Programación en Lenguaje Java

Tema 9. Tratamiento de errores

Michael González Harbour

Mario Aldea Rivas

Departamento de Matemáticas,
Estadística y Computación

Programación en Java

1 ..., 2 ..., 3 ...

4. Datos Compuestos

5. Entrada/salida

6. Clases, referencias y objetos

7. Modularidad y abstracción

8. Herencia y polimorfismo

9. Tratamiento de errores

- Excepciones Java: generación y propagación automática. Bloques de tratamiento excepciones. La cláusula `finally`. Patrones de tratamiento de excepciones. Lanzar excepciones. Creación de excepciones propias. Excepciones "comprobadas" y cláusula `throws`. Notificación de errores mediante excepciones. Usos incorrectos de las excepciones

10. Entrada/salida con ficheros

11. Pruebas

9.1 Introducción

Durante la ejecución de una aplicación se dan ***circunstancias de error*** que impiden a los métodos realizar su cometido

Ejemplos de circunstancias de error:

- Llamada a un método con parámetros incorrectos
  - Valor numérico fuera de rango, …
- Estado incorrecto del objeto a la hora de invocar el método
  - Tratar de escribir en un fichero cerrado, …
- Operación incorrecta debida a un error en un algoritmo
  - División por 0, raíz cuadrada de número negativo, …
- Falta de un recurso
  - Espacio de memoria o disco agotado, dispositivo de red inalcanzable, …
- …

Los errores en un método ***nunca deben pasar inadvertidos***

Los ***errores previsibles***:

- Deben ser ***detectados lo antes posible***
- Deben ser ***notificados al método llamante*** (y quizá también al usuario)
- Su efecto debe ser ***corregido*** por la aplicación (siempre que sea posible)

Los ***errores imprevistos***

- es preferible que finalicen la aplicación (con un mensaje que permita su diagnosis),
- a que pasen inadvertidos causando un mal funcionamiento del sistema de difícil diagnóstico

Notificación de errores por valor de retorno

Es la única opción disponible en lenguajes de programación "antiguos" (C, Fortran, ...)

- Cada método o función retorna un valor
  - generalmente un código numérico o un booleano indicando si se ha producido un error o no, y qué error ha sido

Dos ***inconvenientes***:

- ***código poco legible***: el código de chequeo de error aparece por todas partes, mezclado con el código normal
- ***el chequeo de error se omite*** en muchos casos por "pereza" o desconocimiento
  - lleva a situaciones de error que pasan inadvertidas

Ejemplo

El método `añadeAlumno` de una supuesta clase `Curso` sería:

```
/** añade un alumno. Retorna false si no se ha
 *  podido añadir … */
public boolean añadeAlumno(Alumno a) {...}
```

Un fragmento de código que añade dos alumnos sería:

```
if (!curso.añadeAlumno(a1)) {
  error.escribe("Error añadiendo alumno");
  return;
}
if (!curso.añadeAlumno(a2)) {
  error.escribe("Error añadiendo alumno");
  return;
}
```

¡El tratamiento de los errores hace difícil entender el código!

Excepciones

Son un mecanismo especial presente en lenguajes "modernos" (Java, Ada, C++,...), para ***gestionar errores***

+ Permiten separar el tratamiento de errores del código normal
+ Evitan que los errores pasen inadvertidos
+ Permiten propagar de forma automática los errores desde los métodos más internos a los más externos
+ Permiten agrupar en un lugar común el tratamiento de errores que ocurren en varios lugares del programa
- mecanismo costoso en uso de memoria y procesador

Las excepciones se pueden ***lanzar***:

- ***automáticamente***, cuando el sistema detecta un error
- ***explícitamente*** cuando el programador lo establezca

Conceptos asociados a las excepciones

Lanzar

- La excepción se lanza para avisar de que hay un error
  - automáticamente
  - o explícitamente con la instrucción `throw`

Propagar

- La excepción se propaga de un bloque al siguiente hasta se trata

Tratar

- Ejecutar las instrucciones de un manejador de excepción

Manejador

- Instrucciones que se escriben para resolver un error

9.2 Excepciones Java: generación y propagación automática

Una excepción es generada de forma ***automática*** cuando

- la ***máquina virtual detecta una condición de error*** inesperada

Condición de error	Excepción generada
Error aritmético (x/0, ...)	`ArithmeticException`
Índice de array fuera de límites (<0 o >=length)	`ArrayIndexOutOfBoundsException`
Intento de convertir a una clase incorrecta	`ClassCastException`
Índice fuera de límites (p.e., en un `ArrayList`)	`IndexOutOfBoundsException`
Tamaño de array negativo	`NegativeArraySizeException`
Uso de una referencia nula	`NullPointerException`
Formato de número incorrecto	`NumberFormatException`
Índice fuera de límites en un String	`StringIndexOutOfBounds`
...	`...`

Ej. de generación automática: División por cero

```
public static void main(String[] args) {
int dividendo = 10, divisor = 0, resultado;

   System.out.println("Divide...");

   resultado = dividendo/divisor;

   System.out.println("resultado=" + resultado);
}
```

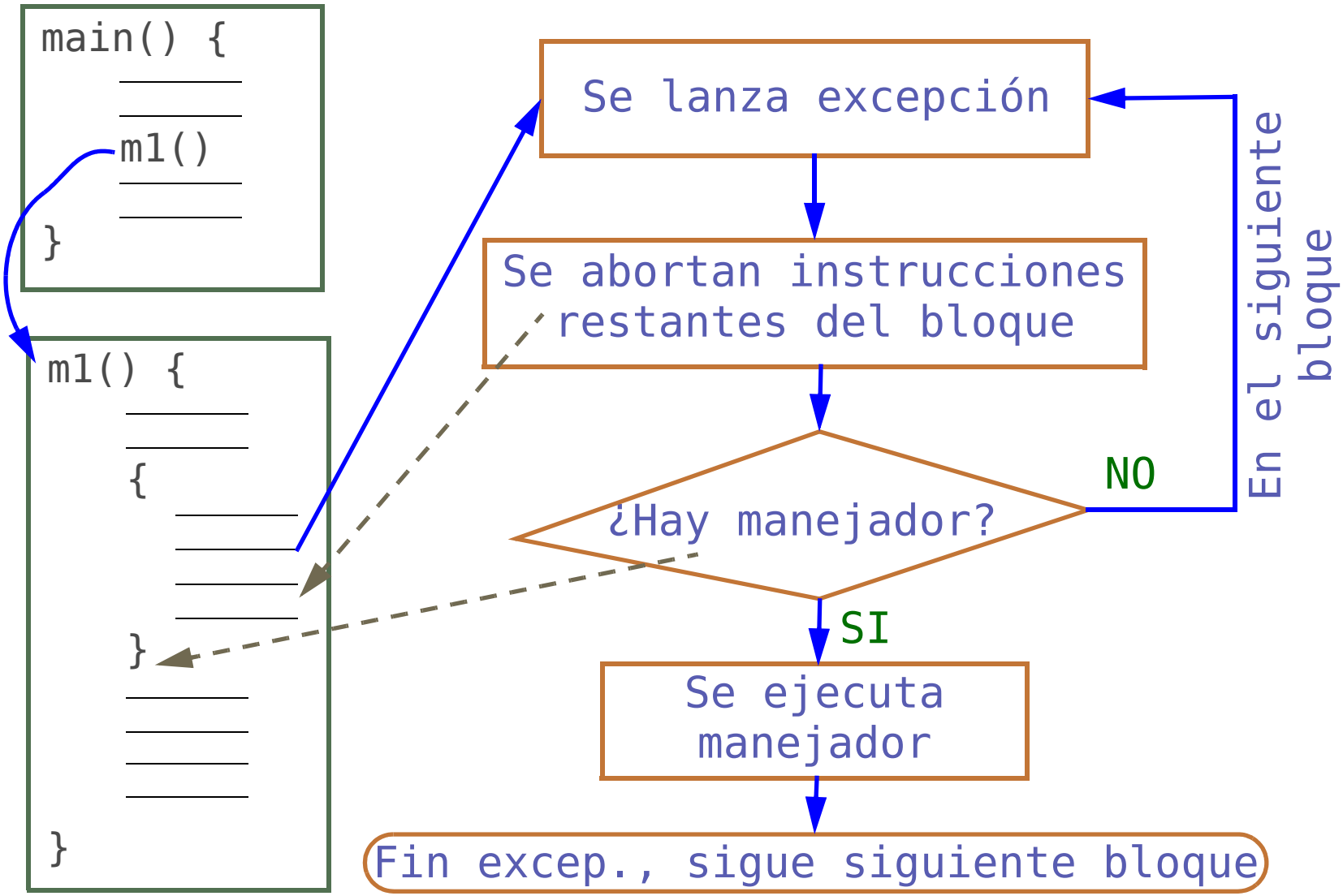
si `divisor` vale 0
se lanza la excepción
`ArithmeticException`

cuando se lanza la excepción
esta línea no se ejecuta

Salida por consola:

```
Divide...
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at tema05.div_por_cero.DivCeroSimple.main(DivCeroSimple.java:12)
```

Propagación de excepciones



Propagación de excepciones

1. Una línea de código ***genera o lanza*** (`throw`) una excepción
2. El bloque (método) que contiene esa línea de código se aborta en ese punto
3. Si el bloque ***trata*** esa excepción (tiene un ***manejador*** para ella), el manejador se ejecuta
   - diremos que el bloque ha ***tratado*** (`catch`) la excepción
   - la "vida" de la excepción finaliza en este punto
4. Si no tiene manejador, la excepción se ***propaga*** al bloque superior
   - que, a su vez, podrá coger o dejar pasar la excepción
5. Si la excepción alcanza el bloque principal (`main`) y éste tampoco coge la excepción, el programa finaliza con un mensaje de error

Ejemplo: propagación de excepciones

```
private static int divide(int a, int b) {
  System.out.println("divide: antes de dividir");
  int div = a/b;
  System.out.println("divide: después de dividir");
  return div;
}
private static void intermedio() {
  System.out.println("intermedio: antes de divide");
  int div = divide(2,0);
  System.out.println("intermedio: resultado:" +div);
}
public static void main(String[] args) {
  System.out.println("main: antes de intermedio");
  intermedio();
  System.out.println("main: después de intermedio");
}
```

cuando b vale 0 se lanza la excepción ArithmeticException

La salida generada será:

```
   main: antes de intermedio
   intermedio: antes de divide
   divide: antes de dividir
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Propaga.divide(Propaga.java:14)
   at Propaga.intermedio(Propaga.java:21)
   at Propaga.main(Propaga.java:27)
```

9.3 Bloques de tratamiento excepciones

La forma general de escribir un bloque en el que se tratan excepciones es:

```
try {
  instrucciones;
} catch (ClaseExcepción1 e) {
  instrucciones de tratamiento;
} catch (ClaseExcepción2 | ClaseExcepción3 e) {
  instrucciones de tratamiento;
}
```

Los "`catch`" se evalúan por orden:

- una excepción se coge en el primer "`catch`" para esa excepción o para una de sus superclases

Ejemplo: propagación con bloque `try-catch`

En el ejemplo "propagación de excepciones" anterior, añadimos un bloque `try-catch` al método intermedio:

```
private static void intermedio() {
  try {
    System.out.println("intermedio: antes de " +
                       "divide");
    int div=divide(2,0);
    System.out.println("intermedio: resultado:" +
                       div);
  } catch (ArithmeticException e) {
    System.out.println("intermedio: cazada " + e);
  }
}
```

La salida por consola que obtenemos ahora es:

```
main: antes de intermedio
intermedio: antes de divide
divide: antes de dividir
intermedio: cazada ArithmeticException: / by zero
main: después de intermedio
```

- en este caso la ***excepción es cazada***, por lo que
  - el ***programa NO finaliza*** de forma abrupta
  - NO aparece un mensaje del sistema indicando que se ha producido una excepción

Tratamiento específico

En el ejemplo anterior, el manejador realiza ***únicamente*** el tratamiento de la excepción `ArithmeticException`

```
try {
   ...;
} catch (ArithmeticException e) {
   ...;
}
```

Es posible poner un ***tratamiento común*** para cualquier excepción

```
try {
   ...;
} catch (Exception e) {
   ...;
}
```

- es cómodo pero ***no es recomendable***, ya que puede ocurrir un tratamiento inadecuado para una excepción no prevista

9.4 La cláusula `finally`

Permite crear un bloque de código que ***se ejecuta siempre*** después del bloque `try-catch` haya habido excepción o no

- incluso si se sale a causa de `return`, `break` o `continue`

```
try {
    operaciones;
} catch (ClaseExcepción1 e) {
    tratamiento de la excepción;
} catch (ClaseExcepción2 e) {
    tratamiento de la excepción;
} finally {
    instrucciones finales;
}
```

- la cláusula `finally` es opcional
- todo `try` debe tener al menos un `catch` o un `finally`

Secuencia con cláusula `finally`

Si ***NO se genera una excepción***, la secuencia es:

1. Operaciones
2. Instrucciones finales

Si se genera una ***excepción tratada***, la secuencia es:

1. Operaciones (incompletas)
2. Tratamiento
3. Instrucciones finales

Si se genera una ***excepción NO tratada***, la secuencia es:

1. Operaciones (incompletas)
2. Instrucciones finales
3. Se lanza la excepción en el siguiente bloque

9.5 Patrones de tratamiento de excepciones

Según la gravedad del error:

- *leve*: se notifica el error, pero la aplicación continúa
- *grave*: se notifica el error y se finaliza la aplicación
- *recuperable*: se reintenta la operación

Esquema de tratamiento de un ***error leve***

```
try {
  instrucciones
} catch (ClaseExcepción e) {
  notificación del error leve
}
```

Esquema de tratamiento de un *error grave*

```
try {
   instrucciones
} catch (ClaseExcepción e) {
   notificación del error grave
   System.exit(-1);  // finaliza la aplicación
}
```

Esquema de tratamiento de *error recuperable*

```
correcto = false
do {
   try {
      instrucciones a reintentar
      correcto = true
   } catch (ClaseExcepción e) {
      tratamiento
   }
} while (!correcto);
```

Ejemplo recuperable: lee dos notas

```
double nota1, nota2;
boolean notasCorrectas = false;
Lectura lec = new Lectura("Lee notas");
lec.creaEntrada("Nota parcial 1",5.0);
lec.creaEntrada("Nota parcial 2",5.0);
do {
  lec.esperaYCierra("Introduce notas");
  try {
    nota1=lec.leeDouble("Nota parcial 1");
    nota2=lec.leeDouble("Nota parcial 2");
    notasCorrectas = true; // sale del lazo
  } catch (NumberFormatException e) {
    // no muestra mensaje de error porque ya
    // lo hace leeDouble
  }
} while (!notasCorrectas);
```

Ejemplo: lee notas en el rango [0.0, 10.0]

```
do {
  lec.esperaYCierra("Introduce notas");
  try {
    nota1=lec.leeDouble("Nota parcial 1");
    nota2=lec.leeDouble("Nota parcial 2");
    if (nota1>=0.0 && nota1<=10.0 &&
        nota2>=0.0 && nota2<=10.0) {
      notasCorrectas = true; // sale del lazo
    } else {
      Mensaje error = new Mensaje();
      error.escribe("Rango incorrecto");
    }
  } catch (NumberFormatException e) {
    // no muestra mensaje de error porque ya
    // lo hace leeDouble
  }
} while (!notasCorrectas);
```

9.6 Lanzar excepciones

Pueden ***lanzarse explícitamente*** desde el programa utilizando la palabra reservada **`throw`**:

```
throw new ClaseExcepción();
```

En ocasiones puede ser más conveniente usar el constructor con un string como parámetro

```
throw new ClaseExcepción("mensaje");
```

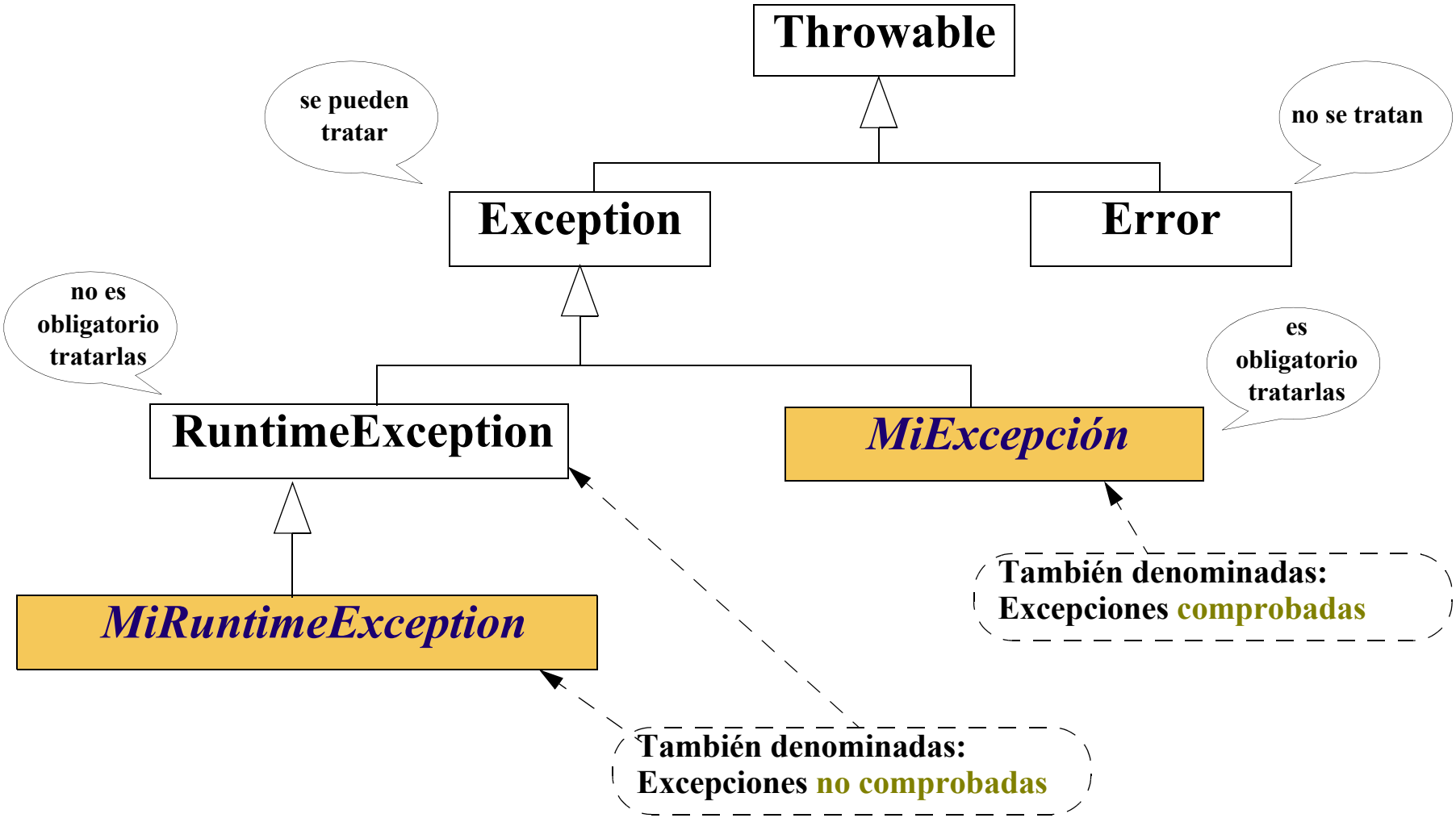
- que sirve para dar información adicional sobre la causa de la excepción

Ejemplo:

```
...
if (clave==null) {
  throw new NullPointerException("clave es nula");
}
```

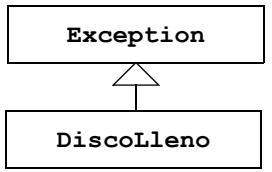
9.7 Creación de excepciones propias

Jerarquía de las excepciones y excepciones propias:



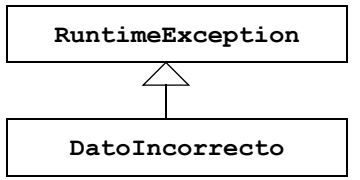
Crear excepciones propias

El programador puede crear sus propias excepciones extendiendo las clases `Exception` o `RuntimeException`:




```
public static class DiscoLleno extends Exception {}
```

Ponemos `static` cuando la excepción se declara como una clase anidada de otra clase principal




```
public static class DatoIncorrecto
                    extends RuntimeException {}
```

Si queremos asociar mensajes de texto con información adicional sobre el error, debemos escribir los constructores:


```
public static class MiExcepción extends Exception {
    /**
     * constructor sin mensaje de error
     */
    public MiExcepción() {
      super();
    }
    /**
     * constructor con mensaje de error
     */
    public MiExcepción(String infoAdicional) {
      super(infoAdicional);
    }
}
```

Ejemplo de uso de excepciones propias

```
public class Empleado {
   private double sueldo;
   ... // otros atributos
   /**
    * Lanzada por asignaSueldo cuando se trata de asignar un sueldo
    * negativo
    */
   public static class SueldoIncorrecto extends RuntimeException {}
      ... // otros métodos
      /**
    * Asigna el sueldo al empleado
    * @param sueldo sueldo a asignar al empleado
    * @throws SueldoIncorrecto si se trata de asignar un sueldo
    * negativo
    */
   public void asignaSueldo(double sueldo) {
      if (sueldo < 0) {
         throw new SueldoIncorrecto();
      }
      this.sueldo = sueldo;
   }
}
```

9.8 Excepciones "comprobadas" y cláusula `throws`

Las excepciones propias que extienden a la clase `Exception`

- son ***excepciones comprobadas***
- (si extendieran `RuntimeException` serían "no comprobadas")

Un método que ***podría lanzar o propagar*** una excepción comprobada, deberá:

- ***tratarla*** con un bloque `try-catch`
- ***o declarar que la propaga*** con una cláusula `throws`

Sintaxis de la cláusula `throws`

```
public tipo nombreMétodo(parámetros)
            throws ClaseExcepción1, ClaseExcepción2 {
  instrucciones; // podrían lanzar las excepciones
}
```

Cláusula throws en métodos anidados

```
public void método3() throws MiExcepción {
   ...
   if (...)
      throw new MiExcepción();
}
public void método2() throws MiExcepción {
   ...
   método3();
}
public void método1() {
   ...
   try {
     método2();
   } catch (MiExcepción e) {
      ...
   }
}
```

podría lanzar la excepción, por eso lo indica

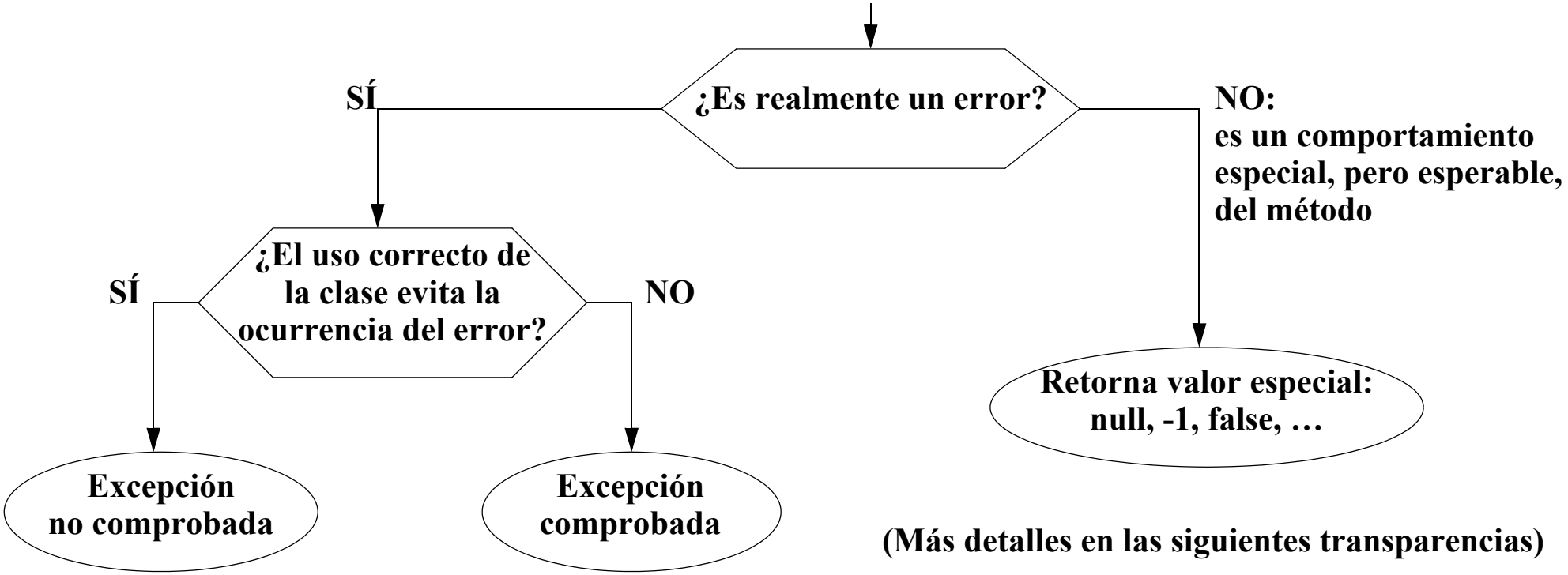
NO trata la excepción (la propaga), por eso indica que podría lanzarla

trata la excepción, por eso no tiene cláusula `throws`

9.9 Notificación de errores mediante excepciones

Para decidir la ***forma de notificar un error*** debemos preguntarnos:

1. ¿Se trata realmente de un error?
2. Si es un error: ¿Uso excepción comprobada o no comprobada?



Comportamientos especiales de los métodos

Comportamientos especiales:

- Método que busca la primera ocurrencia de un carácter en un string: ¿cómo notificar que el carácter no está en el string?
- Método que busca un alumno en un curso: ¿cómo notificar que el alumno no se encuentra matriculado en ese curso?

En muchas ocasiones se confunden con errores, pero no lo son:

- Son ***situaciones esperables*** que siempre hay que comprobar después de llamar al método
- Se notifican ***retornando un valor especial*** (`null`, `-1`, `false`, ...)

Ejemplo de retorno de un valor especial

```
/**
 * Busca una palabra en el diccionario
 * @param palabra palabra de la que se busca el significado
 * @return significado de la palabra o
 *      null en el caso de que la palabra no esté en el diccionario
 */
public String significado(String palabra) {...}
```

Notificación de errores mediante excepciones comprobadas

Utilizaremos ***excepciones comprobadas*** cuando ***no es sencillo anticipar*** si va a ocurrir el error o no

- en ese caso la excepción constituye la única manera de saber que se ha producido el error

Al tratarse de ***excepciones comprobadas*** el programador se ve ***obligado a incluir su tratamiento***

Muy utilizadas en la gestión de ficheros

- Situaciones como "Falta de permisos", "No queda espacio en el disco", etc. son ejemplos claros de errores no fácilmente anticipables que el programa no debe ignorar

Ejemplo de uso de ***excepción comprobada***:

- Clase `InterfazRed` que permite enviar mensajes a otro computador
- El computador destino puede dejar de estar alcanzable:
  - La red puede sufrir cortes momentáneos, el computador destino puede apagarse, ...
- En un instante dado el computador destino puede estar alcanzable y al siguiente no estarlo y viceversa

```
public class InterfazRed {
   ... // atributos
   /**
    * Lanzada por enviaMensaje cuando se detecta que el computador
    * de destino no es alcanzable
    */
   public static class DestinoNoAlcanzable extends Exception {}
   ... // otros métodos
   /**
    * Envía un mensaje a la dirección indicada
    * @param dirDestino dirección del computador destino
    * @param msj mensaje a enviar
    * @throws DestinoNoAlcanzable si se detecta que el computador
    * de destino no es alcanzable
    */
   public void enviaMensaje(String dirDestino,
                            String msj) throws DestinoNoAlcanzable {
      ...
      if (detecta que el destino no es alcanzable)
         throw new DestinoNoAlcanzable();
      ...
   }
}
```

excepción comprobada

Avisa de que el método podría lanzar la excepción

Lanza la excepción (si se da la situación de error)

Ejemplo de uso de la clase `InterfazRed`:

```
InterfazRed interfazRed = new InterfazRed();

// lazo de envío de mensajes
while (true) {
   try {
      String mensaje = pideMensajeAlUsuario();
      interfazRed.enviaMensaje("192.168.0.1",
                               mensaje);
   } catch (InterfazRed.DestinoNoAlcanzable e) {
      error("El destino no es alcanzable");
   }
}
```

Notificación de errores mediante excepciones no comprobadas

Utilizaremos ***excepciones no comprobadas*** cuando:

1. Es ***posible anticipar el error*** (mediante otros métodos de la clase)
   - por lo tanto, si la clase se usa correctamente, la excepción no tendría porqué generarse nunca
   - los bloques `try-catch` serían superfluos
   - en este caso la excepción sirve para detectar (y corregir) usos incorrectos de la clase
2. Se trata de un ***error interno*** del método, ante el que el programador que usa la clase poco puede hacer:
   - fallos en precondiciones de métodos privados
   - fallos en postcondiciones en métodos públicos o privados

Ejemplo de uso de ***excepciones no comprobadas***:

- Clase `Curso` con un cupo máximo de alumnos matriculados
- Al añadir un alumno al curso pueden producirse dos errores:
  - el curso está completo (ha cubierto su cupo)
  - ya existe un alumno en el curso con el mismo DNI que el que se pretende matricular
- Existen los métodos `completo()` y `buscaAlumnoPorDNI()` que permiten anticipar los errores

Utilizaremos las ***excepciones no comprobadas*** `Completo` y `DNIAlumnoRepetido`

- para no obligar a poner bloques `try-catch` en operaciones en las que se sabe que el error no se va a producir

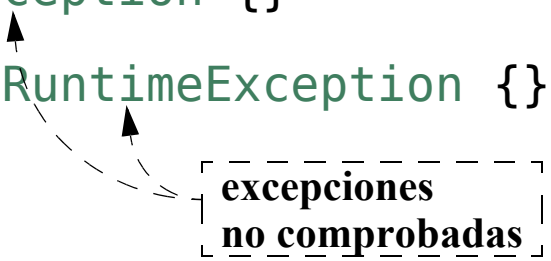
```
public class Curso {
    private final int cupo;
    private ArrayList<Alumno> listaAlumnos = new ArrayList<Alumno>();

    public static class Completo extends RuntimeException {}

    public static class DNIAlumnoRepetido extends RuntimeException {}

    /**
     * Construye un curso con el cupo indicado
     * @param cupo cupo del curso
     */
    public Curso(int cupo) {
        this.cupo = cupo;
    }

    /**
     * Indica si el curso está completo
     * @return true si el curso está completo
     */
    public boolean completo() {
        return listaAlumnos.size() >= cupo;
    }
```



```
/**
 * Busca en el curso un alumno con el DNI indicado
 * @param dni DNI del alumno buscado
 * @return el alumno con el DNI indicado o
 *           null si no hay ningún alumno con ese DNI
 */
public Alumno buscaAlumnoPorDNI(String dni) {
    for(Alumno a: listaAlumnos) {
        if (dni.equals(a.dni())){
            return a;
        }
    }
    return null;
}
```

- El que no exista ningún alumno con el DNI indicado es un ***comportamiento especial no erróneo*** para el que NO debe utilizarse una excepción

```
    /**
     * Añade el alumno al curso.
     * @param alumno alumno a añadir
     * @throws Completo cuando se ha alcanzado el cupo del curso
     * @throws AlumnoConDNIRepetido cuando ya existe otro alumno
     * con el mismo DNI que el que se pretende añadir
     */
    public void añadeAlumno(Alumno alumno) throws Completo,
                                                  DNIAlumnoRepetido {

        if (completo()) {
            throw new Completo();
        }
        if (buscaAlumnoPorDNI(alumno.dni()) != null) {
            throw new DNIAlumnoRepetido();
        }

        listaAlumnos.add(alumno);
    }

    ... // otros métodos
}
```

Al tratarse de excepciones "No comprobadas" no es obligatorio poner el "throws", pero le ponemos por claridad

Código para añadir un alumno al curso:

```
case AÑADE_ALUMNO:
  if (curso.completo()) {
     error("Alcanzado el cupo del curso");

  } else {
     Alumno alumno = pideDatosAlumno();

     try {
        curso.añadeAlumno(alumno);
     } catch (Curso.DNIAlumnoRepetido e) {
        error("Existe alumno con el mismo DNI");
     }

  }
  break;
```

Hay que tratar `AlumnoConDNIRepetido` puesto que el usuario podría confundirse al introducir los datos del alumno

No hace falta tratar `Completo` puesto que ya hemos descartado ese error

Código para añadir varios alumnos al curso

- Por la lógica de mi programa debería verificarse que ninguno de los alumnos prematriculados está en el curso
- Si por algún error anterior, algún alumno prematriculado estuviera ya en el curso, la excepción nos serviría para detectar el error y corregirlo en la siguiente versión del código

```
Alumno[] alumnosPrematriculados = ...;
...
int i=0;
while (i<alumnosPrematriculados.length &&
       !curso.completo())
{
  curso.añadeAlumno(alumnosPrematriculados[i]);
  i++;
}
```

El que las excepciones sean no comprobadas nos ahorra poner los bloques `try-catch` que en este caso son innecesarios

9.10 Usos incorrectos de las excepciones

No deben usarse las excepciones en ***casos que no sean de error***

- por eficiencia
- y porque hacen más difícil entender el programa

No debe ***lanzarse una excepción y tratarla en el mismo método***

- en lugar de lanzar la excepción, realizar el tratamiento directamente
- excepto si ya hay un manejador escrito que sea adecuado

No deben ***usarse excepciones para realizar el control de flujo***

- por ejemplo para salirse de un lazo o una operación
- (ver ejemplo en la siguiente transparencia)

Forma ***incorrecta*** de añadir varios alumnos a un curso

- utiliza la excepción para realizar el flujo de control

```
Alumno[] alumnosPrematriculados = ...;
...

try {

  for(int i=0; i<alumnosPrematriculados.length; i++) {
    curso.añadeAlumno(alumnosPrematriculados[i]);
  }

} catch (Curso.Completo e) {
   // no hace nada, sólo sirve para salir del lazo
   // cuando el curso está completo
}
```

La forma correcta es la que aparece en la transparencia página 45