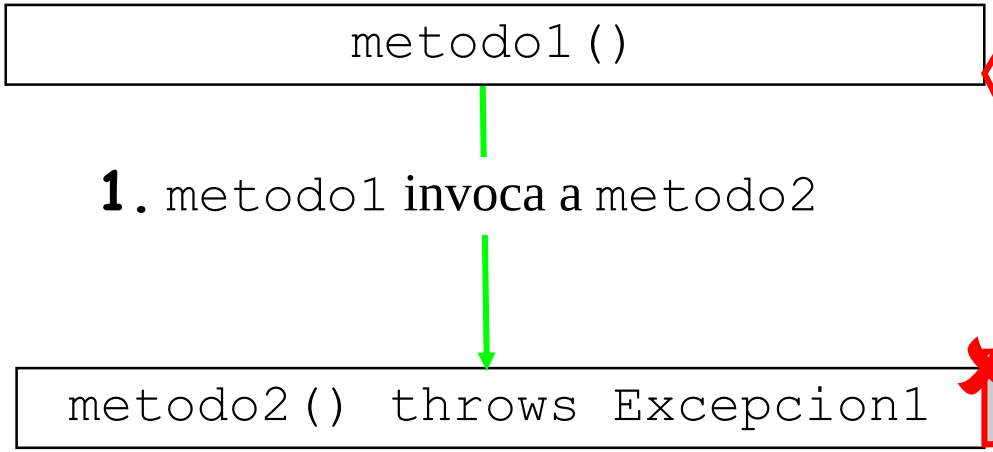
¿Qué hace `metodo1` cuando le llega una excepción?



Control de excepciones - Resumen

a) `metodo1` define un manejador para tratar el error

```
void metodo1{
   try{
      metodo2();
   catch(Excepcion1 e){
       //manejador de la situación de error
   }
}
```



3. `metodo1` maneja el fallo y continúa la ejecución

2. `metodo2` encuentra un fallo y lo notifica a `metodo1` lanzando una excepción

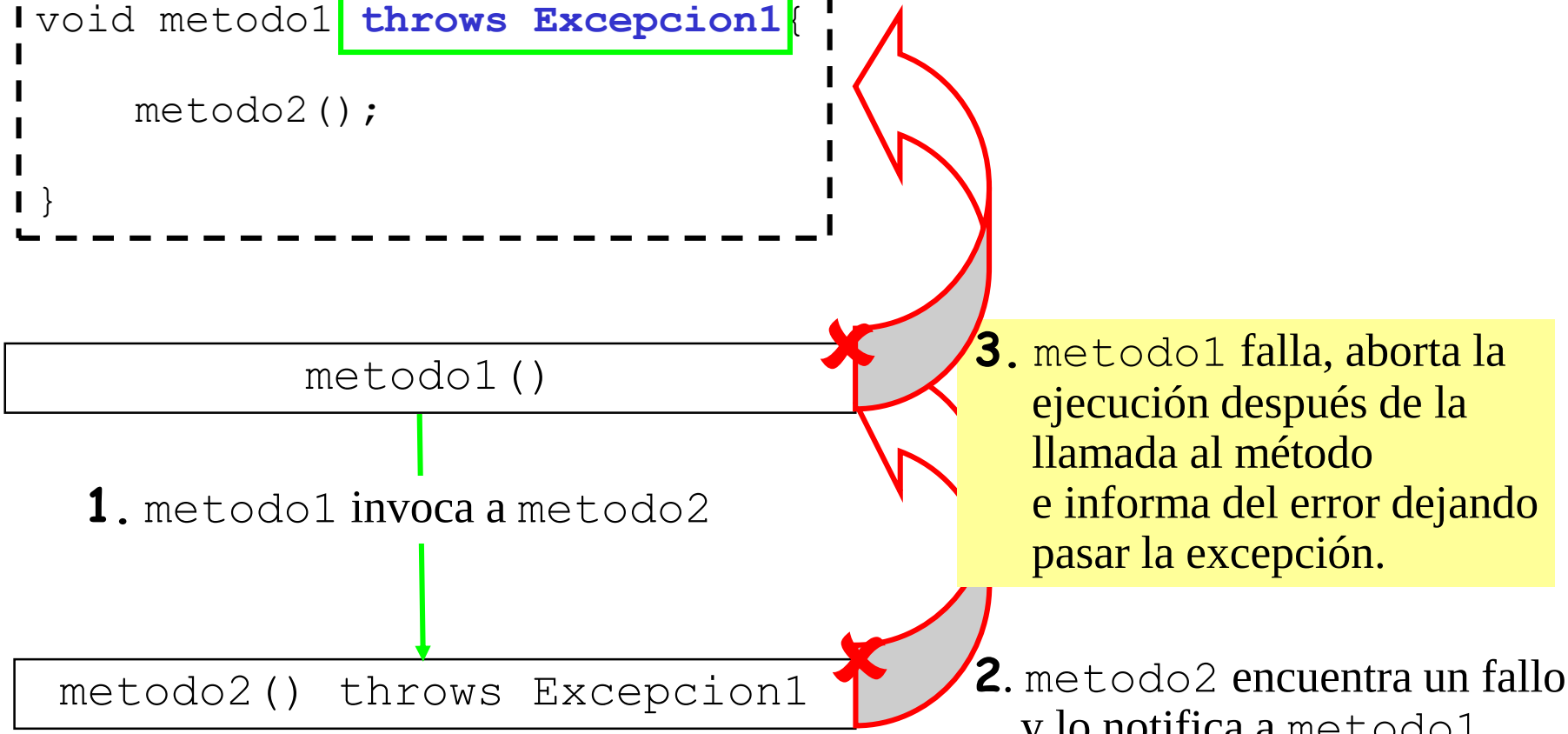
Control de excepciones - Resumen

b) `metodo1` no maneja el error, lo deja pasar

```
void metodo1 throws Excepcion1{

    metodo2();

}
```



Excepciones no tratadas

- Una **excepción no tratada** aborta la ejecución de una operación en el punto en el que se produce.
- Asimismo, el **lanzamiento de una excepción** también finaliza la ejecución de la operación en el punto en el que se lanza.
- Es posible que una excepción pueda propagarse a través de varias operaciones.
- **Si una excepción escapa al método `main()`** de la aplicación, **el programa finaliza con un error**.

Excepciones no tratadas

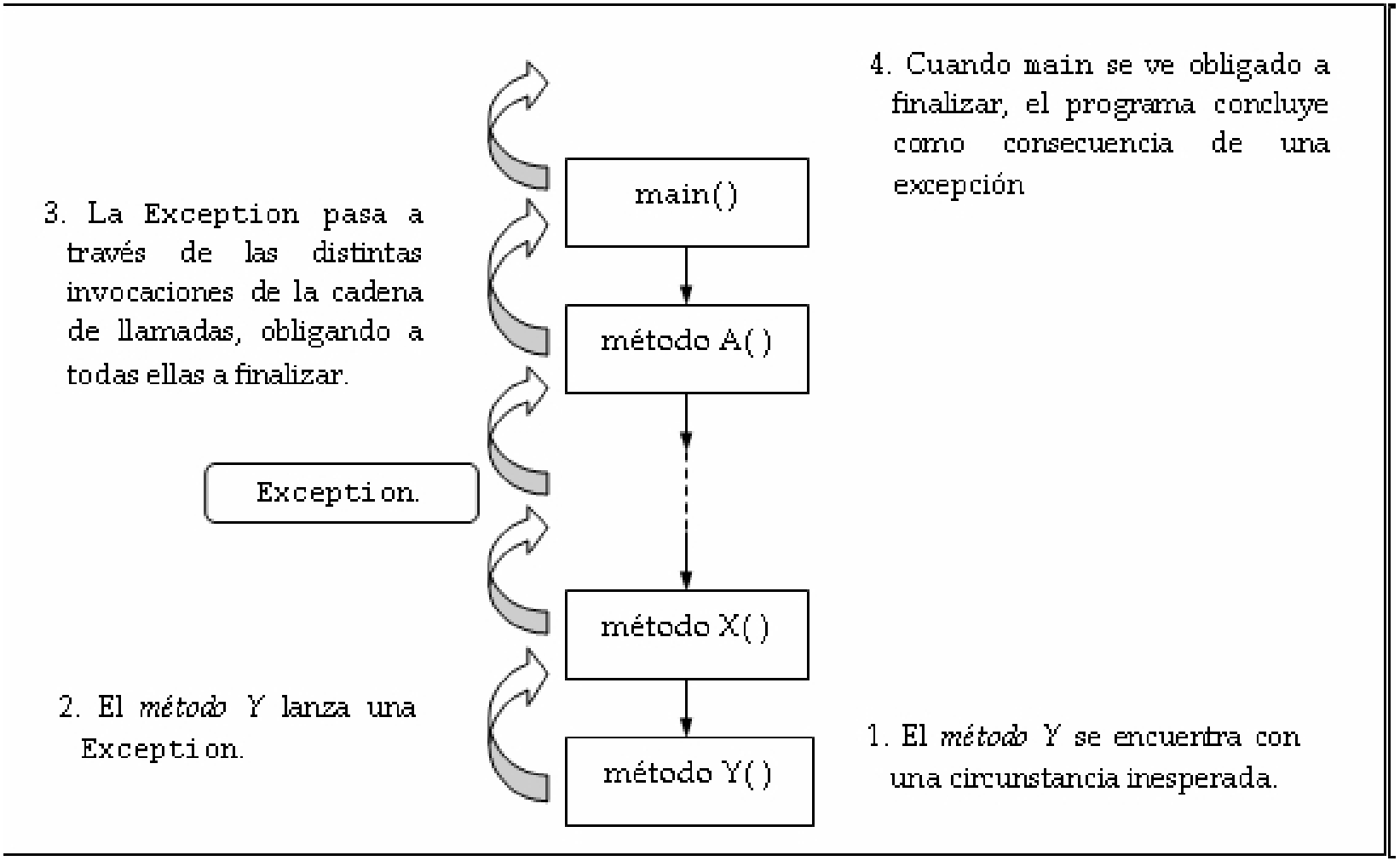


Imagen tomada de: **http:/binarycodefree.blogspot.com/2010/02/control-de-excepciones-net.html/**

Contexto

- En las diapositivas anteriores se ha presentado el mecanismo ofrecido por el lenguaje Java para notificar situaciones de error con el fin de ser recuperadas.
- El mecanismo de excepciones permite programar **aplicaciones robustas**, esto es, que responden adecuadamente a situaciones de error.

- En las siguientes diapositivas se utiliza el mecanismo de excepciones con otro fin.
- Las excepciones se utilizan para controlar el **uso correcto de las operaciones**, esto es, notifican errores de programación. Su objetivo es detener la ejecución de la aplicación.
- Esta forma de utilizar las excepciones da lugar a una nueva categoría de excepciones denominada *runtime*.

Errores de programación

- En los lenguajes tipados como Java o C++, la obligación a declarar el tipo de las variables permite al compilador identificar numerosos errores de programación.
- **Ejemplo**: constructor de la clase `Burbuja`

```
public Burbuja(Circulo region, int velocidadMax) {
    this.region = new Circulo(region.getCentro(), region.getRadio());
    this.velocidadMax = velocidadMax;
    this.velocidadActual = 0;
    this.explotada = false;
}
```

- El compilador no aceptaría utilizar el constructor estableciendo como segundo parámetro una cadena:

```
Burbuja burbuja = new Burbuja(circulo1, "10"); // Error
```

Errores de programación

- Sin embargo, no debemos confiar en que un código que compila va a funcionar correctamente:

```
Circulo circulo1 = null; // luego lo creo ...

// se me olvidó crear el círculo
Burbuja burbuja = new Burbuja(circulo1, 10);
```

- La máquina virtual lanza un error de referencia nula.

```
public Burbuja(Circulo region, int velocidadMax) {
    this.region = new Circulo(region.getCentro(), region.getRadio());
    // ...
}
```

Errores de programación

- Los lenguajes de programación interpretados (no compilados) responden a los fallos de programación con notificaciones que finalizan la ejecución.
- La máquina virtual controla algunos errores de programación que escapan al compilador, como por ejemplo, el uso de una referencia nula.
- Sin embargo, no siempre es así:

```
Circulo circulo1 = new Circulo(new Punto(50, 50), 50);

Burbuja basica = new Burbuja(circulo1, -100);
```

- El videojuego no falla, pero no se comporta correctamente. La burbuja no asciende nunca.

Errores de programación

- **Problema**: Un programador debe ser responsable de controlar los errores que escapan al compilador y a la máquina virtual.
- El modo de actuar debería ser detener el funcionamiento de la aplicación.

- **Solución**: utilicemos excepciones para notificar fallos de programación.
  - Las excepciones que se notifican como fallos de programación se dejan pasar hasta que finaliza el programa.

Errores de programación

- **Ejemplo**:
  - En el caso de estudio de la burbuja, creamos la excepción `EnteroNegativoException`.
  - Notificamos la excepción si la velocidad máxima no es positiva.

```
public Burbuja(Circulo region, int velocidadMax)
                              throws EnteroNegativoException {

  if (velocidadMax <= 0)
    throw new EnteroNegativoException("velocidadMax debe ser positivo");

// ...
```

- Los métodos que utilicen el constructor deben declarar y dejar pasar la excepción.

Errores de programación

- **Problema**:
  - Es un inconveniente utilizar excepciones para notificar fallos de programación debido a que es necesario declarar que las excepciones se dejan pasar.
- **Solución**:
  - En Java se decidió definir una categoría de excepciones “descafeinadas” con el fin de ser utilizadas para notificar errores de programación.
  - Estas excepciones se denominan “runtime” porque su objetivo es notificar errores de programación en tiempo de ejecución.
  - Además, estos errores no deben recuperarse, no es una situación excepcional (fallo de red, fallo de disco, etc.)

Excepciones runtime

- Estas excepciones son **subtipos de `RuntimeException`**.
- En general, se definen y utilizan como el resto de excepciones. Sin embargo, ofrecen algunas **ventajas**:
  - Si una operación lanza una excepción *runtime*, **no hay obligación** a declararla en la cabecera (`throws`)
  - Si una operación utiliza otra que lanza una excepción *runtime*, **no hay obligación** a tratarla (try-catch) o declarar que escapa.
  - No siguen las reglas en la redefinición de métodos (podemos declarar las que queramos).

➔ En general, **estas excepciones no se tratan**.

Por este motivo, también se conocen como “no comprobadas”.

Excepciones runtime

- Por tanto, el **uso incorrecto del código** (errores de programación) se notifica con excepciones *runtime.*

- La librería de Java proporciona varias (heredan de la clase **`RuntimeException`**):
  - **`NullPointerException`**: excepción de uso de una referencia nula.
  - **`IllegalArgumentException`**: se está estableciendo un argumento incorrecto a un método.
  - **`IllegalStateException`**: la aplicación de un método no es permitida por el estado del objeto.
  - …

Excepciones runtime

- El lenguaje Java utiliza excepciones *runtime* cuando se utilizan incorrectamente las construcciones del lenguaje o las librerías:
  - Error de **casting** notificado con la excepción `ClassCastException`
  - **Acceso incorrecto a un array** notificado con `ArrayIndexOutOfBoundsException`
  - **Acceso incorrecto a una lista** lanza la excepción `NoSuchElementException`
  - **Aplicación de un método sobre una referencia nula** provoca la excepción `NullPointerException`

Corrección vs Robustez

- **Corrección**:
  - Es la capacidad del software de realizar con exactitud su tarea (**cumplir su especificación**).
  - El incumplimiento de los **requisitos** de uso de las operaciones implica la finalización de la ejecución.

- **Robustez**:
  - Es la capacidad de los productos software de reaccionar adecuadamente ante **situaciones excepcionales**.
  - Estas situaciones se notifican con excepciones que deben ser tratadas con el fin de recuperar los errores.

Diseño por contrato

- Las operaciones tienen **requisitos** de uso, también denominados **precondiciones**.
- Se considera un **error de programación** no cumplir con los requisitos de una operación.
- El incumpliendo de los requisitos implica que la aplicación no debe continuar (**corrección**).
- El programador que hace uso de una operación es responsable de asegurar que cumple las precondiciones.

- A este modo de proceder ante los errores de programación se denomina **Diseño por contrato**.

Diseño por contrato

- Aunque podemos declarar nuestras propias excepciones *runtime*, habitualmente se utilizan las que ofrece la librería.
- **Ejemplo**: para **argumentos** incorrectos utilizamos **`IllegalArgumentException`**

```
public Burbuja(Circulo region, int velocidadMax) {

   if (region == null)
     throw new IllegalArgumentException("region no debe ser nulo");

   if (velocidadMax <= 0)
     throw new IllegalArgumentException("velocidadMax debe ser positivo");

   // ...
```

- Observa que al ser una excepción *runtime* no se declara en la operación.

Diseño por contrato

- Además de los requisitos sobre los parámetros, también se considera precondición el **estado** en el que debe estar un objeto para ejecutar algunas operaciones.
- En este caso, la excepción utilizada para notificar el error es **`IllegalStateException`**.
- **Ejemplo**: el método `ascender()` de `Burbuja` no se debe utilizar si está explotada.

```
public void ascender() {

   if (this.explotada)
     throw new IllegalStateException("la burbuja ha sido explotada");

   // ...
```

Conclusiones

- En Java se utilizan **excepciones** para notificar **situaciones excepcionales de error** que deben ser recuperadas.
- También se utilizan excepciones para **notificar errores de programación**:
  - Las excepciones que se utilizan para este fin se denominan *runtime* y tienen como objetivo detener la ejecución.
  - Estas excepciones en general no se tratan ni hay obligación a declararlas. Por este motivo también se conocen como “no comprobadas”.
- Las excepciones ordinarias suelen denominarse “comprobadas” para distinguirlas de las *runtime*.