# 10: Manejo de errores con excepciones

La filosofía básica de Java es que "el código mal formado no se ejecutará".

El momento ideal para capturar un error es en tiempo de compilación, antes incluso de intentar ejecutar el programa. Sin embargo, no todos los errores se pueden detectar en tiempo de compilación. El resto de los problemas deberán ser manejados en tiempo de ejecución, mediante alguna formalidad que permita a la fuente del error pasar la información adecuada a un receptor que sabrá hacerse cargo de la dificultad de manera adecuada.

En C y en otros lenguajes anteriores, podía haber varias de estas formalidades, que generalmente se establecían por convención y no como parte del lenguaje de programación. Habitualmente, se devolvía un valor especial o se ponía un indicador a uno, y el receptor se suponía, que tras echar un vistazo al valor o al indicador, determinaría la existencia de algún problema. Sin embargo, con el paso de los años, se descubrió que los programadores que hacían uso de bibliotecas tendían a pensar que eran invencibles —como en "Sí, puede que los demás cometan errores, pero no en *mi* código". Por tanto, y lógicamente, éstos no comprobaban que se dieran condiciones de error (y en ocasiones las condiciones de error eran demasiado estúpidas como para comprobarlas)[1]. Si uno *fuera* tan exacto como para comprobar todos los posibles errores cada vez que se invocara a un método, su código se convertiría en una pesadilla ilegible. Los programadores son reacios a admitir la verdad: este enfoque al manejo de errores es una grandísima limitación especialmente de cara a la creación de programas grandes, robustos y fácilmente mantenibles.

La solución es extraer del manejo de errores la naturaleza casual de los mismos y forzar la formalidad. Esto se ha venido haciendo a lo largo de bastante tiempo, dado que las implementaciones de *manejo de excepciones* se retornan hasta los sistemas operativos de los años 60, e incluso al "**error goto**" de BASIC. Pero el manejo de excepciones de C++ se basaba en Ada, y el de Java está basado fundamentalmente en C++ (aunque se parece incluso más al de Pascal Orientado a Objetos).

La palabra "excepción" se utiliza en el sentido de: "Yo me encargo de la excepción a eso". En cuanto se da un problema puede desconocerse qué hacer con el mismo, pero se sabe que simplemente no se puede continuar sin más; hay que parar y alguien, en algún lugar, deberá averiguar qué hacer. Pero puede que no se disponga de información suficiente en el contexto actual como para solucionar el problema. Por tanto, se pasa el problema a un contexto superior en el que alguien se pueda encargar de tomar la decisión adecuada (algo semejante a una cadena de comandos).

El otro gran beneficio de las excepciones es que limpian el código de manejo de errores. En vez de comprobar si se ha dado un error en concreto y tratar con él en diversas partes del programa, no es necesario comprobar nada más en el momento de invocar al método (puesto que la excepción ga-

---

[1] Como ejemplo de esta afirmación, un programador en C puede comprobar el valor que devolvía la función **printf( )**.

rantizará que alguien la capture). Y es necesario manejar el problema en un solo lugar, el denominado *gestor de excepciones*. Éste salva el código, y separa el código que describe qué se desea hacer a partir del código en ejecución cuando algo sale mal. En general, la lectura, escritura, y depuración de código se vuelve mucho más sencilla con excepciones, que cuando se hace uso de la antigua manera de gestionar los errores.

Debido a que el manejo de excepciones se ve fortalecido con el compilador de Java, hay numerosísimos ejemplos en este libro que permiten aprender todo lo relativo al manejo de excepciones. Este capítulo presenta el código que es necesario escribir para gestionar adecuadamente las excepciones, y la forma de generar excepciones si algún método se mete en problemas.

# Excepciones básicas

Una *condición excepcional* es un problema que evita la continuación de un método o el alcance actual. Es importante distinguir una condición excepcional de un problema normal, en el que se tiene la suficiente información en el contexto actual como para hacer frente a la dificultad de alguna manera. Con una condición excepcional no se puede continuar el proceso porque no se tiene la información necesaria para tratar el problema, en el *contexto actual*. Todo lo que se puede hacer es salir del contexto actual y relegar el problema a un contexto superior. Esto es lo que ocurre cuando se lanza una excepción.

Un ejemplo sencillo es una división. Si se corre el riesgo de dividir entre cero, merece la pena comprobar y asegurarse de que no se seguirá adelante y se llegará a ejecutar la división. Pero ¿qué significa que el denominador sea cero? Quizás se sabe, en el contexto del problema que se está intentando solucionar en ese método particular, cómo manejar un denominador cero. Pero si se trata de un valor que no se esperaba, no se puede hacer frente a este error, por lo que habrá que lanzar una excepción en vez de continuar hacia delante.

Cuando se lanza una excepción, ocurren varias cosas. En primer lugar, se crea el objeto excepción de la misma forma en que se crea un objeto Java: en el montículo, con **new**. Después, se detiene el cauce normal de ejecución (el que no se podría continuar) y se lanza la referencia al objeto excepción desde el contexto actual. En este momento se interpone el mecanismo de gestión de excepciones que busca un lugar apropiado en el que continuar ejecutando el programa. Este lugar apropiado es el *gestor de excepciones*, cuyo trabajo es recuperarse del problema de forma que el programa pueda, o bien intentarlo de nuevo, o bien simplemente continuar.

Como un ejemplo sencillo de un lanzamiento de una excepción, considérese una referencia denominada **t**. Es posible que se haya recibido una referencia que no se haya inicializado, por lo que sería una buena idea comprobarlo antes de que se intentara invocar a un método utilizando esa referencia al objeto. Se puede enviar información sobre el error a un contexto mayor creando un objeto que represente la información y "arrojándolo" fuera del contexto actual. A esto se le llama *lanzamiento de una excepción*. Tiene esta apariencia:

```
if (t == null)
  throw new NullPointerException();
```

Esto lanza una excepción, que permite —en el contexto actual— abdicar la responsabilidad de pensar sobre este aspecto más adelante. Simplemente se gestiona automáticamente en algún otro sitio. El *dónde* se mostrará más tarde.

## Parámetros de las excepciones

Como cualquier otro objeto en Java, las excepciones siempre se crean en el montículo haciendo uso de **new**, que asigna espacio de almacenamiento e invoca a un constructor. Hay dos constructores en todas las excepciones estándar: el primero es el constructor por defecto y el segundo toma un parámetro string de forma que se pueda ubicar la información pertinente en la excepción:

```
if (t == null)
  throw new NullPointerException("t = null");
```

Este string puede extraerse posteriormente utilizando varios métodos, como se verá más adelante.

La palabra clave **throw** hace que ocurran varias cosas relativamente mágicas. Habitualmente, se usará primero **new** para crear un objeto que represente la condición de error. Se da a **throw** la referencia resultante. En efecto, el método "devuelve" el objeto, incluso aunque ese tipo de objeto no sea el que el método debería devolver de forma natural. Una manera natural de pensar en las excepciones es como si se tratara de un mecanismo de retorno alternativo, aunque el que lleve esta analogía demasiado lejos acabará teniendo problemas. También se puede salir del ámbito ordinario lanzando una excepción. Pero se devuelve un valor, y el método o el ámbito finalizan.

Cualquier semejanza con un método de retorno ordinario acaba aquí, puesto que el punto de retorno es un lugar completamente diferente del punto al que se sale en una llamada normal a un método. (Se acaba en un gestor de excepciones adecuado que podría estar a cientos de kilómetros —es decir, mucho más bajo dentro de la pila de invocaciones— del punto en que se lanzó la excepción.)

Además, se puede lanzar cualquier tipo de objeto lanzable **Throwable** que se desee. Habitualmente, se lanzará una clase de excepción diferente para cada tipo de error. La información sobre cada error se representa tanto dentro del objeto excepción como implícitamente en el tipo de objeto excepción elegido, puesto que alguien de un contexto superior podría averiguar qué hacer con la excepción. (A menudo, la única información es el tipo de objeto excepción, y no se almacena nada significativo junto con el objeto excepción.)

# Capturar una excepción

Si un método lanza una excepción, debe asumir que esa excepción será "capturada" y que será tratada. Una de las ventajas del manejo de excepciones de Java es que te permite concentrarte en el problema que se intenta solucionar en un único sitio, y tratar los errores que ese código genere en otro sitio.

Para ver cómo se captura una excepción, hay que entender primero el concepto de *región guardada*, que es una sección de código que podría producir excepciones, y que es seguida del código que maneja esas excepciones.

# El bloque **try**

Si uno está dentro de un método y lanza una excepción (o lo hace otro método al que se invoque), ese método acabará en el momento en que haga el lanzamiento. Si no se desea que una **excepción** implique abandonar un método, se puede establecer un bloque especial dentro de ese método para que capture la excepción. A este bloque se le denomina el *bloque try* puesto que en él se "intentan" varias llamadas a métodos. El bloque try es un ámbito ordinario, precedido de la palabra clave **try**:

```
try {
  // Código que podría generar excepciones
}
```

Si se estuviera comprobando la existencia de errores minuciosamente en un lenguaje de programación que no soporte manejo de excepciones, habría que rodear cada llamada a método con código de prueba de invocación y errores, incluso cuando el mismo método fuese invocado varias veces. Esto significa que el código es mucho más fácil de escribir y leer debido a que no se confunde el objetivo del código con la comprobación de errores.

# Manejadores de excepciones

Por supuesto, la excepción que se lance debe acabar en algún sitio. Este "sitio" es el *manejador de excepciones* y hay uno por cada tipo de excepción que se desee capturar. Los manejadores de excepciones siguen inmediatamente al bloque try y se identifican por la palabra clave **catch**:

```
try {
  // Código que podría generar excepciones
} catch(Tipo1 id1) {
  // Manejo de excepciones de Tipo1
} catch(Tipo2 id2) {
  // Manejo de excepciones de Tipo2
} catch(Tipo3 id3) {
  // Manejo de excepciones de Tipo3
}

// etc. . .
```

Cada cláusula *catch* (manejador de excepciones) es semejante a un pequeño método que toma uno y sólo un argumento de un tipo en particular. El identificador (**id1**, **id2**, y así sucesivamente) puede usarse dentro del manejador, exactamente igual que un parámetro de un método. En ocasiones nunca se usa el identificador porque el tipo de la excepción proporciona la suficiente información como para tratar la excepción, pero el identificador debe seguir ahí.

Los manejadores deben aparecer directamente tras el bloque *try*. Si se lanza una excepción, el mecanismo de gestión de excepciones trata de cazar el primer manejador con un argumento que coincida con el tipo de excepción. Posteriormente, entra en esa cláusula *catch*, y la excepción se da por manejada. La búsqueda de manejadores se detiene una vez que se ha finalizado la cláusula *catch*. Sólo se ejecuta la cláusula *catch*; no es como una sentencia **switch** en la que haya que colocar un **break** después de cada **case** para evitar que se ejecute el resto.

Fíjese que, dentro del bloque *try*, varias llamadas a métodos podrían generar la misma excepción, pero sólo se necesita un manejador.

### Terminación o reanudación

Hay dos modelos básicos en la teoría de manejo de excepciones. En la *terminación* (que es lo que soportan Java y C++) se asume que el error es tan crítico que no hay forma de volver atrás a resolver dónde se dio la excepción. Quien quiera que lanzara la excepción decidió que no había forma de resolver la situación, y no *quería* volver atrás.

La alternativa es el *reanudación*. Significa que se espera que el manejador de excepciones haga algo para rectificar la situación, y después se vuelve a ejecutar el método que causó el error, presumiendo que a la segunda no fallará. Desear este segundo caso significa que se sigue pensando que la excepción continuará tras el manejo de la excepción. En este caso, la excepción es más como una llamada a un método —que es como deberían establecerse en Java aquellas situaciones en las que se desea este tipo de comportamiento. (Es decir, es mejor llamar a un método que solucione el problema antes de lanzar una excepción.) Alternativamente, se ubica el bloque **try** dentro de un bucle **while** que sigue intentando volver a entrar en el bloque **try** hasta que se obtenga el resultado satisfactorio.

Históricamente, los programadores que usaban sistemas operativos que soportaban el manejo de excepciones reentrantes acababan usando en su lugar código con terminación. Por tanto, aunque la técnica de los reintentos parezca atractiva a primera vista, no es tan útil en la práctica. La razón dominante es probablemente el *acoplamiento* resultante: el manejador debe ser, a menudo, consciente de dónde se lanza la excepción y contener el código no genérico específico del lugar de lanzamiento. Esto hace que el código sea difícil de escribir y mantener, especialmente en el caso de sistemas grandes en los que la excepción podría generarse en varios puntos.

## Crear sus propias excepciones

No hay ninguna limitación que obligue a utilizar las excepciones existentes en Java. Esto es importante porque a menudo será necesario crear sus propias excepciones para indicar un error especial que puede crear su propia biblioteca, pero que no fue previsto cuando se creó la jerarquía de excepciones de Java.

Para crear su propia clase excepción, se verá obligado a heredar de un tipo de excepción existente, preferentemente uno cercano al significado de su nueva excepción (sin embargo, a menudo esto no es posible). La forma más trivial de crear un nuevo tipo de excepción es simplemente dejar que el compilador cree el constructor por defecto, de forma que prácticamente no haya que escribir ningún código:

```
//: c10:DemoExcepcionSencilla.java
// Heredando sus propias excepciones.
class ExcepcionSencilla extends Exception {}

public class DemoExcepcionSencillaDemo {
  public void f() throws ExcepcionSencilla {
    System.out.println(
      "Lanzando ExcepcionSencilla desde f()");
    throw new ExcepcionSencilla ();
  }
  public static void main(String[] args) {
    DemoExcepcionSencilla sed =
      new DemoExcepcionSencilla();
    try {
      sed.f();
    } catch(ExcepcionSencilla e) {
      System.err.println("¡Capturada!");
    }
  }
} ///:~
```

Cuando el compilador crea el constructor por defecto, se trata del que llama automáticamente (y de forma invisible) al constructor por defecto de la clase base. Por supuesto, en este caso no se obtendrá un constructor **ExcepcionSencilla(String)**, pero en la práctica esto no se usa mucho. Como se verá, lo más importante de una excepción es el nombre de la clase, por lo que en la mayoría de ocasiones una excepción como la mostrada arriba es plenamente satisfactoria.

Aquí, se imprime el resultado en la consola de *error estándar* escribiendo en **System.err**. Éste suele ser el mejor sitio para enviar información de error, en vez de **System.out**, que podría estar redirigida. Si se envía la salida a **System.err** no estará redireccionada junto con **System.out**, por lo que el usuario tiene más probabilidades de enterarse.

Crear una clase excepción que tenga también un constructor que tome un **String** como parámetro es bastante sencillo:

```
//: c10:ConstructoresCompletos.java
// Heredando tus propias excepciones.

class MiExcepcion extends Exception {
  public MiExcepcion() {}
  public MiExcepcion(String msg) {
    super(msg);
  }
}

public class ConstructoresCompletos {
  public static void f() throws MiExcepcion {
```

```
    System.out.println(
      "Lanzando MiExcepcion desde f()");
    throw new MiExcepcion();
  }
  public static void g() throws MiExcepcion {
    System.out.println(
      "Lanzando MiExcepcion desde g()");
    throw new MiExcepcion("Originada en g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MiExcepcion e) {
      e.printStackTrace(System.err);
    }
    try {
      g();
    } catch(MiExcepcion e) {
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

El código añadido es poco —la inserción de dos constructores que definen la forma de crear **MiExcepcion**. En el segundo constructor, se invoca explícitamente al constructor de la clase base con un parámetro **String** utilizando la palabra clave **super**.

Se envía a **System.err** información de seguimiento de la pila, de forma que habrá más probabilidades de que se haga notar en caso de que se haya redireccionado **System.out**.

La salida del programa es:

```
Lanzando MiExcepcion desde f()
MiExcepcion
  at FullConstructors.f(FullConstructors.java:16)
  at FullConstructors.main(FullConstructors.java:24)
Lanzando MiExcepcion desde g()
MiExcepcion: originada en g()
  at FullConstructors.g(FullConstructors.java:20)
  at FullConstructors.main(FullConstructors.java:29)
```

Se puede ver la ausencia del mensaje de detalle en la **MiExcepcion** lanzada desde **f( )**.

Se puede llevar aún más lejos el proceso de creación de nuevas excepciones. Se pueden añadir constructores y miembros extra:

```
//: c10:CaracteristicasExtra.java
```

```
// Embellecimiento aún mayor de las clases excepción.

class MiExcepcion2 extends Exception {
  public MiExcepcion2() {}
  public MiExcepcion2(String msg) {
    super(msg);
  }
  public MiExcepcion2(String msg, int x) {
    super(msg);
    i = x;
  }
  public int val() { return i; }
  private int i;
}

public class CaracteristicasExtra {
  public static void f() throws MiExcepcion2 {
    System.out.println(
      "Lanzando MiExcepcion2 desde f()");
    throw new MiExcepcion2();
  }
  public static void g() throws MiExcepcion2 {
    System.out.println(
      "Lanzando MiExcepcion2 desde g()");
    throw new MiExcepcion2("Originada en g()");
  }
  public static void h() throws MiExcepcion2 {
    System.out.println(
      "Lanzando MiExcepcion2 desde h()");
    throw new MiExcepcion2(
      "Originada en h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MiExcepcion2 e) {
      e.printStackTrace(System.err);
    }
    try {
      g();
    } catch(MiExcepcion2 e) {
      e.printStackTrace(System.err);
    }
    try {
      h();
```

```
      } catch(MiExcepcion2 e) {
        e.printStackTrace(System.err);
        System.err.println("e.val() = " + e.val());
      }
    }
} ///:~
```

Se ha añadido un dato miembro **i**, junto con un método que lee ese valor y un constructor adicional. La salida es:

```
Lanzando MiExcepcion2 desde f()
MiExcepcion2
  at CaracteristicasExtra.f(CaracteristicasExtra.java:22)
  at CaracteristicasExtra.main(CaracteristicasExtra.java:34)
Lanzando MyException2 desde g()
MyException2: Originada en g()
  at CaracteristicasExtra.g(CaracteristicasExtra.java:26)
  at CaracteristicasExtra.main(CaracteristicasExtra.java:39)
Lanzando MyException2 desde h()
MyException2: Originada en h()
  at CaracteristicasExtra.h(CaracteristicasExtra.java:30)
  at CaracteristicasExtra.main(CaracteristicasExtra.java:44)
e.val() = 47
```

Dado que una excepción es simplemente otro tipo de objeto, se puede continuar este proceso de embellecimiento del poder de las clases **excepción**. Hay que tener en cuenta, sin embargo, que todo este disfraz podría perderse en los programadores clientes que hagan uso de los paquetes, puesto que puede que éstos simplemente busquen el lanzamiento de la excepción, sin importarles nada más. (Ésta es la forma en que se usan la mayoría de las excepciones de biblioteca de Java.)

# La especificación de excepciones

En Java, se pide que se informe al programador cliente, que llama al método, de las excepciones que podría lanzar ese método. Esto es bastante lógico porque el llamador podrá saber exactamente el código que debe escribir si desea capturar todas las excepciones potenciales. Por supuesto, si está disponible el código fuente, el programador cliente podría simplemente buscar sentencias **throw**, pero a menudo las bibliotecas no vienen con sus fuentes. Para evitar que esto sea un problema, Java proporciona una sintaxis (y *fuerza* el uso de la misma) para permitir decir educadamente al programador cliente qué excepciones lanza ese método, de forma que el programador cliente pueda manejarlas. Ésta es la *especificación de excepciones*, y es parte de la declaración del método, y se sitúa justo después de la lista de parámetros.

La especificación de excepciones utiliza la palabra clave **throws**, seguida de la lista de todos los tipos de excepción potenciales. Por tanto, la definición de un método podría tener la siguiente apariencia:

```
void f() throws DemasiadoGrande, DemasiadoPequeño, DivPorCero { // ...
```

Si se dice

```
void f() { // ...
```

significa que el método no lanza excepciones. (*Excepto* las excepciones de tipo **RuntimeException**, que puede ser lanzado razonablemente desde cualquier sitio —como se describirá más adelante.)

No se puede engañar sobre una especificación de excepciones —si un método provoca excepciones y no las maneja, el compilador lo detectará e indicará que, o bien hay que manejar la excepción o bien hay que indicar en la especificación de excepciones todas las excepciones que el método puede lanzar. Al fortalecer las especificaciones de excepciones de arriba abajo, Java garantiza que se puede asegurar la corrección de la excepción en *tiempo de compilación*[2].

Sólo hay un lugar en el que se pude engañar: se puede decir que se lanza una excepción que verdaderamente no se lanza. El compilador cree en tu palabra, y fuerza a los usuarios del método a tratarlo como si verdaderamente arrojara la excepción. Esto tiene un efecto beneficioso al ser un objeto preparado para esa excepción, de forma que, de hecho, se puede empezar a lanzar la excepción más tarde sin que esto requiera modificar el código ya existente. También es importante para la creación de clases base **abstractas** e **interfaces** cuyas clases derivadas o implementaciones pueden necesitar lanzar excepciones.

# Capturar cualquier excepción

Es posible crear un manejador que capture cualquier tipo de excepción. Esto se hace capturando la excepción de clase base **Exception** (hay otros tipos de excepciones base, pero **Exception** es la clase base a utilizar pertinentemente en todas las actividades de programación):

```
catch(Exception e) {
  System.err.println("Excepcion capturada");
}
```

Esto capturará cualquier excepción, de forma que si se usa, habrá que ponerlo al *final* de la lista de manejadores para evitar que los manejadores de excepciones que puedan venir después queden ignorados.

Dado que la clase **Exception** es la base de todas las clases de excepción que son importantes para el programador, no se logra mucha información específica sobre la excepción, pero se puede llamar a los métodos que vienen de *su* tipo base **Throwable**:

**String getMessage( )**
**String getLocalizedMessage( )**
Toma el mensaje de detalle, o un mensaje ajustado a este escenario particular.

---

[2] Esto constituye una mejora significativa frente al manejo de excepciones de C++, que no captura posibles violaciones de especificaciones de excepciones hasta tiempo de ejecución, donde no es ya muy útil.

**String toString( )**
Devuelve una breve descripción del objeto *Throwable*, incluyendo el mensaje de detalle si es que lo hay.

**void printStackTrace( )**
**void printStackTrace(PrintStream)**
**void printStackTrace(PrintWriter)**
Imprime el objeto y la traza de pila de llamadas lanzada. La pila de llamadas muestra la secuencia de llamadas al método que condujeron al momento en que se lanzó la excepción. La primera versión imprime en el error estándar, la segunda y la tercera apuntan a un flujo de datos de tu elección (en el Capítulo 11, se entenderá por qué hay dos tipos de flujo de datos).

**Throwable fillInStackTrace( )**
Registra información dentro de este objeto **Throwable**, relativa al estado actual de las pilas. Es útil cuando una aplicación está relanzando un error o una excepción (en breve se contará algo más al respecto).

Además, se pueden conseguir otros métodos del tipo base de **Throwable**, **Object** (que es el tipo base de todos). El que podría venir al dedillo para excepciones es **getClass( )** que devuelve un objeto que representa la clase de este objeto. Se puede también preguntar al objeto de esta **Clase** por su nombre haciendo uso de **getName( )** o **toString( )**. Asimismo se pueden hacer cosas más sofisticadas con objetos **Class** que no son necesarios en el manejo de excepciones. Los objetos **Class** se estudiarán más adelante.

He aquí un ejemplo que muestra el uso de los métodos básicos de **Exception**:

```
//: c10:MetodosDeExcepcion.java
// Demostrando los métodos Exception.

public class MetodosDeExcepcion {
  public static void main(String[] args) {
    try {
      throw new Exception("Aquí esta mi excepcion");
    } catch(Exception e) {
      System.err.println("Excepcion capturada");
      System.err.println(
        "e.getMessage(): " + e.getMessage());
      System.err.println(
        "e.getLocalizedMessage(): " +
         e.getLocalizedMessage());
      System.err.println("e.toString(): " + e);
      System.err.println("e.printStackTrace():");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

La salida de este programa es:

```
Excepcion capturada
e.getMessage(): Aquí esta mi excepcion
e.getLocalizedMessage(): Aquí esta mi excepcion
e.toString(): java.lang.Exception:
  Aquí esta mi excepcion
e.printStackTrace():
java.lang.Exception: Aquí esta mi excepcion
 at ExceptionMethods.main(MetodosDeExcepcion.java:7)
java.lang.Exception:
  Aquí esta mi excepcion
 at MetodosDeExcepcion.main(Metodos de Excepcion.java:7)
```

Se puede ver que los métodos proporcionan más información exitosamente —cada una es efectivamente, un superconjunto de la anterior.

# Relanzar una excepción

En ocasiones se desea volver a lanzar una excepción que se acaba de capturar, especialmente cuando se usa **Exception** para capturar cualquier excepción. Dado que ya se tiene la referencia a la excepción actual, se puede volver a lanzar esa referencia:

```
catch(Exception e) {
  System.err.println("Se ha lanzado una excepcion");
  throw e;
}
```

Volver a lanzar una excepción hace que la excepción vaya al contexto inmediatamente más alto de manejadores de excepciones. Cualquier cláusula **catch** subsiguiente del mismo bloque **try** seguirá siendo ignorada. Además, se preserva todo lo relativo al objeto, de forma que el contexto superior que captura el tipo de excepción específico pueda extraer toda la información de ese objeto.

Si simplemente se vuelve a lanzar la excepción actual, la información que se imprime sobre esa excepción en **printStackTrace( )** estará relacionada con el origen de la excepción, no al lugar en el que se volvió a lanzar. Si se desea instalar nueva información de seguimiento de la pila, se puede lograr mediante **fillInStackTrace( )**, que devuelve un objeto excepción creado rellenando la información de la pila actual en el antiguo objeto excepción. Éste es su aspecto:

```
//: c10:Relanzando.java
// Demostrando fillInStackTrace()

public class Relanzando {
  public static void f() throws Exception {
    System.out.println(
      "originando la excepcion en f()");
```

```
    throw new Exception("lanzada desde f()");
  }
  public static void g() throws Throwable {
    try {
      f();
    } catch(Exception e) {
      System.err.println(
        "Dentro de g(), e.printStackTrace()");
      e.printStackTrace(System.err);
      throw e; // 17
      // throw e.fillInStackTrace(); // 18
    }
  }
  public static void
  main(String[] args) throws Throwable {
    try {
      g();
    } catch(Exception e) {
      System.err.println(
        "Capturada en main, e.printStackTrace()");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

Los números de línea importantes están marcados como comentarios con la línea 17 sin comentarios (como se muestra), la salida sería:

```
originando la excepcion en f()
Dentro de g(), e.printStackTrace()
java.lang.Exception: lanzada desde f()
  at Relanzando.f(Relanzando.java:8)
  at Relanzando.g(Relanzando.java:12)
  at Relanzando.main(Relanzando.java:24)
Capturada en el main, e.printStackTrace()
java.lang.Exception: arrojada desde f()
  at Relanzando.g(Relanzando.java:18)
  at Relanzando.g(Relanzando.java:12)
  at Relanzando.main(Relanzando.java:24)
```

De forma que la traza de la pila de excepciones siempre recuerda su punto de origen verdadero, sin que importe cuántas veces se relanza.

Considerando que la línea 17 sea comentario, y no lo sea la línea 18, se usa **fillInStackTrace( )**, siendo el resultado:

```
originando la excepcion en f()
```

```
Dentro de g(), e.printStackTrace()
java.lang.Exception: lanzada desde f()
  at Relanzando.f(Relanzando.java:8)
  at Relanzando.g(Relanzando.java:12)
  at Relanzando.main(Relanzando.java:24)
Capturada en el main, e.printStackTrace()
java.lang.Exception: arrojada desde f()
  at Relanzando.g(Relanzando.java:18)
  at Relanzando.main(Relanzando.java:24)
```

Debido a **fillInStackTrace( )**, la línea 18 se convierte en el nuevo punto de origen de la excepción.

La clase **Throwable** debe aparecer en la especificación de excepciones de **g( )** y **main( )** porque **fillInStackTrace( )** produce una referencia a un objeto **Throwable**. Dado que **Throwable** es una clase base de **Exception**, es posible conseguir un objeto que es un **Throwable** pero *no* un **Exception**, de forma que el manejador para **Exception** en el método **main( )** podría perderla. Para asegurarse de que todo esté en orden, el compilador fuerza una especificación de excepciones que incluya **Throwable**. Por ejemplo, la excepción del siguiente programa *no* se captura en el método **main( )**:

```
//: c10:LanzarFuera.java
public class LanzarFuera {
  public static void
  main(String[] args) throws Throwable {
    try {
      throw new Throwable();
    } catch(Exception e) {
      System.err.println("Capturada en main()");
    }
  }
} ///:~
```

También es posible volver a lanzar una excepción diferente de la capturada. Si se hace esto, se consigue un efecto similar al usar **fillInStackTrace( )** —la información sobre el origen primero de la excepción se pierde, y lo que queda es la información relativa al siguiente **throw**:

```
//: c10:RelanzarNueva.java
// Relanzar un objeto distinto
// del capturado.

class ExcepcionUna extends Exception {
  public ExcepcionUna(String s) { super(s); }
}

class ExcepcionDos extends Exception {
  public ExcepcionDos (String s) { super(s); }
}
```

```
public class RelanzaNuevo {
  public static void f() throws ExcepcionUna {
    System.out.println(
      "originando la excepcion en f()");
    throw new ExcepcionUna("lanzada desde f()");
  }
  public static void main(String[] args)
  throws ExcepcionDos {
    try {
      f();
    } catch(ExcepcionUna e) {
      System.err.println(
        "Capturada en el método main, e.printStackTrace()");
      e.printStackTrace(System.err);
      throw new Excepcion2("desde la main()");
    }
  }
} ///:~
```

La salida es:

```
originando la excepcion en f()
Capturada en la main, e.printStackTrace()
OneException: lanzada desde f()
  at RelanzamientoNuevo.f(RelanzamientoNuevo.java:17)
  at RelanzamientoNuevo.main(RelanzamientoNuevo.java.22)
Excepcion in thread "main" ExcepcionDos: desde la main()
  at RelanzamientoNuevo.main(Rethrow.java:27)
```

La excepción final sólo sabe que proviene de **main( )**, y no de **f( )**.

No hay que preocuparse nunca de limpiar la excepción previa, o cualquier otra excepción en este sentido. Todas son objetos basados en el montículo creados con **new**, por lo que el recolector de basura los limpia automáticamente.

# Excepciones estándar de Java

La clase **Throwable** de Java describe todo aquello que se pueda lanzar en forma de excepción. Hay dos tipos generales de objetos **Throwable** ("tipos de" = "heredados de"). **Error** representa los errores de sistema y de tiempo de compilación que uno no se preocupa de capturar (excepto en casos especiales). **Exception** es el tipo básico que puede lanzarse desde cualquier método de las clases de la biblioteca estándar de Java y desde los métodos que uno elabore, además de incidencias en tiempo de ejecución. Por tanto, el tipo base que más interesa al programador es **Exception**.

La mejor manera de repasar las excepciones es navegar por la documentación HTML de Java que se puede descargar de *java.sun.com*. Merece la pena hacer esto simplemente para tomar un contacto inicial con las diversas excepciones, aunque pronto se verá que no hay nada especial entre las distintas excepciones, exceptuando el nombre. Además, Java cada vez tiene más excepciones; básicamente no tiene sentido imprimirlas en un libro. Cualquier biblioteca nueva que se obtenga de un tercero probablemente tendrá también sus propias excepciones. Lo que es importante entender es el concepto y qué es lo que se debe hacer con las excepciones.

La idea básica es que el nombre de la excepción representa el problema que ha sucedido; de hecho se pretende que el nombre de las excepciones sea autoexplicativo. Las excepciones no están todas ellas definidas en **java.lang**; algunas están creadas para dar soporte a otras bibliotecas como **util**, **net** e **io**. Así, por ejemplo, todas las excepciones de E/S se heredan de **java.io.IOException**.

# El caso especial de **RuntimeException**

El primer ejemplo de este capítulo fue:

```
if (t == null)
  throw new NullPointerException();
```

Puede ser bastante horroroso pensar que hay que comprobar que cualquier referencia que se pase a un método sea **null** o no **null** (de hecho, no se puede saber si el que llama pasa una referencia válida). Afortunadamente, no hay que hacerlo —esto es parte de las comprobaciones estándares de tiempo de ejecución que hace Java, de forma que si se hiciera una llamada a una referencia **null**, Java lanzaría automáticamente una **NullPointerException**. Por tanto, el fragmento de código de arriba es totalmente superfluo.

Hay un grupo completo de tipos de excepciones en esta categoría. Se trata de excepciones que Java siempre lanza automáticamente, y no hay que incluirlas en las especificaciones de excepciones. Además, están convenientemente agrupadas juntas bajo una única clase base denominada **RuntimeException**, que es un ejemplo perfecto de herencia: establece una familia de tipos que tienen algunas características y comportamientos en común. Tampoco se escribe nunca una especificación de excepciones diciendo que un método podría lanzar una **RuntimeException**, puesto que se asume. Dado que indican fallos, generalmente una **RuntimeException** nunca se captura —se maneja automáticamente. Si uno se viera forzado a comprobar las excepciones de tipo **RuntimeException**, su código se volvería farragoso. Incluso aunque generalmente las **RuntimeExceptions** no se capturan, en los paquetes que uno construya se podría decidir lanzar algunas **RuntimeExceptions**.

¿Qué ocurre si estas excepciones no se capturan? Dado que el compilador no fortalece las especificaciones de excepciones en este caso, es bastante verosímil que una **RuntimeException** pudiera filtrar todo el camino hacia el exterior hasta el método **main( )** sin ser capturada. Para ver qué ocurre en este caso, puede probarse el siguiente ejemplo:

```
//: c10:NuncaCapturado.java
// Ignorando RuntimeExceptions.
```

```
public class NuncaCapturado {
  static void f() {
    throw new RuntimeException("Desde f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
  }
} ///:~
```

Ya se puede ver que una **RuntimeException** (y cualquier cosa que se herede de la misma) es un caso especial, puesto que el compilador no exige una especificación de excepciones para ellas.

La salida es:

```
Exception in thread "main"
java.lang.RuntimeException: Desde f()
  at NuncaCapturado.f(NuncaCapturado.java:9)
  at NuncaCapturado.g(NuncaCapturado.java:12)
  at NuncaCapturado.main(NuncaCapturado.java:15)
```

Por tanto la respuesta es: si una **excepción en tiempo de ejecución** consigue todo el camino hasta el método **main( )** sin ser capturada, se invoca a **printStackTrace( )** para esa excepción, y el programa finaliza su ejecución.

Debe tenerse en cuenta que sólo se pueden ignorar en un código propio las **excepciones en tiempo de ejecución**, puesto que el compilador obliga a realizar el resto de gestiones. El razonamiento es que una **excepción en tiempo de ejecución** representa un error de programación:

1. Un error que no se puede capturar (la recepción de una referencia **null** proveniente de un programador cliente por parte de un método, por ejemplo).

2. Un error que uno, como programador, debería haber comprobado en su código (como un **ArrayIndexOutOfBoundsException** en la que se debería haber comprobado el tamaño del array).

Se puede ver fácilmente el gran beneficio aportado por estas excepciones, puesto que éstas ayudan en el proceso de depuración.

Es interesante saber que no se puede clasificar el manejo de excepciones de Java como si fuera una herramienta de propósito específico. Aunque efectivamente está diseñado para manejar estos errores de tiempo de compilación que se darán por motivos externos al propio código, simplemente es esencial para determinados tipos de fallos de programación que el compilador no pueda detectar.

# Limpiando con finally

A menudo hay algunos fragmentos de código que se desea ejecutar independientemente de que se lancen o no excepciones dentro de un bloque **try**. Esto generalmente está relacionado con operaciones distintas de recuperación de memoria (puesto que de esto ya se encarga el recolector de basura). Para lograr este efecto, se usa una cláusula **finally**[3] al final de todos los manejadores de excepciones. El esquema completo de una sección de manejo de excepciones es por consiguiente:

```
try {
   // La region protegida: Actividades peligrosas que
   // podrían lanzar A, B o C
} catch (A a1) {
   // Manejador para la situación A
} catch (B b1) {
   // Manejador para la situación B
} catch (C c1) {
   // Manejador para la situación C
} finally {
   // Actividades que se dan siempre
}
```

Para demostrar que la cláusula **finally** siempre se ejecuta, puede probarse el siguiente programa:

```
//: c10:EjecucionFinally.java
// La clausula finally se ejecuta siempre.

class ExcepcionTres extends Exception {}

public class EjecucionFinally {
   static int contador = 0;
   public static void main(String[] args) {
      while(true) {
         try {
            // El post-incremento es cero la primera vez:
            if(contador++ == 0)
               throw new ExcepcionTres();
            System.out.println("Sin excepcion");
         } catch(ExcepcionTres e) {
            System.err.println("ExcepcionTres");
         } finally {
            System.err.println("Inicio de clausula finally");
            if(contador == 2) break; // salida del "while"
         }
```

[3] El manejo de excepciones de C++ no tiene la cláusula **finally** porque confía en los destructores para lograr este tipo de limpieza.

```
    }
  }
} ///:~
```

Este programa también da una orientación para manejar el hecho de que las excepciones de Java (al igual que ocurre en C++) no permiten volver a ejecutar a partir del punto en que se lanzó la excepción, como ya se comentó. Si se ubica el bloque **try** dentro de un bloque, se podría establecer una condición a alcanzar antes de continuar con el programa. También se puede añadir un contador **estático** o algún otro dispositivo para permitir al bucle intentar distintos enfoques antes de rendirse. De esta forma se pueden construir programas de extremada fortaleza.

La salida es:

```
ExcepcionTres
Inicio de clausula finally
Sin excepcion
Inicio de clausula finally
```

Se lance o no una excepción, la cláusula **finally** se ejecuta siempre.

# ¿Para qué sirve **finally**?

En un lenguaje en el que no haya recolector de basura *y* sin llamadas automáticas a destructores[4], **finally** es importante porque permite al programador garantizar la liberación de memoria independientemente de lo que ocurra en el bloque **try**. Pero Java tiene recolección de basura, por lo que la liberación de memoria no suele ser un problema. Además, no tiene destructores a los que invocar. Así que, ¿cuándo es necesario usar **finally** en Java?

La cláusula **finally** es necesaria cuando hay que hacer algo *más* que devolver la memoria a su estado original. Éste es un tipo de limpieza equivalente a la apertura de un fichero o de una conexión de red, algo que seguro se habrá manejado más de una vez, y que se modela en el ejemplo siguiente:

```
//: c10:EncenderApagarInterrumpir.java
// ¿Por que usar finally?

class Interrumpir {
  boolean estado = false;
  boolean leer() { return estado; }
  void encender() { estado = true; }
  void apagar() { estado = false; }
}
class ExcepcionEncenderApagar1 extends Exception {}
class ExcepcionEncenderApagar2 extends Exception {}
```

[4] Un destructor es una función a la que se llama siempre que se deja de usar un objeto. Siempre se sabe exactamente cuándo y dónde llamar al destructor. C++ tiene llamadas automáticas a destructores, pero las versiones 1 y 2 del Objeto Pascal de Delphi no (lo que cambia el significado y el uso del concepto de destructor en estos lenguajes).

```
public class EncenderApagarInterruptor {
   static Interruptor sw = new Interruptor();
   static void f() throws
     ExcepcionEncenderApagar1, ExcepcionEncenderApagar2 {}
   public static void main(String[] args) {
     try {
       sw.encender();
       // Código que puede lanzar excepciones...
       f();
       sw.apagar();
     } catch(OnOffException1 e) {
       System.err.println("OnOffException1");
       sw.apagar();
     } catch(ExcepcionEncenderApagar2 e) {
       System.err.println("ExcepcionEncenderApagar2");
       sw.apagar();
     }
   }
} ///:~
```

Aquí el objetivo es asegurarse de que el interruptor esté apagado cuando se complete el método **main( )**, por lo que **sw.apagar( )** se coloca al final del bloque try y al final de cada manejador de excepción. Pero es posible que se pudiera lanzar una excepción no capturada aquí, por lo que se perdería el **sw.apagar( )**. Sin embargo, con **finally**, se puede colocar código de limpieza simplemente en un lugar:

```
//: c10:ConFinally.java
// Finally garantiza la limpieza.

public class ConFinally {
   static Interruptor sw = new Interruptor();
   public static void main(String[] args) {
     try {
       sw.encender();
       // Código que puede lanzar excepciones...
       EncenderApagarInterruptor.f();
     } catch(ExcepcionEncenderApagar1 e) {
       System.err.println("OnOffException1");
     } catch(ExcepcionEncenderApagar2 e) {
       System.err.println("ExcepcionEncenderApagar2");
     } finally {
       sw.apagar();
     }
   }
} ///:~
```

Aquí se ha movido el **sw.apagar( )** simplemente un lugar, en el que se garantiza su ejecución independientemente de lo que pase.

Incluso en las clases en las que la excepción no se capture en el conjunto de cláusulas **catch**, se ejecutará **finally** antes de que el mecanismo de manejo de excepciones continúe su búsqueda de un manejador de nivel superior:

```
//: c10:SiempreFinally.java
// Finally se ejecuta siempre.

class ExceptionCuatro extends Exception {}

public class SiempreFinally {
  public static void main(String[] args) {
    System.out.println(
      "Entrando en el primer bloque try");
    try {
      System.out.println(
        "Entrando en el segundo bloque try");
      try {
        throw ExcepcionCuatro();
      } finally {
        System.out.println(
          "finally en el segundo bloque try");
      }
    } catch(ExcepcionCuatro e) {
      System.err.println(
        "Capturada ExcepcionCuatro en el primer bloque try");
    } finally {
      System.err.println(
        "finally en el primer bloque try");
    }
  }
} ///:~
```

La salida de este programa muestra lo que ocurre:

```
Entrando en el primer bloque try
Entrando en el segundo bloque try
finally en el segundo bloque try
Capturada ExcepcionCuatro en el primer bloque try
finally en el primer bloque try
```

La sentencia **finally** también se ejecutará en aquellos casos en que se vean involucradas sentencias **break** y **continue**. Fíjese que, con las sentencias **break** y **continue**, **finally** elimina la necesidad de una sentencia **goto** en Java.

# Peligro: la excepción perdida

En general, la implementación de las excepciones en Java destaca bastante, pero desgraciadamente tiene un problema. Aunque las excepciones son una indicación de una crisis en un programa, y nunca debería ignorarse, es posible que simplemente se pierda una excepción. Esto ocurre con una configuración particular al hacer uso de la cláusula **finally**:

```
//: c10:MensajePerdido.java
// Cómo puede perderse una excepcion.

class ExceptionMuyImportante extends Exception {
  public String toString() {
    return "¡Una excepcion muy importante!";
  }
}

class ExcepcionTrival extends Exception {
  public String toString() {
    return "Una excepcion trivial";
  }
}

public class MensajePerdido {
  void f() throws ExcepcionMuyImportante {
    throw new ExcepcionMuyImportante();
  }
  void disponer() throws ExcepcionTrivial {
    throw new ExcepcionTrivial();
  }
  public static void main(String[] args)
      throws Exception {
    MensajePerdido lm = new MensajePerdido();
    try {
      lm.f();
    } finally {
      lm.disponer();
    }
  }
} ///:~
```

La salida es:

```
Exception in thread "main" Una excepcion trivial
  at MensajePerdido.disponer(MensajePerdido.java:21)
  at MensajePerdido.main(MensajePerdido.java:29)
```

Se puede ver que no hay evidencia de la **ExcepcionMuyImportante**, que simplemente es reemplazada por la **ExcepcionTrivial** en la cláusula **finally**. Ésta es una trampa bastante seria, puesto que significa que una excepción podría perderse completamente, incluso de forma más oculta y difícil de detectar que en el ejemplo de arriba. Por el contrario, C++ trata la situación en la que se lanza una segunda excepción antes de que se maneje la primera como un error de programación fatal. Quizás alguna versión futura de Java repare este problema (por otro lado, generalmente se envuelve todo método que lance alguna excepción, tal como **dispose( )** dentro de una cláusula **try-catch)**.

# Restricciones a las excepciones

Cuando se superpone un método, sólo se pueden lanzar las excepciones que se hayan especificado en la versión de la clase base del método. Ésta es una restricción útil, pues significa que todo código que funcione con la clase base funcionará automáticamente con cualquier objeto derivado de la clase base (un concepto fundamental en POO, por supuesto), incluyendo las excepciones.

Este ejemplo demuestra los tipos de restricciones impuestas (en tiempo de compilación) a las excepciones:

```
//: c10:CarreraTormentosa.java
// Los métodos superpuestos sólo pueden lanzar las
// excepciones especificadas en su versión clase base,
// o excepciones derivadas de las excepciones de la
// clase base.

class ExcepcionBeisbol extends Exception {}
class Falta extends ExcepcionBeisbol {}
class Strike extends ExcepcionBeisbol {}

abstract class Carrera {
  Carrera() throws ExcepcionBeisbol {}
  void evento () throws ExcepcionBeisbol {
   // De hecho no tiene que lanzar nada
  }
  abstract void batear() throws Strike, Foul;
  void caminar() {} // No lanza nada
}

class ExcepcionTormenta extends Exception {}
class Llueve extends ExceptionTormenta {}
class Eliminacion extends Falta {}

interface Tormenta {
  void evento() throws Llueve;
  void lloverFuerte() throws Llueve;
}
```

```
public class CarreraTormentosa extends Carrera
    implements Tormenta {
  // Ok para añadir nuevas excepciones a
  // los constructores, pero hay que hacerlo
  // con las excepciones del constructor base:
  CarreraTormentosa() throws Llueve,
    ExcepcionBeisbol {}
  CarreraTormentosa(String s) throws Falta,
   Excepcion Beisbol {}
  // Los métodos normales deben estar conformes con
  // la clase base:
//! void caminar() throws eliminación {} //Error de compilación
  // Una Interfaz NO PUEDE añadir excepciones a los
  // métodos existentes de la clase base:
//! public void evento() throws Llueve {}
  // Si el método no existe aún en la clase base,
  // OK con la excepción:
  public void lloverFuerte() throws Llueve {}
  // Se puede elegir no lanzar excepciones,
  // incluso si lo hace la versión base:
  public void evento() {}
  // Los métodos superpuestos pueden lanzar
  // excepciones heredadas:
  void batear() throws Eliminacion {}
  public static void main(String[] args) {
    try {
      CarreraTormentosa si = new CarreraTormentosa();
      si.batear();
    } catch(Eliminacion e) {
      System.err.println("Eliminacion");
    } catch(Llueve e) {
      System.err.println("Llueve");
    } catch(ExcepcionBeisbol e) {
      System.err.println("Error generico");
    }
    // Strike no se lanza en la versión derivada.
    try {
      // ¿Qué ocurre si se hace una conversión hacia arriba?
      Carrera i = new CarreraTormentosa();
      i.batear();
      // Hay que capturar las excepciones desde
      // la versión clase base del método:
    } catch(Strike e) {
      System.err.println("Strike");
    } catch(Falta e) {
```

```
        System.err.println("Falta");
      } catch(Llueve e) {
        System.err.println("Llueve");
      } catch(ExcepcionBeisbol e) {
        System.err.println(
          "Excepcion beisbol generica");
      }
    }
} ///:~
```

En **Carrera**, se puede ver que tanto el método **evento( )** como el constructor dicen que lanzarán una excepción, pero nunca lo hacen. Esto es legal porque permite forzar al usuario a capturar cualquier excepción que se pueda añadir a versiones superpuestas de **evento( )**. Como se ve en **batear( )**, en el caso de métodos **abstractos** se mantiene la misma idea.

La **interfaz Tormenta** es interesante porque contiene un método (**evento( )**) que está definido en **Carrera**, y un método que no lo está. Ambos métodos lanzan un nuevo tipo de excepción, **llueve**. Cuando **CarreraTormentosa hereda de carrera** e **implementa Tormenta**, se verá que el método **evento( )** de **Tormenta** *no puede* cambiar la interfaz de excepciones de **evento( )** en **Carrera**. De nuevo, esto tiene sentido porque de otra forma nunca se sabría si se está capturando lo correcto al funcionar con la clase base. Por supuesto, si un método descrito en una **interfaz** no está en la clase base, como ocurre con **lloverFuerte( )**, entonces no hay problema si lanza excepciones.

La restricción sobre las excepciones no se aplica a los constructores. En **CarreraTormentosa** se puede ver que un constructor puede lanzar lo que desee, independientemente de lo que lance el constructor de la clase base. Sin embargo, dado que siempre se llamará de una manera u otra a un constructor de clase base (aquí se llama automáticamente al constructor por defecto), el constructor de la clase derivada debe declarar cualquier excepción del constructor de la clase base en su especificación de excepciones. Fíjese que un constructor de clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.

La razón por la que **CarreraTormentosa.caminar( )** no compilará es que lanza una excepción, mientras que **Carrera.caminar( )** no lo hace. Si se permitiera esto, se podría escribir código que llamara a **Carrera.caminar( )** y que no tuviera que manejar ninguna excepción, pero entonces, al sustituir un objeto de una clase derivada de **Carrera** se lanzarían excepciones, causando una ruptura del código. Forzando a los métodos de la clase derivada a ajustarse a las especificaciones de excepciones de los métodos de la clase base, se mantiene la posibilidad de sustituir objetos.

El método **evento( )** superpuesto muestra que una versión de clase derivada de un método puede elegir no lanzar excepciones, incluso aunque lo haga la versión de clase base. De nuevo, esto es genial, puesto que no rompe ningún código escrito —asumiendo que la versión de clase base lanza excepciones. A **batear( )** se le aplica una lógica semejante, pues ésta lanza **Eliminación**, una excepción derivada de **Falta**, lanzada por la versión de clase base de **batear( )**. De esta forma, si alguien escribe código que funciona con **Carrera** y llama a **batear( )**, debe capturar la excepción **Falta**. Dado que **Eliminación** deriva de **Falta**, el manejador de excepciones también capturará **Eliminación**.

El último punto interesante está en el método **main( )**. Aquí se puede ver que si se está tratando con un objeto **CarreraTormentosa**, el compilador te fuerza a capturar sólo las excepciones específicas a esa clase, pero si se hace un conversión hacia arriba al tipo base, el compilador te fuerza (correctamente) a capturar las excepciones del tipo base. Todas estas limitaciones producen un código de manejo de excepciones más robusto[5].

Es útil darse cuenta de que aunque las especificaciones de excepciones se ven reforzadas por el compilador durante la herencia, las especificaciones de excepciones no son parte del tipo de un método, que está formado sólo por el nombre del método y los tipos de parámetros. Además, justo porque existe una especificación de excepciones en una versión de clase base de un método, no tiene por qué existir en la versión de clase derivada del mismo. Esto es bastante distinto de lo que dictaminan las reglas de herencia, según las cuales todo método de la clase base debe existir también en la clase derivada. Dicho de otra forma, "la interfaz de especificación de excepciones" de un método particular puede estrecharse durante la herencia y superponerse, pero no puede ancharse —esto es precisamente lo contrario de la regla de la interfaz de clases durante la herencia.

# Constructores

Cuando se escribe código con excepciones, es particularmente importante que siempre se pregunte: "Si se da una excepción, ¿será limpiada adecuadamente?" La mayoría de veces es bastante seguro, pero en los constructores hay un problema. El constructor pone el objeto en un estado de partida seguro, pero podría llevar a cabo otra operación —como abrir un fichero— que no se limpia hasta que el usuario haya acabado con el objeto y llame a un método de limpieza especial. Si se lanza una excepción desde dentro de un constructor, puede que estos comportamientos relativos a la limpieza no se den correctamente. Esto significa que hay que ser especialmente cuidadoso al escribir constructores.

Dado que se acaba de aprender lo que ocurre con **finally**, se podría pensar que es la solución correcta. Pero no es tan simple, puesto que **finally** ejecuta *siempre* el código de limpieza, incluso en las situaciones en las que no se desea que se ejecute este código de limpieza hasta que acabe el método de limpieza. Por consiguiente, si se lleva a cabo una limpieza en **finally**, hay que establecer algún tipo de indicador cuando el constructor finaliza normalmente, de forma que si el indicador está activado no se ejecute nada en **finally**. Dado que esto no es especialmente elegante (se está asociando el código de un sitio a otro), es mejor si se intenta evitar llevar a cabo este tipo de limpieza en el método **finally**, a menos que uno se vea forzado a ello.

En el ejemplo siguiente, se crea una clase llamada **ArchivoEntrada** que abre un archivo y permite leer una línea (convertida a **String**) de una vez. Usa las clases **FileReader** y **BufferedReader** de la biblioteca estándar de E/S de Java que se verá en el Capítulo 11, pero que son lo suficientemente simples como para no tener ningún problema en tener su uso básico:

---

[5] ISO C++ añadió limitaciones semejantes que requieren que las excepciones de métodos derivados sean las mismas, o derivadas, de las excepciones que lanza el método de la clase base. Este es un caso en el que C++, de hecho, puede comprobar las especificaciones de excepciones en tiempo de compilación.

```
//: c10:Limpieza.java
// Prestando atención a las excepciones
// en los constructores.
import java.io.*;

class ArchivoEntrada {
  private BufferedReader entrada;
  ArchivoEntrada(String nombref) throws Exception {
    try {
      in =
        new BufferedReader(
          new FileReader(nombref));
      // Otro código que podría lanzar excepciones
    } catch(FileNotFoundException e) {
      System.err.println(
        "No se pudo abrir " + nombref);
      // No se abrió, así que no se cierra
      throw e;
    } catch(Exception e) {
      // Todas las demás excepciones deben cerrarlo
      try {
        entrada.close();
      } catch(IOException e2) {
        System.err.println(
          "entrada.close() sin exito");
      }
      throw e; // Relanzar
    } finally {
      // ¡¡¡No cerrarlo aquí!!!
    }
  }
  String obtenerLinea() {
    String s;
    try {
      s = entrada.readLine();
    } catch(IOException e) {
      System.err.println(
        "obtenerLinea() sin exito");
      s = "fallo";
    }
    return s;
  }
  void limpiar() {
    try {
      entrada.close();
```

```
      } catch(IOException e2) {
        System.err.println(
          "entrada.close() sin exito");
      }
    }
}

public class Limpieza {
  public static void main(String[] args) {
    try {
      ArchivoEntrada entrada =
        new ArchivoEntrada("Limpieza.java");
      String s;
      int i = 1;
      while((s = entrada.obtenerLinea()) != null)
        System.out.println(""+ i++ + ": " + s);
      entrada.limpiar();
    } catch(Exception e) {
      System.err.println(
        "Capturando en el método main, e.printStackTrace()");
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

El constructor de **ArchivoEntrada** toma un parámetro **String**, que es el nombre del archivo que se desea abrir. Dentro de un bloque **try**, crea un **FileReader** usando el nombre de archivo. Un **FileReader** no es particularmente útil hasta que se usa para crear un **BufferedReader** con el que nos podemos comunicar —nótese que uno de los beneficios de **ArchivoEntrada** es que combina estas dos acciones.

Si el constructor **FileReader** no tiene éxito, lanza una **FileNotFoundException** que podría ser capturada de forma separada porque éste es el caso en el que no se quiere cerrar el archivo, puesto que éste no se abrió con éxito. Cualquier *otra* cláusula de captura debe cerrar el archivo, puesto que *fue* abierto en el momento en que se entra en la cláusula **catch**. (Por supuesto, esto es un truco si **FileNotFoundException** puede ser lanzado por más de un método. En este caso, se podría desear romper todo en varios bloques **try**.) El método **close( )** podría lanzar una excepción, por lo que es probado y capturado incluso aunque se encuentra dentro de otro bloque de otra cláusula **catch** — es simplemente otro par de llaves para el compilador de Java. Después de llevar a cabo operaciones locales, se relanza la excepción, lo cual es apropiado porque este constructor falló, y no se desea llamar al método asumiendo que se ha creado el objeto de manera adecuada o que sea válido.

En este ejemplo, que no usa la técnica de los indicadores anteriormente mencionados, la cláusula **finally** *no* es definitivamente el lugar en el que **close( )** (cerrar) el archivo, puesto que lo cerraría cada vez que se complete el constructor. Dado que queremos que se abra el fichero durante la vida útil del objeto **ArchivoEntrada** esto no sería apropiado.

EL método **obtenerLinea( )** devuelve un **String** que contiene la línea siguiente del archivo. Llama a **leerLinea( )**, que puede lanzar una excepción, pero esa excepción se captura de forma que **obtenerLinea( )** no lanza excepciones. Uno de los aspectos de diseño de las excepciones es la decisión de si hay que manejar una excepción completamente en este nivel, o hay que manejarla parcialmente y pasar la misma excepción (u otra), o bien simplemente pasarla. Pasarla, siempre que sea apropiada, puede definitivamente simplificar la codificación. El método **obtenerLinea( )** se convierte en:

```
String obtenerLinea() throws IOException {
  return entrada.readLine();
}
```

Pero por supuesto, el objeto que realiza la llamada es ahora el responsable de manejar cualquier **IOException** que pudiera surgir.

El usuario debe llamar al método **limpiar( )** al acabar de usar el objeto **ArchivoEntrada**. Esto liberará los recursos del sistema (como los manejadores de archivos) que fueron usados por el **BufferedReader** y/u objetos **FileReader**[6]. Esto no se quiere hacer hasta que se acabe con el objeto **InputFile**, en el momento en el que se le deje marchar. Se podría pensar en poner esta funcionalidad en un método **finalize( )**, pero como se mencionó en el Capítulo 4, no se puede estar seguro de que se invoque siempre a **finalize( )** (incluso si se *puede* estar seguro de que se invoque, no se sabe *cuándo*). Éste es uno de los puntos débiles de Java: toda la limpieza —que no sea la limpieza de memoria— no se da automáticamente, por lo que hay que informar al programador cliente de que es responsable, y debe garantizar que se dé la limpieza usando **finalize( )**.

En **Limpieza.java** se crea un **ArchivoEntrada** para abrir el mismo archivo fuente que crea el programa, se lee el archivo de línea en línea, y se añaden números de línea. Se capturan de forma genérica todas las excepciones en el método **main( )**, aunque se podría elegir una granularidad mayor.

Uno de los beneficios de este ejemplo es mostrar por qué se presentan las excepciones en este punto del libro —no se puede hacer E/S básica sin usar las excepciones. Las excepciones son tan integrantes de la programación de Java, especialmente porque el compilador las fortalece, que se puede tener éxito si no se conoce bien cómo trabajar con ellas.

# Emparejamiento de excepciones

Cuando se lanza una excepción, el sistema de manejo de excepciones busca en los manejadores más "cercanos" en el mismo orden en que se escribieron. Cuando hace un emparejamiento, se considera que la excepción ya está manejada y no se llevan a cabo más búsquedas.

Emparejar una excepción no exige un proceso perfecto entre la excepción y su manejador. Un objeto de clase derivada se puede emparejar con un manejador de su clase base, como se ve en el ejemplo siguiente:

---

[6] En C++ un destructor se encargaría de esto por ti.

```
//: c10:Humano.java
// Capturando jerarquías de excepciones.

class Molestia extends Exception {}
class Estornudo extends Molestia {}

public class Humano {
  public static void main(String[] args) {
    try {
      throw new Estornudo();
    } catch(Estornudo s) {
      System.err.println("Estornudo capturado");
    } catch(Molestia a) {
      System.err.println("Molestia capturada");
    }
  }
} ///:~
```

La excepción **Estornudo** será capturada por la primera cláusula **catch** con la que se empareje —que es la primera, por supuesto. Sin embargo, si se anula la primera cláusula **catch** dejando sólo:

```
    try {
      throw new Estornudo();
    } catch(Molestia a) {
      System.err.println("Molestia capturada");
    }
```

El código seguirá funcionando porque está capturando la clase base de **Estornudo**. Dicho de otra forma, **catch(Molestia e)** capturará una **Molestia** o *cualquier otra clase derivada de ésta*. Esto es útil porque si se decide añadir nuevas excepciones derivadas a un método, el código del programador cliente no necesitará ninguna modificación mientras el cliente capture las excepciones de clase base.

Si se intenta "enmascarar" las excepciones de la clase derivada poniendo la cláusula **catch** de la clase base la primera, como en:

```
    try {
      throw new Estornudo();
    } catch(Molestia a) {
      System.err.println("Molestia capturada");
    } catch(Estornudo s) {
      System.err.println("Estornudo capturado");
    }
```

el compilador dará un mensaje de error, puesto que ve que nunca se alcanzará la cláusula catch de **Estornudo**.

## Guías de cara a las excepciones

Utilice excepciones para:

1. Arreglar el problema y llamar de nuevo al método que causó la excepción.
2. Arreglar todo y continuar sin volver a ejecutar el método.
3. Calcular algún resultado alternativo en vez de lo que se suponía que iba a devolver el método.
4. Hacer lo que se pueda en el contexto actual y relanzar la *misma* excepción a un contexto superior.
5. Hacer lo que se pueda en el contexto actual y lanzar una excepción *diferente* a un contexto superior.
6. Terminar el programa.
7. Simplificar. (Si tu esquema de excepción hace algo más complicado, es una molestia utilizarlo.)
8. Hacer más seguros la biblioteca y el programa. (Ésta es una inversión a corto plazo de cara a la depuración, y también una inversión a largo plazo de cara a la fortaleza de la aplicación.)

# Resumen

La recuperación mejorada de errores es una de las formas más poderosas de incrementar la fortaleza del código. La recuperación de errores es fundamental en todos los programas que se escriban, pero es especialmente importante en Java, donde uno de los objetivos principales es crear componentes de programa para que otros los usen. *Para crear un sistema robusto, cada componente debe ser robusto.*

Los objetivos del manejo de excepciones en Java son simplificar la creación de programas grandes y seguros utilizando menos código de lo actualmente disponible, y con garantías de que la aplicación no tenga errores sin manejar.

Las excepciones no son terribles de aprender, y son una de esas características que proporcionan beneficios inmediatos y significativos al proyecto. Afortunadamente, Java fortalece todos los aspectos de las excepciones, por lo que se garantiza que se usarán consistentemente por parte tanto de los diseñadores de bibliotecas como de programadores cliente.

# Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *www.BruceEckel.con*.

1. Crear una clase con un método **main( )** que lance un objeto de tipo **Exception** dentro de un bloque **try**. Dar al constructor de **Exception** un parámetro **String**. Capturar la excepción en una cláusula **catch** e imprimir el parámetro **String**. Añadir una cláusula **finally** e imprimir un mensaje para probar que uno paso por ahí.

2. Crear una clase de excepción usando la palabra clave **extends**. Escribir un constructor para esta clase que tome un parámetro **String** y lo almacene dentro del objeto con una referencia **String**. Escribir un método que imprima el **String** almacenado. Crear una cláusula **try-catch** para probar la nueva excepción.

3. Escribir una clase con un método que lance una excepción del tipo creado en el Ejercicio 2. Intentar compilarla sin especificación de excepción para ver qué dice el compilador. Añadir la especificación apropiada. Probar la clase y su excepción dentro de una cláusula try-catch.

4. Definir una referencia a un objeto e inicializarla a **null**. Intentar llamar a un método mediante esta referencia. Ahora envolver el código en una cláusula **try-catch** para capturar la excepción.

5. Crear una clase con dos métodos, **f( )** y **g( )**. En **g( )**, lanzar una excepción de un nuevo tipo a definir. En **f( )**, invocar a **g( )**, capturar su excepción y, en la cláusula **catch**, lanzar una excepción distinta (de tipo diferente al definido). Probar el código en un método **main( )**.

6. Crear tres nuevos tipos de excepciones. Escribir una clase con un método que lance las tres. En el método **main( )**, invocar al método pero usar sólo una única cláusula **catch** para capturar los tres tipos de excepción.

7. Escribir código para generar y capturar una **ArrayIndexOutOfBoundsException**.

8. Crear tu propio comportamiento reiniciador utilizando un bucle **while** que se repita hasta que se deje de lanzar una excepción.

9. Crear una jerarquía de excepciones de tres niveles. Crear a continuación una clase base **A** con un método que lance una excepción a la base de la jerarquía. Heredar **B** de **A** y superponer el método de forma que lance una excepción en el nivel dos de la jerarquía. Repetir heredando la clase **C** de **B**. En el método **main( )**, crear un objeto de tipo **C** y hacer un conversión hacia arriba a **A**. Después, llamar al método.

10. Demostrar que un constructor de clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.

11. Mostrar que **EncenderApagarInterruptor.java** puede fallar lanzando una **RuntimeException** dentro del bloque **try**.

12. Mostrar que **ConFinally.java** no falla lanzando una **RuntimeException** dentro de un bloque **try**.

13. Modificar el Ejercicio 6 añadiendo una cláusula **finally**. Mostrar que esta cláusula **finally** se ejecuta, incluso si se lanza una **NullPointerException**.

14. Crear un ejemplo en el que se use un indicador para controlar si se llama a código de limpieza, tal y como se comentó en el segundo párrafo del epígrafe "Constructores".

15. Modificar **CarreraTormentosa.java** añadiendo un tipo de excepción **ArgumentoArbitro** y métodos que la lancen. Probar la jerarquía modificada.

16. Eliminar la primera cláusula **catch** de **Humano.java** y verificar que el código se sigue ejecutando y compilando correctamente.

17. Añadir un segundo nivel de pérdida de excepciones a **MensajePerdido.java** de forma que la propia **ExcepcionTrivial** se vea reemplazada por una tercera excepción.

18. En el Capítulo 5, encontrar los dos programas denominados **Afirmacion.java** y modificarlos de forma que lancen su propio tipo de excepción en vez de imprimir a **System.err**. Esta excepción debería estar en una clase interna que extienda **RuntimeException**.

19. Añadir un conjunto apropiado de excepciones a **c08:ControlesInvernadero.java**.