# Tema 4 – Corrección y Robustez
# Excepciones en Java

Programación Orientada a Objetos

Curso 2014/2015

DIS
Departamento de
Informática y Sistemas

UNIVERSIDAD DE
MURCIA

# Contenido

- **Parte I: Corrección del código**
  - Introducción
  - Excepciones *runtime* y Diseño por Contrato
  - Asertos
- **Parte II: Excepciones en Java**
  - Excepciones comprobadas
  - Caso de estudio
  - Tratamiento de excepciones
  - Control de excepciones
  - Excepciones y herencia
  - Excepciones vs. valor especial de retorno
- Excepciones comprobadas y no comprobadas
- Consejos de uso de excepciones

# Parte I: Corrección del código

- **Corrección**:
  - Es la capacidad de los productos software de realizar con exactitud su tarea (**cumplir su especificación**).

- **Robustez**:
  - Es la capacidad de los productos software de reaccionar adecuadamente ante **situaciones excepcionales**.

- La reutilización y extensibilidad no deben lograrse a expensas de la **fiabilidad** (corrección y robustez).

# Especificación

- Un código es **correcto** si cumple su especificación.
- La **especificación** (formal) de la semántica de una **rutina** está definida por:
  - **Precondiciones**: condiciones para que una rutina funcione correctamente.
  - **Postcondiciones**: describen el efecto de la rutina.

- El estado que deben cumplir los objetos de una clase se denomina **invariante**.
  - Restricciones que deben ser satisfechas por cada objeto tras la ejecución de los métodos y constructores.

# Corrección de código en Java

- Java proporciona dos herramientas para tratar con la corrección del código:
  - **Excepción**: mecanismo para notificar un error durante la ejecución.
  - **Aserto**: condición que debe cumplirse en el código para que sea correcto (depuración).

# Introducción a las excepciones

- Mecanismo proporcionado por el lenguaje de programación para **notificar y tratar errores en tiempo de ejecución**.
- La información del error, **excepción**, es un **objeto** que se propaga a todos los objetos afectados por el error.
- Las excepciones pueden tratarse con el propósito de dar una solución al error: **recuperación de errores**.

- **Nota**: las errores notificados por el uso incorrecto del código no suelen tratarse.

# Excepciones runtime

- El **uso incorrecto del código** se notifica con excepciones *runtime*, también denominadas "no comprobadas".

- 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.
  - …

# Ejemplos de uso

- El lenguaje Java utiliza excepciones no comprobadas 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`

# Diseño por contrato

- Esas operaciones tienen **precondiciones** de uso para que se ejecuten correctamente:
  - Sólo se debe aplicar un casting si el tipo de la conversión es compatible.
  - El acceso a un array va desde 0 hasta `length - 1`
  - Sólo se puede consultar el primer o último elemento si la lista no está vacía.
  - No se puede aplicar un método sobre una referencia nula.
- Es responsabilidad de quien invoca esas operaciones asegurar que cumple las precondiciones.

# Diseño por contrato

- Asegurar las precondiciones no significa comprobarlas antes de ejecutar la operación.
- **El contexto del código puede garantizar que se cumplen las precondiciones**:

```
public static void main(String[] args) {

        LinkedList<String> lista = new LinkedList<String>();

        if (args.length > 0) // ¿Tiene argumentos?
                lista.add(args[0]);
        else
                lista.add("");

        // La colección no es nula y tiene al menos un elemento
        System.out.println("Primer argumento: " + lista.getFirst()
}
```

# Diseño por contrato

- Las operaciones que establecen precondiciones deben permitir poder comprobarlas:
  - Para comprobar que un casting sea correcto utilizamos el operador `instanceof`
  - Para controlar si se accede correctamente a un array podemos consultar su tamaño con `length`
  - Podemos saber si una referencia es nula si la comparamos con `null`
  - Podemos comprobar si una lista es vacía utilizando el método `isEmpty()` o `size()`

# Control de precondiciones

- Al definir un método o constructor debemos documentar las precondiciones y comprobarlas al comienzo del código.

- Ejemplo: clase **`Cuenta`**, método **`ingreso`**`()`:
  - No se puede realizar un ingreso en una cuenta si no está operativa y la cantidad es negativa.

- Controlar el cumplimiento de las precondiciones permite la **detección de errores de programación** en el punto en el que se producen, facilitando así la depuración.

# Control de precondiciones

- Habitualmente se comprueban dos **tipos de precondiciones**:

  - **Parámetros**: los parámetros de un método son los correctos.

    Ejemplo: cantidad positiva en método **`ingreso()`**.

  - **Estado**: un método no puede ser invocado en el estado del objeto.

    Ejemplo: el método **`ingreso()`** sólo puede ser ejecutado si la cuenta está operativa.

# Control de precondiciones

```
/**
 * Ingresa una cantidad en una cuenta operativa.
 * @param cantidad valor mayor que cero.
 */
public void ingreso (double cantidad) {

   if (cantidad <= 0)
      throw new IllegalArgumentException("Error cantidad");

   if (estado != EstadoCuenta.OPERATIVA)
      throw new IllegalStateException("Estado incorrecto");

   saldo = saldo + cantidad;

}
```

# Control de precondiciones

- El incumplimiento de una precondición es entendido como un **error de programación** y no como una situación anómala de la ejecución.

- Por tanto**, no es obligatorio dar tratamiento** a esas excepciones, ya que se supone que “no deberían” ocurrir.

- El control de precondiciones debe ser consistente con la **ligadura dinámica:** Diseño por Contrato en herencia.

# Diseño por Contrato y Herencia

- Cuando invocamos a un método `met` sobre una entidad polimórfica es posible que se ejecute una versión redefinida.
- `met` establece un contrato (precondiciones y postcondiciones) que es el que tiene la obligación de cumplir el cliente.
- La versión redefinida de `met` establece una especie de "subcontrato" en el que puede:
  - Mantener o **relajar** la **precondición**: lanzar menos excepciones runtime.

  → No debe añadir precondiciones que no tenga el método que se está redefiniendo.

# Excepciones no comprobadas

- En Java existen dos categorías de excepciones: **comprobadas** y **no comprobadas** (*runtime*).
- Las excepciones que se han utilizado para controlar la corrección del código pertenecen a la categoría de "no comprobadas": `IllegalArgumentException`, `IllegalStateException`, ...
- En la segunda parte del tema se presentan las excepciones comprobadas.
- Las excepciones comprobadas tienen restricciones más fuertes de uso, como se verá más adelante.

# Asertos

- Construcción proporcionada por el lenguaje que permite **comprobar si una condición se cumple en el código**.
- Declaración de un aserto:
  - `assert` *`expresión booleana;`*
  - `assert expresión booleana: "mensaje de error";`
- La **violación de un aserto** (evaluación *false*) provoca un error en la aplicación notificado con una excepción (**`AssertError`**).
- Se pueden incluir asertos **en cualquier punto del código** donde queramos asegurar que se cumple una condición → útil para **depuración**.

# Asertos – Ejemplo

- **Ejemplo**: aseguramos que el método `ingreso()` siempre incrementa al saldo.

```
public void ingreso (double cantidad) {

  double oldSaldo = saldo;

  saldo = saldo + cantidad;

  assert saldo > oldSaldo;
}
```

# Asertos

- Los asertos son útiles para la **depuración** el código.
- Por defecto, la máquina virtual de Java no comprueba asertos.
- Se activan con el parámetro `-ea` de la máquina virtual.

➔ No es recomendable utilizar asertos para controlar las precondiciones, ya que pueden desactivarse.

# Parte II: Excepciones en Java

- **Corrección**:
  - Es la capacidad de los productos software de realizar con exactitud su tarea (**cumplir su especificación**).

- **Robustez**:
  - Es la capacidad de los productos software de reaccionar adecuadamente ante **situaciones excepcionales**.

- La reutilización y extensibilidad no deben lograrse a expensas de la **fiabilidad** (corrección y robustez).

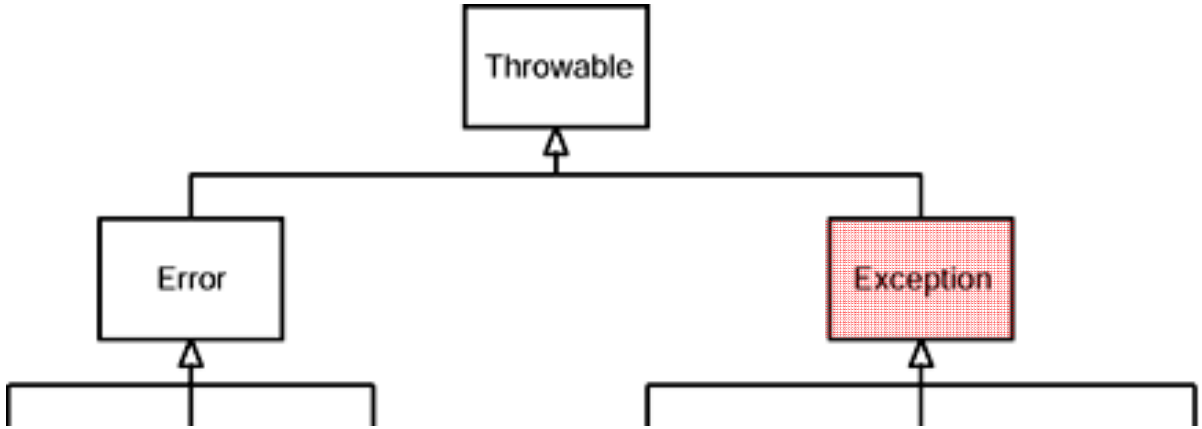
# Excepciones

- Mecanismo proporcionado por el lenguaje de programación para **notificar y tratar errores en tiempo de ejecución**.
- Soporte para la **robustez** del código.
- La información del error, **excepción**, es un **objeto** que se propaga a todos los objetos afectados por el error.
- Las excepciones pueden tratarse con el propósito de dar una solución al error: **recuperación de errores**.

# Situaciones de error

- Habitualmente las excepciones se utilizan en **situaciones de error que no pueden ser resueltas por el programador**:
  - Error en el hardware o sistema operativo: sacar un lápiz de memoria mientras se lee un fichero, la red no está disponible, etc.
  - Fallos en la ejecución de la máquina virtual.

# Jerarquía de excepciones en Java




- La jerarquía `Error` describe errores internos y agotamiento de recursos del sistema de ejecución de Java.
- El programador no debe lanzar objetos de tipo `Error`.
- El programador debe centrarse en las excepciones de tipo `Exception`.

# Excepciones comprobadas

- En Java las excepciones son objetos y **se definen utilizando una clase**.
- Las **excepciones comprobadas** heredan de `Exception`.

```
public class RedNoDisponible extends Exception {

        public RedNoDisponible() {
                super();
        }

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

- Todas las excepciones contienen un mensaje de error.

# Excepciones

- Las excepciones **se declaran** en los métodos y constructores que pueden lanzar esos errores (**throws**).

```
public String leerLinea() throws RedNoDisponible { … }
```

- Una excepción es **lanzada** utilizando **throw**:

```
throw new RedNoDisponible("La red no está disponible");
```

# Caso de estudio

- **Navegador web**.
- El navegador web define el método **`visualiza`**`()` encargado de **representar una página**.
- El método **`visualiza`**`()` hace uso de la clase **`Conexion`** encargada de establecer una conexión con un servidor web y recuperar un recurso (página web).
- La clase **`Conexion`** ofrece la siguiente funcionalidad:
  - Establece una conexión con el servidor web y abre el fichero cuando se construye el objeto.
  - Ofrece un método, **`leerLinea()`**, que devuelve las líneas del fichero.
  - Define un método para cerrar la conexión.

# Caso de estudio

- El programador de la clase **`Conexion`** se enfrenta a las siguientes **situaciones de error**:
  - *La red no está disponible*. Esta situación afecta al constructor y al método que lee las líneas.
  - *No se puede resolver la dirección del recurso*. Afecta al constructor.
- Estas situaciones de error evitan que las operaciones cumplan su especificación (**fallo en la postcondición**).

# Caso de estudio

- Para cada una de esas situaciones de error se definen **excepciones comprobadas**:
  - **`RedNoDisponible`**, **`RecursoNoEncontrado`**.

- Declara las excepciones en el constructor y los métodos:

```
public class Conexion {

   public Conexion(String url)
               throws RedNoDisponible, RecursoNoEncontrado
       { … }

   public String leerLinea() throws RedNoDisponible { … }

   public void cerrar() { … }
}
```

# Caso de estudio

- En el código, ante situaciones de error se lanzan excepciones:

```
public class Conexion {

   public Conexion(String url)
               throws RedNoDisponible, RecursoNoEncontrado
   {
   ...

   // La red no está disponible.
   // Lanza una excepción notificando el error

   throw new RedNoDisponible("La red no está disponible");

   }
   ...
}
```

# Caso de estudio

- En el navegador web , el método **`visualiza()`** realiza los siguientes pasos:
  - Crea un objeto conexión.
  - Lee las líneas del fichero para construir la representación de la página.
  - Representa la página.
  - Cierra la conexión.

# Caso de estudio

```
public void visualiza(String url) {

   Conexion conexion = new Conexion(url);

   String linea;
   do {
      linea = conexion.leerLinea();
      if (linea != null) {
         construyeRepresentacion(linea);
      }
   } while (linea != null);

   representacion();
   conexion.cerrar();
}
```

# Caso de estudio

- El navegador debe dar **tratamiento a las excepciones** de la clase **`Conexion`**.

- Al crear el objeto conexión:
  - *Red no disponible*: realizar varios reintentos esperando un intervalo de tiempo entre ellos. Si no se recupera, mostrar página de error.
  - *Recurso no encontrado*: mostrar página de error.

- Al leer la línea:
  - *Red no disponible*: igual que al crear la conexión, realizar varios intentos.

# Tratamiento de excepciones

- Java ofrece la construcción **try-catch** para tratar las excepciones que puedan producirse en el código.

- Esta construcción 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).

# Tratamiento de excepciones

- En Java las excepciones son objetos.
- Al producirse un error en el bloque *try* se revisa **por orden de declaración** los manejadores que pueden tratar el error.
- El primer manejador que sea compatible con el objeto de la excepción dará tratamiento al error.

  → **Sólo un manejador trata el error**.
- Esta comprobación utiliza la **compatibilidad de tipos (`instanceof`)**

# Tratamiento de excepciones

```
Conexion conexion = null;
int intentos = 0;
while (intentos < 20) {
    try {
        conexion = new Conexion(url);
        break;
    } catch (RedNoDisponible e) {
        Thread.sleep(1000); // Espera un segundo
        intentos++;
    } catch (RecursoNoEncontrado e) {
        paginaError("Recurso no encontrado");
        return;
    }
}
if (intentos == 20) {
     paginaError("Red no disponible");
}
```

# Tratamiento de excepciones

- En el ejemplo anterior se ha dado **tratamiento a las dos posibles excepciones** (**Caso 1**).

- **Caso 2**: Si no sabemos cómo dar tratamiento a un error, no se declara el manejador y **la excepción se deja pasar**.

- Es **obligatorio declarar las excepciones que escapan** en la cabecera del método.

# Tratamiento de excepciones

```
public void visualiza(String url)
            throws RecursoNoEncontrado  {

   Conexion conexion = null;
   int intentos = 0;
   while (intentos < 20) {
      try {
         conexion = new Conexion(url);
         break;
      } catch (RedNoDisponible e) {
         Thread.sleep(1000); // Espera un segundo
         intentos++;
      }
      // No se trata la excepción RecursoNoEncontrado
      // La excepción saldría del método.
   }
   …
```

# Tratamiento de excepciones

- **Caso 3**: Un solo manejador puede tratar varios tipos de excepciones que sean compatibles con su tipo.

- Ejemplo: `Exception` es la raíz de toda las excepciones. Se da un tratamiento común a las dos excepciones.

```
Conexion conexion = null;

try {
        conexion = new Conexion(url);

} catch (Exception e) {
        paginaError("Error de visualización");
        return;
}
```

# Relanzar una excepción

- **Caso 4**: es posible volver a lanzar una excepción utilizando **`throw`** → es tratada y sale del bloque try-catch.
- Ejemplo: si se alcanza el máximo de reintentos se relanza.
- El método **debe declarar la excepción en la cabecera**.

```
while (intentos < 20) {
   try {
      conexion = new Conexion(url);
      break;
   } catch (RedNoDisponible e) {
      Thread.sleep(1000); // Espera un segundo
      intentos++;
      if (intentos == 20) {
         throw e;
      }
   } …
```

# Excepciones significativas

- Un tipo de tratamiento de excepciones suele ser lanzar una **excepción más significativa**.

- Es útil para ocultar errores de bajo nivel:
  - No se puede abrir el socket de red, error de entrada/salida, etc.

- Se declara una excepción significativa en el método que lanza el error.

- El método atrapa las excepciones de bajo nivel y lanza la excepción más significativa.

# Excepciones significativas

- **Caso 5**:
  - En el caso de que el método `visualiza()` deje escapar las excepciones, podría definirse la excepción `ErrorVisualizacion` representando cualquier tipo de error producido en el método.
  - Las excepciones se atrapan y se lanza la nueva excepción.

```
Conexion conexion = null;

try {
      conexion = new Conexion(url);

} catch (Exception e) {
      throw new ErrorVisualizacion("Fallo conexión");
}
```

# Bloque `finally`

- El bloque `finally` es **opcional**.
- Si se declara, **siempre se ejecuta**, haya o no excepción, incluso si la excepción escapa.

```
...
try {
   conexion = new Conexion(url);
   break;
} catch (RedNoDisponible e) {
   Thread.sleep(1000); // Espera un segundo
   intentos++;
}
// No se trata la excepción RecursoNoEncontrado
// La excepción saldría del método.
finally {
   // Este bloque siempre se ejecuta
   comprobarCancelacion();
}
...
```

# Control de excepciones

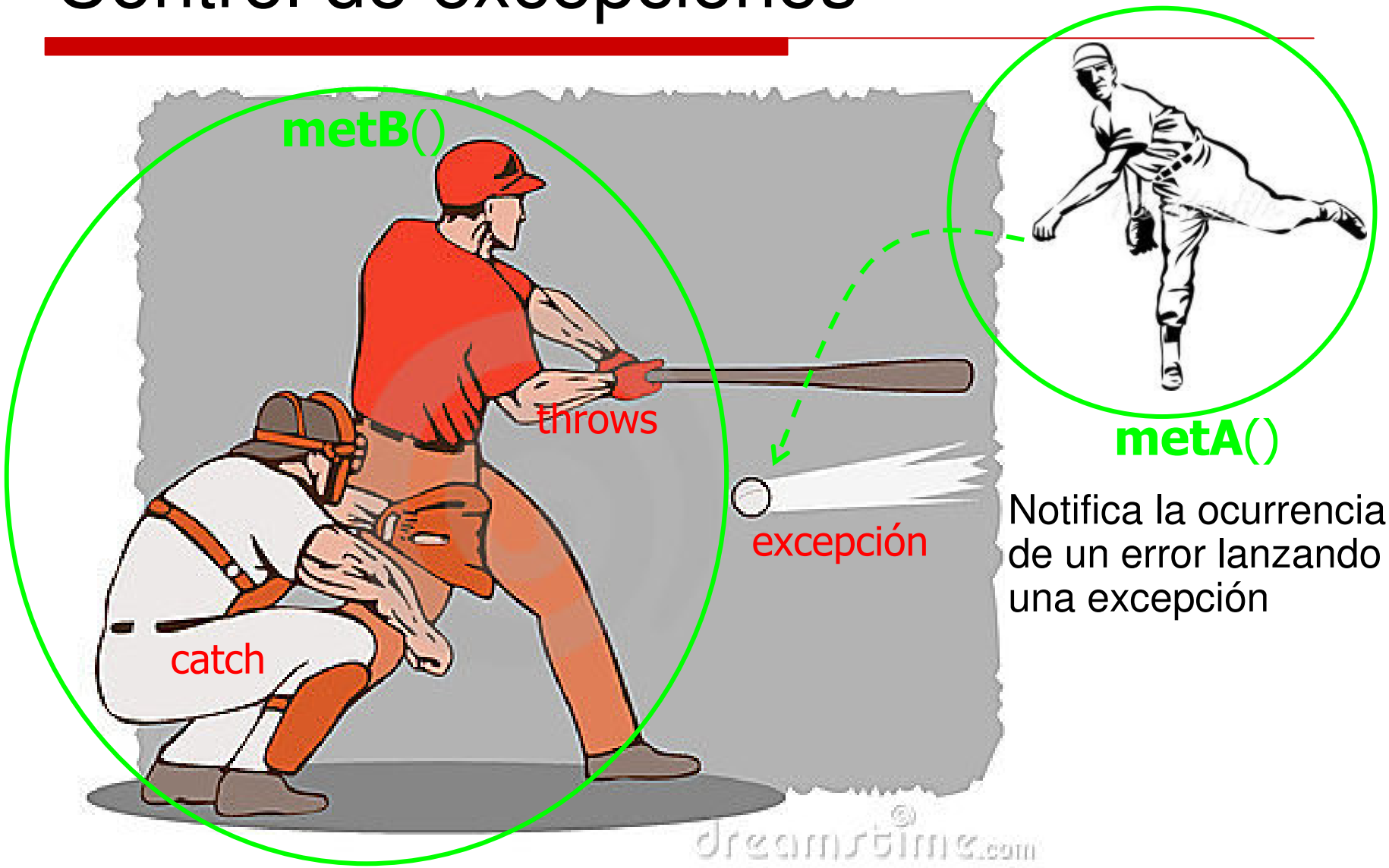
- El compilador realiza un **control de las excepciones comprobadas**.

- Si un método utiliza código 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*).

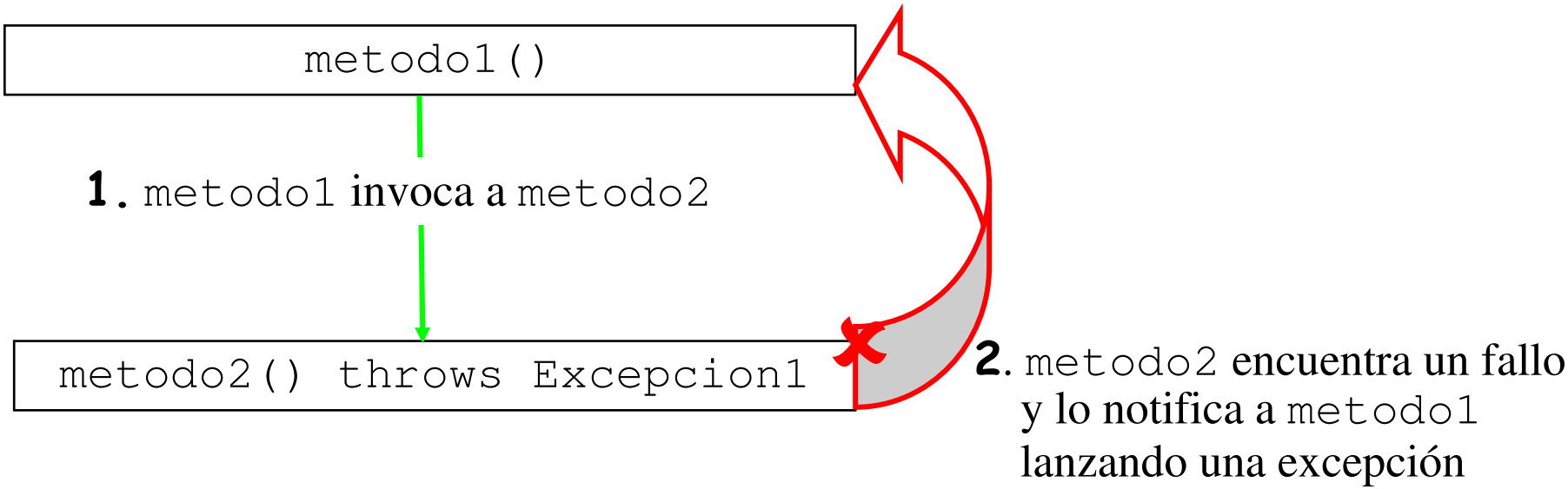
# Control de excepciones




Notifica la ocurrencia de un error lanzando una excepción

# Control de excepciones

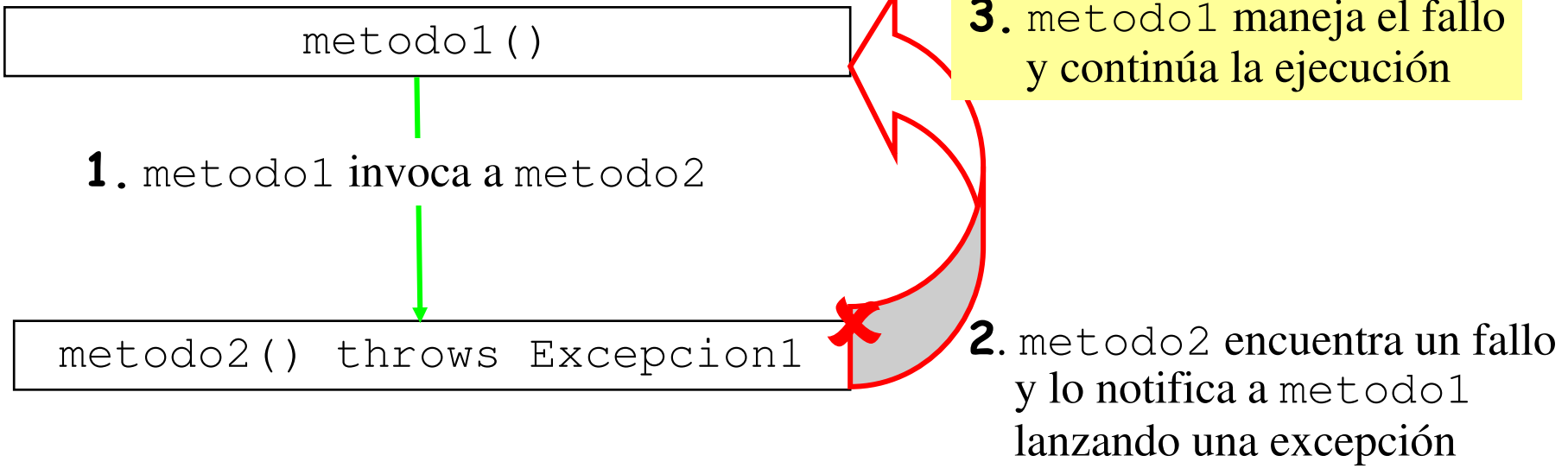
¿Qué hace `metodo1` cuando le llega una excepción?



# Control de excepciones

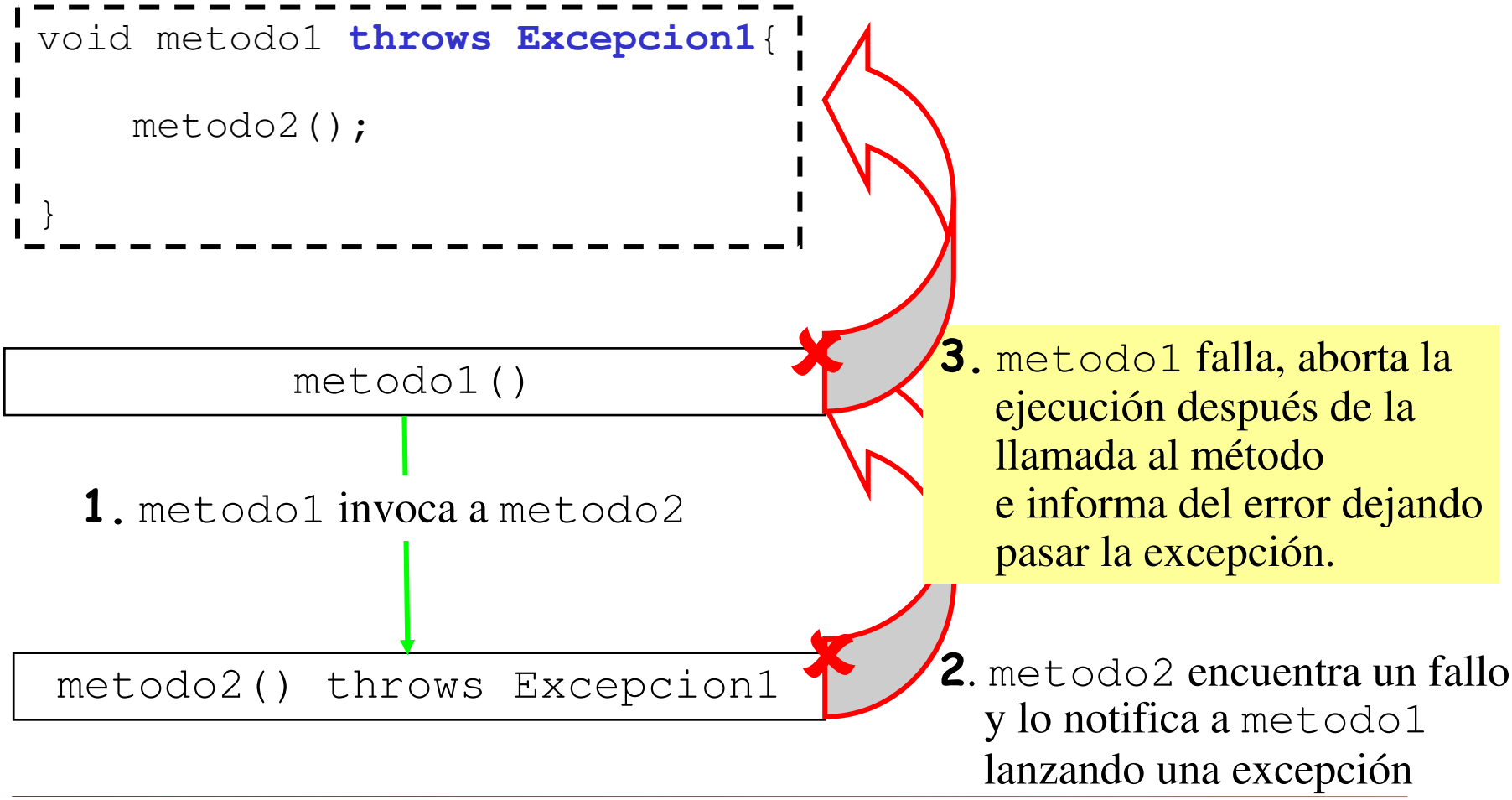
## a) `metodo1` define un manejador para tratar el error

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



# Control de excepciones

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



# Excepciones no tratadas

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

# Excepciones no tratadas

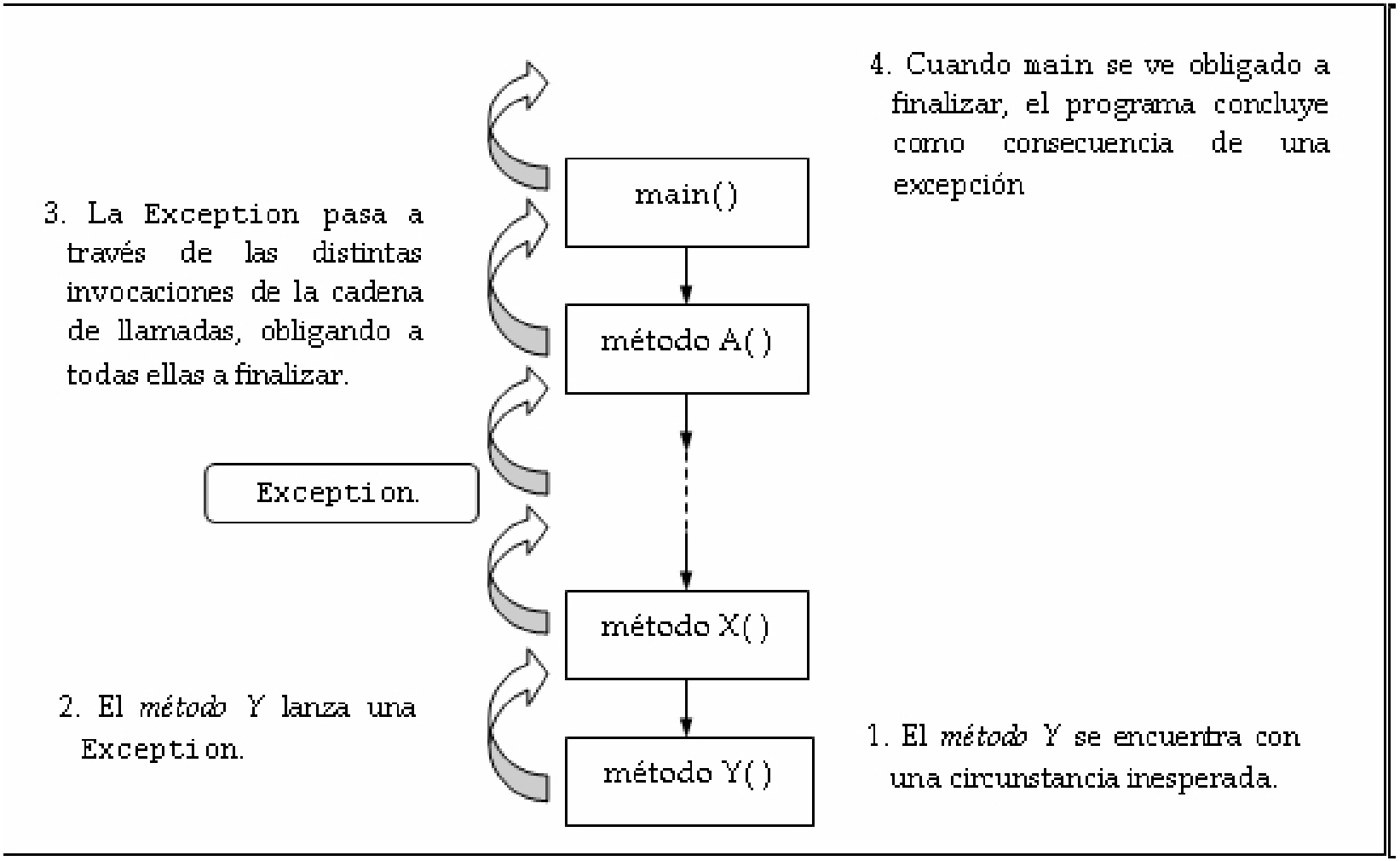



Imagen tomada de: **http://binarycodefree.blogspot.com/2010/02/control-de-excepciones-net.html**

# Excepciones no tratadas

- Algunas excepciones sólo pueden ser recuperadas con la **intervención del usuario**.

- Las excepciones para las que no existe recuperación de error en el código suelen propagarse hasta la **interfaz de usuario** (pantalla, página web).

- En la interfaz se notifica al usuario el error para que lo resuelva:
  - Ejemplo: los errores en el método `visualiza` del navegador se notifican mediante una página de error.

# Resumen. Excepciones comprobadas

- Las excepciones presentadas en el bloque II del tema reciben el nombre de **excepciones comprobadas**.
- Excepciones que representan una situación de error de la que **es posible tratar de recuperarse** en tiempo de ejecución.
- Subclases de `Exception`.
- **Tienen que declararse en la cabecera del método**.
- El compilador controla:
  - Que la excepción lanzada en el cuerpo del método es compatible con la declaración.
  - Que un método maneja las excepciones declaradas en otro método invocado.

# Excepciones comprobadas y Herencia

- Al redefinir un método heredado **podemos modificar la declaración de las excepciones** (*throws*).
- Sólo es posible **reducir** la lista de excepciones **comprobadas**.
- No se puede incluir una nueva *excepción comprobada* que no lance el método de la clase padre.
- Es posible indicar una excepción más específica que la que se hereda:
  - Ejemplo: en la clase padre el método lanza `IOException` y la redefinición `FileNotFoundException` que es un subtipo.

# Valores de retorno

- **¿Hay que notificar siempre los errores con excepciones?**
- También se pueden utilizar valores de retorno.

- Ejemplo: `boolean visualiza(String url)`
  - Si se produce un error, se notifica devolviendo un valor `false`.

# Excepciones vs. Valores de retorno

- **Problemas del uso de valores de retorno**:
  - Los **constructores** no tienen valor de retorno.
  - A veces no se puede devolver un valor especial.
    - Ejemplo: `int parseInt(String valor)`
  - Devolver un valor booleano es poco significativo.
    - Ejemplo: el método `visualiza()` puede tener dos tipos de errores.
  - **Java permite ignorar el valor de retorno al llamar a un método**.
- **¿Excepciones o valores de retorno?**
  - Depende del nivel de gravedad del error y la necesidad de información.

# Excepciones no comprobadas

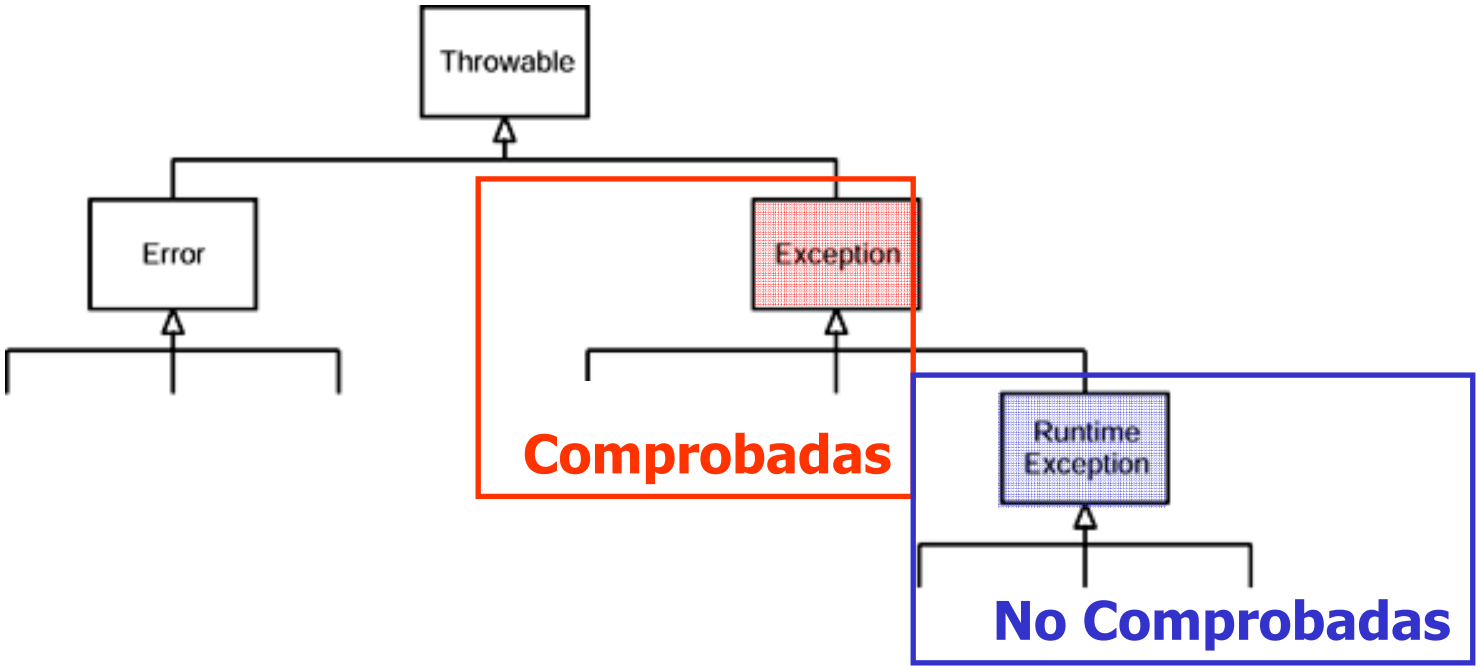
- **Nota**: en este apartado se contextualiza el uso de excepciones para controlar la **corrección del código** (**parte I del tema**)

- En Java también se utilizan excepciones para controlar el uso correcto del código (**Diseño por Contrato**).
- Las excepciones que se utilizan para notificar estos errores se denominan **excepciones *runtime***.
- También se conocen como "**no comprobadas**":
  - Si un método lanza una excepción no comprobada, no hay obligación de declararla.
  - Si un método utiliza otro método que lanza una excepción no comprobada, no hay obligación de tratarla.

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

# Excepciones no comprobadas

- Estas excepciones son subtipos de `RuntimeException`



# Excepciones no comprobadas

- Una excepción no comprobada se crea definiendo una clase que herede de **`RuntimeException`**.

- En general, no es necesario crear nuevas excepciones no comprobadas, ya que el lenguaje proporciona varias:
  - **`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.
  - …

# Consejos uso de excepciones

- **No debemos silenciar el tratamiento de una excepción** (manejador de excepción vacío)

  → Antes es preferible no tratarla y dejarla escapar.

- Si el tratamiento de error es notificar al usuario, la **notificación depende de la interfaz** (textual, gráfica).

- Al lanzar una excepción **establece el mensaje de error**. El mensaje de error puede ser mostrado al usuario:
  - `e.getMessage();`

# Consejos uso de excepciones

- Si varias instrucciones lanzan excepciones con el mismo tratamiento, es recomendable que un solo bloque try-catch envuelva a todas ellas.

- Para depurar una excepción muestra la **traza de del error**:
  - `e.printStackTrace()`

# Seminario 4

- El seminario 4 incluye varios ejemplos de uso de excepciones para controlar la corrección y robustez del código.
- Se aplica **diseño por contrato** en los ejemplos de los seminarios anteriores.
- Se desarrolla un ejemplo de una librería para el almacenamiento de empleados.