Capítulo 7

Excepciones

¿Qué son las excepciones?

Una excepción es un error excepcional, infrecuente, raro, que ocurre en un porcentaje pequeño de veces en la ejecución de un programa bajo condiciones normales esperadas. Es importante distinguir aquí lo ambiguo de esta definición: ¿A partir de qué porcentaje de frecuencia se le considera a un error infrecuente?, ¿qué condiciones se consideran normales? A lo largo de este capítulo se mostrarán ejemplos comunes de aplicación de excepciones así como algunos criterios de decisión que pueden ayudar a decidir cuándo tratar a un error como una excepción y cuándo no.

Algunos ejemplos típicos de excepciones son:

- Acceso a un elemento de un arreglo fuera de los límites de éste, o en general, a una dirección inválida de memoria.
- Una división por cero.
- Una llamada a un método con valores errados en sus parámetros. Ejemplos: Utilizar un descriptor de archivo aun no inicializado, o ya cerrado, para leer o escribir en un archivo; realizar una conversión de una texto a un número, cuando el texto no contiene una representación válida de un número; pasar una referencia nula de un objeto a una función que espera recibir una referencia válida, etc.
- Una falla en la reserva de memoria (la sentencia *new* falla).

Los errores tratados como excepciones suelen ser también aquellos para los que, dentro del ámbito del programa donde ocurre dicho error, no es posible darle una solución satisfactoria. Note que aquí también hay ambigüedad: ¿Qué es una solución satisfactoria?

Si dicho ámbito abarca a todo el programa, éste por consiguiente no puede continuar ejecutándose de la manera esperada, por lo que debería finalizar. Si en un ámbito que engloba al ámbito de la excepción, se cuenta con los elementos necesarios para darle una solución satisfactoria, el manejo de dicha excepción en el ámbito englobante podría permitir estabilizar el programa, de forma que vuelva a un estado de ejecución normal. Si ningún ámbito englobante, en todo el programa, puede solucionar la excepción, el programa podría notificar al usuario de lo ocurrido y finalizar ordenadamente, liberando los recursos que fueron reservados. En resumen, el tratamiento de excepciones en un programa permite:

- Que el programa se recupere de la excepción y siga ejecutándose normalmente.
- Que el programa notifique la excepción y finalice de manera controlada.

Ahora bien, dado que las excepciones son básicamente errores infrecuentes, podrían tratarse con las mismas técnicas tradicionales que se utilizan para los errores frecuentes. El siguiente pseudocódigo muestra una técnica típica de tratamiento de errores con estructuras de control de flujo.

```
Ejecutar una acción
Si ocurrió un error
       Reportar el error y finalizar
Ejecutar una acción
Si ocurrió un error
       Reportar el error y finalizar
...
```

Bajo esta técnica se tiene las siguientes ventajas:

- El tratamiento de un error se realiza en la vecindad donde ocurre.
- Es fácil reconocer el código que maneja cada error en el programa, y por tanto, entenderlo. Como contraparte, es fácil reconocer la fuente de un error.
- Es fácil realizar un seguimiento a la ejecución del programa.

Las desventajas son:

- Es fácil que el código del programa termine minado con código de manejo de error, lo que hace difícil distinguir la tarea limpia del programa, ésto es, la tarea que se intenta realizar independientemente de los errores que puedan ocurrir o asumiendo que no ocurren. Un código así es difícil de mantener.
- Si un error es infrecuente pero potencialmente puede ocurrir en distintas partes del programa, el programa final contendrá mucho código repetitivo de manejo de dichos errores.
- Si un error es infrecuente, el programador podría pasarlo por alto inadvertidamente o bien tendría la tendencia a dejar su tratamiento para después, siendo dicho error olvidado o bien tratado sin un esfuerzo mucho mayor del que hubiese sido necesario en un inicio. Este tipo de errores son un motivo común de la falta de tolerancia a fallos de los programas.
- Si un error es infrecuente pero potencialmente puede ocurrir en distintas partes del programa, todas las verificaciones correspondientes a dicho error se realizarán, aún cuando éste no ocurra, lo que le resta eficiencia (mayor tiempo de CPU y recursos) al programa.

Como puede apreciarse, el tratamiento tradicional de errores es adecuado cuando éstos son frecuentes, pero no cuando son infrecuentes, como las excepciones.

Las excepciones pueden ser generadas por errores detectados por el hardware (como una división entera entre cero) y capturados por el sistema operativo, producidos por el propio sistema operativo debido a un error de lógica interno (detección de un nivel de memoria disponible debajo de un límite de seguridad), producidos por alguna librería utilizada por nuestro programa o por nuestro mismo programa.

Los lenguajes de programación que distinguen el concepto de excepción, como C++, Java y C#, utilizan una técnica muy diferente al tratamiento tradicional de errores. Esta técnica se basa en la idea de separar el código principal del programa (el código propio de la tarea que el programa desea realizar, un código limpio de verificación de excepciones), del código de manejo de las excepciones. Como puede entenderse, este código limpio seguirá conteniendo el código de manejo de los errores frecuentes, bajo las técnicas tradicionales.

En las siguientes secciones veremos cómo se implementa el tratamiento de excepciones en C++, Java y C#, así como las ventajas y desventajas de su uso.

Implementación

Tanto Java como C# se basan en el modelo de C++ para el tratamiento de excepciones, tema que revisaremos a continuación para después tratar las diferencias que existen entre los tres lenguajes.

C++

Para manejar excepciones que pueden ocurrir dentro de un bloque de código, es necesario "delimitar" dicho bloque. Para ésto, se utiliza la palabra reservada ***try***. La sintaxis a utilizar es:

```
try {
    // Aquí va el código del que se desea controlar las excepciones
    // que produzca.
}
```

Dentro del bloque *try* se colocará el código del que se espera monitorear las excepciones que produzca. Cuando se produce una excepción, se ejecutará un determinado bloque de código llamado ***manejador de excepción***. Estos bloques de código deben de ir inmediatamente después del bloque *try* e igualmente deben ser "delimitados" para cada tipo en particular de excepción. Para ésto, se utiliza la palabra reservada ***catch***. La sintaxis a utilizar será:

```
catch( TipoDeExcepcion e ) {
    // Aquí va el código que se ejecutara en caso que se produzca
// una excepción del tipo "TipoDeExcepcion".
}
```

Un bloque *try* puede ir seguido por tantos bloques *catch* como tipos de excepciones se desee manejar. El tipo de variable TipoDeExcepcion determina el tipo de excepción que manejara un bloque *catch*.

Dentro del bloque *try*, una excepción puede ser producida por:

- Una sentencia *throw* que explícitamente dispare la excepción.
- Una llamada a una función que dispare la excepción. Dicha función podría disparar la excepción explícitamente o llamar a otra función (y ésta a otra y así sucesivamente) la cual sea quien realmente dispare la excepción.
- La creación de un objeto, debido a un error dentro del constructor utilizado.
- Un error del sistema operativo durante la ejecución de cualquier sentencia dentro del bloque *try*.
- Una interrupción capturada por el sistema operativo y notificada al programa en ejecución a manera de una excepción.

El siguiente código muestra un ejemplo simple de un código en C++ que produce y maneja una excepción.

```
#include <stdio.h>

class Fecha {
        int dia, mes, anho;
public:
        Fecha(char* pszFecha) {
                if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3)
                        throw -1;
```

```
                if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
                        throw -2;
        }
        void Imprimir() {
                printf("[%d/%d/%d]", dia, mes, anho);
        }
};

void main() {
        printf("Inicio\n");
        try {
                char szFecha[20];
                puts("Ingrese una fecha :");
                scanf("%s", szFecha);
                Fecha fecha(szFecha);
                puts("La fecha ingresada es :");
                fecha.Imprimir();
        }
        catch(int ex) {
                printf("Excepción: código = %d\n", ex);
        }
        printf("Fin\n");
}
```

En el código anterior, si el constructor de la clase *Fecha* encuentra que existe un error en la información pasada como parámetro, no tiene forma de remediarlo sin tener que agregar código duro al programa (por ejemplo, una dato miembro de *Fecha* que funcione como bandera y que indique si el constructor falló o no, de forma que cualquier llamada a métodos de la clase deban verificar dicho valor antes de realizar su tarea).

Note que en el constructor de la clase *Fecha* se utiliza la llamada a una sentencia *throw* para producir un error tipo excepción. A esto se le conoce como *disparar una excepción*, y al lugar donde ocurre, *punto de excepción*. La sentencia *throw* es seguida de una expresión cuyo tipo de dato al que se evalúa determina el tipo de la excepción. En el ejemplo, la excepción es de tipo *int*. Igualmente, el tipo de dato del argumento del bloque *catch* determina el tipo de este bloque.

El flujo de ejecución del programa es:

- Si no ocurre ninguna excepción, se ejecuta todo el bloque *try* y luego se salta la ejecución a la siguiente instrucción debajo del último bloque *catch*.

- Si ocurre una excepción, la ejecución se detiene en el punto de excepción. Si alguno de los bloques *catch* que siguen al bloque *try* coincide con el tipo de la excepción generada, entonces se ejecuta dicho bloque (se dice que dicho bloque *catch* ha capturado la excepción), luego de lo cual, se salta la ejecución a la siguiente instrucción debajo del último bloque *catch*. Si ninguno de los bloques *catch* tiene un tipo adecuado al tipo de la excepción, la ejecución salta fuera de la función o método donde ocurrió, en este caso, *main*.

De lo anterior se puede deducir que:

- Cuando ocurre una excepción, el código que sigue desde el punto de excepción hasta el fin del bloque try, nunca se ejecuta. A este modelo de manejo de excepciones se le llama ***termination model.***

- No todas las excepciones son necesariamente capturadas donde se generan o siquiera por el mismo programa. Más adelante veremos estos casos.

El código anterior genera excepciones de tipo int, sin embargo, dado que es posible manejar, mediante el parámetro del bloque catch, la información sobre la excepción, lo más usual es utilizar clases, en lugar de datos

primitivos, como tipos de excepción. El siguiente código modifica el ejemplo anterior para utilizar clases en lugar de datos primitivos.

```
#include <stdio.h>

class ErrorEnFormato {};
class ErrorFechaInvalida {};
class Fecha {
    int dia, mes, anho;
public:
    Fecha(char* pszFecha) {
        if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3) {
           ErrorEnFormato ex;
           throw ex;
        }
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
           throw ErrorFechaInvalida();
    }
    void Imprimir() {
        printf("[%d/%d/%d]", dia, mes, anho);
    }
};

void main() {
    try {
        char szFecha[20];
        puts("Ingrese una fecha :");
        scanf("%s", szFecha);
        Fecha fecha(szFecha);
        puts("La fecha ingresada es :");
        fecha.Imprimir();
    }
    catch(ErrorEnFormato ex) {
        printf("Error: el formato es incorrecto\n");
    }
    catch(ErrorFechaInvalida ex) {
        printf("Error: la fecha es inválida\n");
    }
}
```

El utilizar clases en lugar de datos primitivos permite la posibilidad de guardar más información acerca de la excepción ocurrida (como en qué archivo fuente ocurrió y dentro de éste, en qué línea) y pasar esta información al manejador de excepción correspondiente. El siguiente código modifica el ejemplo anterior para utilizar información adicional dentro de las clases de excepciones.

```
#include <stdio.h>

class ErrorEnFormato {
    char* pszLugar;
public:
    ErrorEnFormato(char* psz) : pszLugar(psz) {}
    char* Lugar() { return pszLugar; }
};

class ErrorFechaInvalida {
    char* pszLugar;
    char* pszFecha;
public:
    ErrorFechaInvalida(char* pszL, char* pszF) : pszLugar(pszL), pszFecha(pszF) {}
    char* Lugar() { return pszLugar; }
    char* LaFecha() { return pszFecha; }
};

class Fecha {
    int dia, mes, anho;
public:
    Fecha(char* pszFecha) {
        if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3)
           throw ErrorEnFormato("Fecha::Fecha");
```

```
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw ErrorFechaInvalida("Fecha::Fecha", pszFecha);
    }
    void Imprimir() {
        printf("[%d/%d/%d]", dia, mes, anho);
    }
};

void main() {
    try {
        char szFecha[20];
        puts("Ingrese una fecha :");
        scanf("%s", szFecha);
        Fecha fecha(szFecha);
        puts("La fecha ingresada es :");
        fecha.Imprimir();
    }
    catch(ErrorEnFormato ex) {
        printf("Error: el formato es incorrecto en %s\n", ex.Lugar());
    }
    catch(ErrorFechaInvalida ex) {
        printf("Error: el formato de la fecha [%s] es incorrecto en %s\n",
            ex.LaFecha(), ex.Lugar());
    }
}
```

Note que en el código anterior, existe información común entre ambos tipos de excepciones. Este aspecto es explotado utilizando el polimorfismo en la implementación en C++. El siguiente código modifica el ejemplo anterior para utilizar una clase base para ambas excepciones.

```
#include <stdio.h>

class Excepcion {
    char* pszLugar;
public:
    Excepcion(char* psz) : pszLugar(psz) {}
    char* Lugar() { return pszLugar; }
};

class ErrorEnFormato : Excepcion {
public:
    ErrorEnFormato(char* psz) : Excepcion(psz) {}
};

class ErrorFechaInvalida : Excepcion {
    char* pszFecha;
public:
    ErrorFechaInvalida(char* pszL, char* pszF)
 : Excepcion(pszL), pszFecha(pszF) {}
    char* LaFecha() { return pszFecha; }
};

class Fecha {
    int dia, mes, anho;
public:
    Fecha(char* pszFecha) {
        if(pszFecha == 0)
            throw Excepcion("Fecha::Fecha");
        if(sscanf(pszFecha, "%d-%d-%d", &dia, &mes, &anho) != 3)
            throw ErrorEnFormato("Fecha::Fecha");
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw ErrorFechaInvalida("Fecha::Fecha", pszFecha);
    }
    void Imprimir() {
        printf("[%d/%d/%d]", dia, mes, anho);
    }
};

void main() {
    try {
```

```
        char szFecha[20];
        puts("Ingrese una fecha :");
        scanf("%s", szFecha);
        Fecha fecha(szFecha);
        puts("La fecha ingresada es :");
        fecha.Imprimir();
    }
    catch(ErrorEnFormato ex) {
        printf("Error: el formato es incorrecto en %s\n", ex.Lugar());
    }
    catch(ErrorFechaInvalida ex) {
        printf("Error: el formato de la fecha [%s] es incorrecto en %s\n",
           ex.LaFecha(), ex.Lugar());
    }
    catch(Excepcion ex) {
        printf("Error: en %s\n", ex.Lugar());
    }
}
```

Note, en el código anterior, el uso de una clase base para las excepciones anteriormente definidas. En el constructor se realiza la verificación de un nuevo tipo de error, que la cadena pasada sea un puntero nulo, para lo cual se usa la clase base *Exception*, dado que las otras no son adecuadas. Para esta clase base se le agrega un bloque *catch*.

Para mostrar porqué se dice que el comportamiento es polimórfico, modifique este código de forma que el bloque *catch Excepcion* esté antes del bloque *catch ErrorFechaInvalida*. Corra el programa y notará que las excepciones del tipo *ErrorFechaInvalida* son ahora capturadas por el bloque *catch Excepcion*. Si el bloque *catch Excepcion* lo colocara antes del bloque *catch ErrorEnFormato*, todas las excepciones serían capturadas por el bloque *catch Excepcion*. La regla de selección del bloque *catch* que capturará una excepción en C++ es:

> ***"Dada una excepción ocurrida dentro de un bloque try, se ejecutará el primer bloque catch, según el orden de declaración de arriba hacia abajo, cuyo tipo coincida o sea una clase base del tipo de la excepción."***

En el código anterior, pudo utilizarse únicamente el bloque *catch Excepcion* para capturar todas las excepciones. Sin embargo, como puede entenderse, hay información extra que cada tipo de excepción podría manejar y ser útil para el adecuado manejo de ésta.

C++ permite definir además un bloque *catch* que capture todas las excepciones, sin importar su tipo, con el siguiente formato:

```
catch(...) {
    // Aquí va el manejador de la excepción
}
```

Este bloque *catch* debe ser el último de todos, sino se produce un error de compilación.

Aunque en todos los ejemplos anteriores, las excepciones generadas eran capturadas y todas se manejaban en un mismo bloque try, no siempre es así. El siguiente código muestra los diferentes casos que pueden ocurrir al generarse una excepción.

```
#include <iostream.h>

class Excepcion {};

void NoGeneraExcepcion() {
    cout << "Inicio de 'NoGeneraExcepcion'" << endl;
    try {
        cout << "Dentro del try de 'NoGeneraExcepcion'" << endl;
    }
    catch(Excepcion ex) {
```

```
        cout << "Dentro del catch de 'NoGeneraExcepcion'" << endl;
    }
    cout << "Fin de 'NoGeneraExcepcion'" << endl;
}

void GeneraCapturaExcepcion() {
    cout << "Inicio de 'GeneraCapturaExcepcion'" << endl;
    try {
        cout << "Dentro del try de 'GeneraCapturaExcepcion'" << endl;
        throw Excepcion();
        cout << "Este saludo nunca se muestra" << endl;
    }
    catch(Excepcion ex) {
        cout << "Dentro del catch de 'GeneraCapturaExcepcion'" << endl;
    }
    cout << "Fin de 'GeneraCapturaExcepcion'" << endl;
}

void GeneraNoCapturaExcepcion() {
    cout << "Inicio de 'GeneraNoCapturaExcepcion'" << endl;
    try {
        cout << "Dentro del try de 'GeneraNoCapturaExcepcion'" << endl;
        throw Excepcion();
        cout << "Este saludo nunca se muestra" << endl;
    }
    catch(int ex) {
        cout << "Dentro del catch de 'GeneraNoCapturaExcepcion'" << endl;
    }
    cout << "Fin de 'GeneraNoCapturaExcepcion'" << endl;
}

void GeneraCapturaRedisparaExcepcion() {
    cout << "Inicio de 'GeneraCapturaRedisparaExcepcion'" << endl;
    try {
        cout << "Dentro del try de 'GeneraCapturaRedisparaExcepcion'" << endl;
        throw Excepcion();
        cout << "Este saludo nunca se muestra" << endl;
    }
    catch(Excepcion ex) {
        cout << "Dentro del catch de 'GeneraCapturaRedisparaExcepcion'" << endl;
        throw ex;
    }
    cout << "Fin de 'GeneraCapturaRedisparaExcepcion'" << endl;
}

void main() {
    cout << "Llamando a 'NoGeneraExcepcion'" << endl;
    NoGeneraExcepcion();

    cout << "Llamando a 'GeneraCapturaExcepcion'" << endl;
    GeneraCapturaExcepcion();

    try {
        cout << "Llamando a 'GeneraNoCapturaExcepcion'" << endl;
        GeneraNoCapturaExcepcion();
    }
    catch(Excepcion ex) {
        cout << "Excepcion capturada desde 'GeneraNoCapturaExcepcion' en"
           <<" 'main'" << endl;
    }

    try {
        cout << "Llamando a 'GeneraCapturaRedisparaExcepcion'" << endl;
        GeneraCapturaRedisparaExcepcion();
    }
    catch(Excepcion ex) {
        cout << "Excepcion capturada desde 'GeneraCapturaRedisparaExcepcion'"
           <<" en 'main'" << endl;
    }
    throw ex;
    cout << "Fin del programa";
```

```
}
```

Al ejecutarse se mostrará una salida en una ventana de comandos, como la mostrada en la figura 7.1

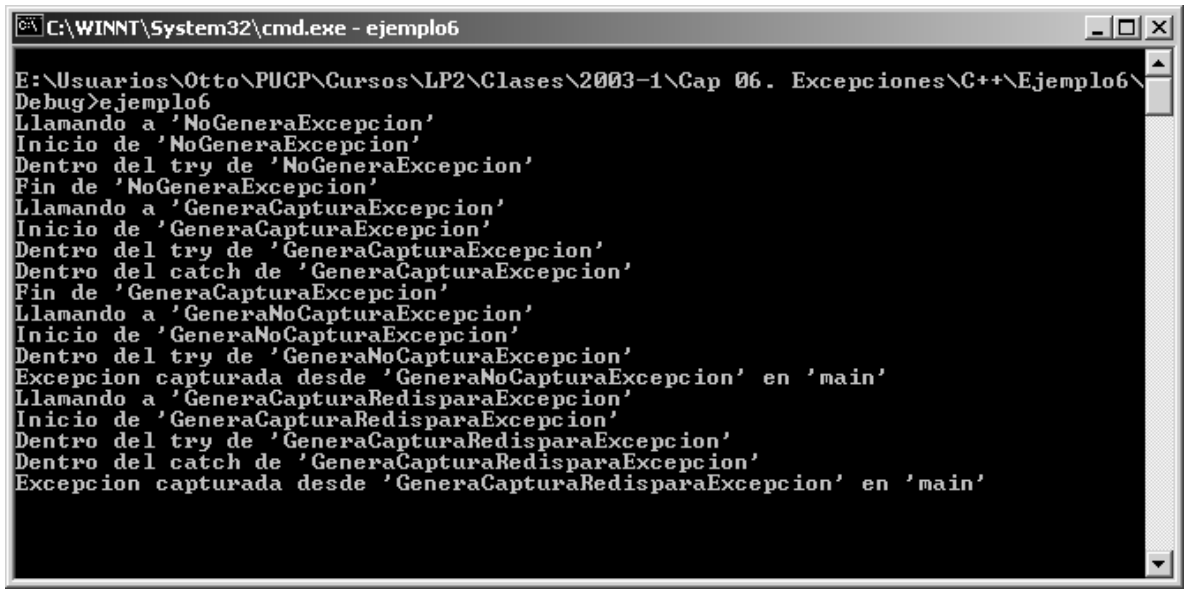

Figura 7.1. Ejecución de ejemplo de excepciones en C++

Inmediatamente después se muestra una ventana con el siguiente mensaje:

Figura 7.2. Mensaje en la ejecución de ejemplo de excepciones en C++

Como puede entenderse por la ejecución del programa, las excepciones que suceden en la llamada a una función son disparadas fuera de éste cuando la función no la captura. Luego, desde donde se llamó a la función que generó la excepción (la llamada a *GeneraNoCapturaExcepcion* desde *main*) se considera que dicha función generó una excepción, por lo que es factible capturarla. En el caso de la última excepción, al final de *main*, ésta no es capturada por nadie por lo que sale fuera del método *main*, siendo capturada por el propio sistema operativo que se encarga de mostrar la ventana mostrada en la figura 7.2 o una similar dependiendo la forma en que se compiló el programa y las capacidades de manejo de excepciones del sistema operativo.

Otro punto importante relacionado al código anterior, es que dentro de un manejador de excepción, un bloque *catch*, también es posible que ocurra una excepción, que es el caso de la función *GeneraCapturaRedisparaExcepcion*. En este caso, la excepción se dispara directamente fuera de la función donde está dicho bloque *catch*, aún cuando pudiera existir algún bloque *catch* debajo del anterior, cuyo tipo sea adecuado al tipo de la excepción generada.

Existe un punto importante no tratado en los códigos de ejemplo anteriores, la pérdida de recursos debido a una excepción. Es importante entender que en los códigos anteriores ésto no ocurre, debido a que los objetos creados en el bloque *try* eran reservados en la memoria de pila, por lo que al salir la ejecución del bloque *try* (sea o no por haberse producido una excepción) esta memoria es liberada automáticamente. Para mostrar un caso donde sí se pierden recursos, revisemos el siguiente código.

```
class Excepcion {};

void main() {
    try {
        char* pCadena = new char[10];
        throw Excepcion();
        delete [] pCadena;
    }
    catch(Excepcion ex) {
    }
}
```

En el código anterior, debido a que ocurre una excepción antes de poder liberar el recurso reservado, esta liberación nunca podrá realizarse. Tampoco es posible liberar el recurso en el bloque *catch* debido a que la variable *pObj* es local al bloque try, por lo que para hacer esto posible debería definirse dicha variable como local a *main*. De esta forma, la liberación de memoria se debería hacer al final de todos los bloques *catch* del bloque *try*. Sin embargo, si la excepción producida no es capturada por ninguno de los bloques *catch*, entonces no habrá manera de liberar el recurso, a menos que la variable *pObj* se declare como *global* de forma que pueda ser liberado por algún otro manejador de excepción.

Como puede verse, C++ no ofrece una solución estándar a este tipo de pérdida de recursos. Más adelante veremos cómo es que Java y C# tratan este aspecto.

Por último, es importante señalar que el modelo de manejo de excepciones en C++ sólo permite manejar como excepciones, errores síncronos. Un ejemplo de esto y de cómo afecta en la decisión de si un error es buen candidato para ser tratado como excepción, se verá en el siguiente capítulo de *Programación Concurrente*.

Java

Al igual que C++, Java comparte el mismo modelo de manejo de excepciones que C++, pero a diferencia de éste, Java cuenta con un soporte más completo. A continuación se detallan las principales diferencias:

1. Todos los tipos de excepciones heredan de una clase base

El árbol de herencia de las clases para excepciones es:.

```
java.lang.Object
       java.lang.Throwable
              java.lang.Error
              java.lang.Exception
              java.lang.RuntimeException
```

La clase *Throwable* es la clase base de todos los errores y excepciones de Java manejados por el mecanismo de manejo de excepciones. El intérprete de Java y el programador sólo pueden disparar excepciones de un tipo que derive de *Throwable*.

La clase *Error* es la clase base de excepciones generadas por el intérprete de Java, por lo que no deben ser manejadas por el programador, dado que son errores para los que sólo el intérprete puede dar una solución satisfactoria.

La clase *Exception* es la clase base de todas excepciones con las que el programador sí debe lidiar, a excepción de las que derivan de *RuntimeException*, las que tienen un tratamiento especial.

Si un programa deseara crear sus propias excepciones, heredaría de la clase Exception.

2. Las excepciones deben ser tratadas por el programa.

El programador **debe capturar explícitamente** las excepciones de dichos tipos, o bien **indicar explícitamente** que no desea manejarlas. Las excepciones que derivan de *RuntimeException* corresponden a errores de lógica en la programación, por lo que el lenguaje acepta que no sean explícitamente tratadas, dado que sólo son usadas para depurar los programas y eliminar dichos errores. Una vez eliminados estos errores, esas excepciones ya no se presentarán, y el código escrito para manejarlas dejaría de ser útil. A continuación un ejemplo ilustra este aspecto:

```
public class Ejemplo1 {
    public static void main(String args[]) {
        // sentencias fuera del bloque de control de excepciones
        int Arreglo[] = { 1, 2, 3, 4 };

        // definición del bloque try
        try {
           for( int i = 0; i < 5; i++ ) {
               int iValor = Arreglo[ i ];
               System.out.println("Valor " + i + " = " + iValor );
           }
        }
        // definición del bloque catch
        catch( ArrayIndexOutOfBoundsException e ) {
           System.out.println("Ocurrió la siguiente excepción = " + e );
        }
        // definición del bloque finally
        finally {
           System.out.println("Ejecución del bloque finally" );
        }

        // sentencias fuera del bloque de control de excepciones
        System.out.println("Fin del programa." );
    }
}
```

La excepción *ArrayIndexOutOfBounds* hereda de *RuntimeException* y es generada cuando se accede a un elemento de un arreglo utilizando un índice inválido. Esto es justamente lo que sucede en el código anterior, donde se intenta acceder al quinto elemento del arreglo (índice $i = 4$) produciéndose una excepción, dado que el arreglo sólo tiene cuatro elementos. Este error se puede subsanar reemplazando la declaración del bucle, por ejemplo, de la siguiente manera:

```
for( int i = 0; i < Arreglo.length; i++ ) {
```

Por tanto, una vez hecho ésto, dicha excepción nunca volverá a ocurrir, por lo que todo el código escrito para el manejo de ésta pasa a ser inútil. Todo el resto de excepciones que heredan de *Exception* y no de *RuntimeException* **deben ser manejadas explícitamente** por el programador. El siguiente programa muestra este caso:

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

class Ejemplo3 {

    public static void main(String args[]) {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        try {
           System.out.print("Ingrese un primer entero : ");
```

```
            String cadena1 = input.readLine();
            System.out.print("Ingrese un segundo entero : ");
            String cadena2 = input.readLine();
            int entero1 = Integer.parseInt(cadena1);
            int entero2 = Integer.parseInt(cadena2);
            System.out.println("La suma da : " + (entero1 + entero2));
        }
        catch(IOException ex) {
            System.out.println("Ocurrio la sgte. excepcion durante la lectura " + ex);
        }
    }
}
```

El código anterior utiliza el método *readLine* de la clase *BufferedReader* para leer el texto ingresado por teclado desde la ventana de comandos del programa. Dicho método puede producir una excepción del tipo *IOExcepcion*, la cual no deriva de *RuntimeException*, por lo que tenemos la obligación de capturarla. Otra opción es indicar explícitamente que no deseamos manejar dicha excepción mediante la clausula *throws*. Esta cláusula se coloca luego de la declaración de parámetros de un método. El siguiente código modifica el anterior para utilizar una cláusula *throws*.

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

class Ejemplo3 {

    public static void main(String args[]) throws IOException {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Ingrese un primer entero : ");
        String cadena1 = input.readLine();
        System.out.print("Ingrese un segundo entero : ");
        String cadena2 = input.readLine();

        int entero1 = Integer.parseInt(cadena1);
        int entero2 = Integer.parseInt(cadena2);
        System.out.println("La suma resultante es : " + (entero1 + entero2));
    }
}
```

La cláusula *thows* puede ir precedida de una lista de tipos de excepciones, separadas por comas. Esta cláusula le dice al compilador de Java *"se que el código en este método produce estas excepciones, pero no deseo manejarlas aquí, sino en el método que llame a éste"*.

Si el código en nuestro programa produce excepciones que deben ser manejadas explícitamente, ya sea mediante una secuencia *throw-catch* o con una cláusula *throws*, y no lo son, se generarán errores durante la compilación indicando que esto debe hacerse.

3. Se define un bloque especial que siempre se ejecuta.

El problema de la pérdida de recursos del modelo de manejo de excepciones en C++ se soluciona en Java mediante un nuevo bloque, llamado *finally*, que puede ir a continuación de un bloque *try* o del último bloque *catch*, este nuevo bloque siempre se ejecutará, ocurra o no una excepción dentro de *try* y en caso ocurra sea o no capturada por uno de los bloques *catch*. El siguiente código muestra un ejemplo del funcionamiento de *finally*.

```
class Ejemplo3 {
    static public void NoDisparaExcepciones() {
        System.out.println("Inicio de 'NoDisparaExcepciones'");
        try {
            System.out.println("Dentro del bloque try de 'NoDisparaExcepciones'");
        }
        finally {
            System.out.println("Dentro del bloque finally de 'NoDisparaExcepciones'");
        }
```

```
        System.out.println("Fin de 'NoDisparaExcepciones'");
    }

    static public void DisparaCapturaExcepcion() {
        System.out.println("Inicio de 'DisparaCapturaExcepcion'");
        try {
            System.out.println("Dentro del bloque try de 'DisparaCapturaExcepcion'");
            throw new Exception("Excepcion en try de 'DisparaCapturaExcepcion'");
            // el codigo restante de este try nunca se ejecutará
        }
        catch(Exception ex) {
            System.out.println("Dentro del bloque catch de " +
                "'DisparaCapturaExcepcion', excepcion=" + ex.getMessage());
        }
        finally {
            System.out.println("Dentro del bloque finally de 'DisparaCapturaExcepcion'");
        }
        System.out.println("Fin de 'DisparaCapturaExcepcion'");
    }

    static public void DisparaNoCapturaExcepcion() throws Exception {
        System.out.println("Inicio de 'DisparaNoCapturaExcepcion'");
        try {
            System.out.println("Inicio del bloque try de 'DisparaNoCapturaExcepcion'");
            throw new Exception("Excepcion dentro del try de 'DisparaNoCapturaExcepcion'");
            // el codigo restante de este try nunca se ejecutará
        }
        finally {
            System.out.println("Dentro del bloque finally de 'DisparaNoCapturaExcepcion'");
        }
        // el codigo restante de este metodo nunca se ejecutará
    }

    static public void DisparaCapturaRedisparaExcepcion() throws Exception {
        System.out.println("Inicio de 'DisparaCapturaRedisparaExcepcion'");
        try {
            System.out.println("Inicio del try de DisparaCapturaRedisparaExcepcion'");
            throw new Exception("Excepcion en try de 'DisparaCapturaRedisparaExcepcion'");
            // el codigo restante de este try nunca se ejecutará
        }
        catch(Exception ex) {
            System.out.println("Dentro del bloque catch de " +
                "'DisparaCapturaRedisparaExcepcion', excepcion=" + ex.getMessage());
            throw ex;
        }
        finally {
            System.out.println("Dentro del bloque finally de " +
            "'DisparaCapturaRedisparaExcepcion'");
        }
        // el codigo restante de este try nunca se ejecutará
    }

    public static void main(String args[]) {
        System.out.println("Llamando 'NoDisparaExcepciones'");
        NoDisparaExcepciones();
        System.out.println("\nLlamando 'DisparaCapturaExcepcion'");
        DisparaCapturaExcepcion();
        System.out.println("\nLlamando 'DisparaNoCapturaExcepcion'");
        try {
            DisparaNoCapturaExcepcion();
        }
        catch(Exception ex) {
            System.out.println("Dentro del bloque catch 1 de 'main', excepcion=" +
                ex.getMessage());
        }
        finally {
            System.out.println("Dentro del bloque finally 1 de 'main'");
        }
        System.out.println("\nLlamando 'DisparaCapturaRedisparaExcepcion'");
        try {
            DisparaCapturaRedisparaExcepcion();
```

```
            }
            catch(Exception ex) {
                System.out.println("Dentro del bloque catch 2 de 'main', excepcion="
                    +ex.getMessage());
            }
            finally {
                System.out.println("Dentro del bloque finally 2 de 'main'");
            }
        }
    }
```

El código anterior genera una salida semejante a la siguiente:

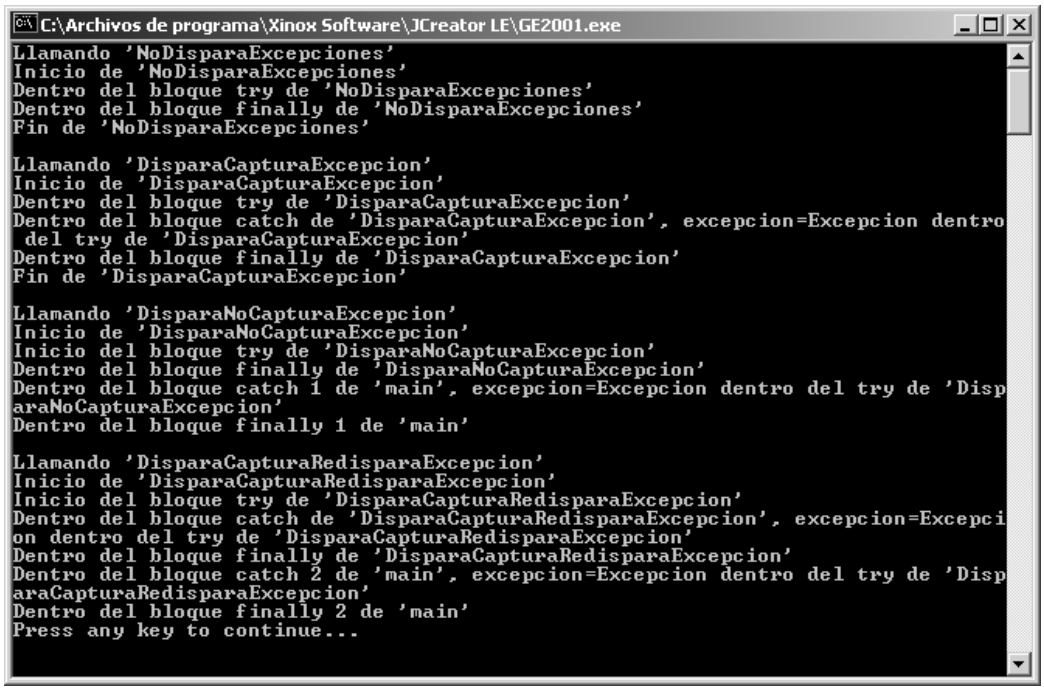

Figura 7.3. Ejecución de ejemplo de excepciones en Java

Ésto da una clara idea de que el bloque *finally* se ejecuta siempre que se haya entrado a ejecutar el bloque *try* correspondiente. La lista de combinaciones *try-catch-finally* que se pueden tener es:

- Un bloque *try* seguido de uno o más bloques *catch*.
- Un bloque *try* seguido de uno o más bloques *catch*, seguidos de un bloque *finally*.
- Un bloque *try* seguido de un bloque *finally*.

C#

C# comparte el mismo modelo de manejo de excepciones que C++, pero a diferencia de éste, C# cuenta con un soporte más completo, muy similar al de Java.

El árbol de herencia de las clases para excepciones es:

```
System.Object
    System.Exception
        System.ApplicationException
        System.SystemException
```

Al igual que Java, la clase *Exception* es la clase base de todos los tipos de excepciones. La clase SystemException es la clase base que utiliza el intérprete de ..NET para todas las excepciones que puede generar. La clase ApplicationException es la clase base que debe utilizar el programador para definir sus propias excepciones.

A diferencia de Java, C# no diferencia entre excepciones que deben tratarse y las que sólo sirven para depuración, es decir, si un determinado código puede generar una excepción, no es obligatorio colocarlo dentro de un bloque *try* seguido de un *catch* que lo capture. Por el mismo motivo, tampoco existe una cláusula *throws* con la que se indique que una excepción no se desea tratar en el método donde ocurre.

Fuera de las diferencias antes indicadas, la implementación en Java y C# es igual. Un código de ejemplo de excepciones en C# es:.

```
using System;

class ExcepcionFormatoInvalido : ApplicationException {
    public ExcepcionFormatoInvalido(string mensaje) : base(mensaje) {}
}
class ExcepcionFechaInvalida : ApplicationException {
    public ExcepcionFechaInvalida(string mensaje) : base(mensaje) {}
}
class Fecha {
    int dia, mes, anho;
    public Fecha(string sFecha) {
        char[] delimitadores = {'-','/'};
        string[] tokens = sFecha.Split(delimitadores);
        if(tokens.Length != 3)
            throw new ExcepcionFormatoInvalido("Los delimitadores son inválidos");
        dia = Convert.ToInt32(tokens[0]);
        mes = Convert.ToInt32(tokens[1]);
        anho = Convert.ToInt32(tokens[2]);
        if(anho < 0 || dia < 1 || 31 < dia || mes < 1 || 12 < mes)
            throw new ExcepcionFechaInvalida("La fecha no existe");
    }
    public void Imprimir() {
        Console.WriteLine("[" + dia + "/" + mes + "/" + anho + "]");
    }
};
class MainClass {
    public static void Main(string[] args) {
        Console.WriteLine("Inicio");
        try {
            Console.Write("Ingrese una fecha : ");
            string sFecha = Console.ReadLine();
            Fecha fecha = new Fecha(sFecha);
            Console.Write("La fecha ingresada es : ");
            fecha.Imprimir();
        }
        catch(Exception ex) {
            Console.WriteLine("Excepción = " + ex);
        }
        finally {
            Console.WriteLine("Ejecutandose el bloque finally");
        }
        Console.WriteLine("Fin");
    }
}
```

Ventajas, desventajas y criterios de uso

La separación del código limpio del código de manejo de excepciones permite eliminar las desventajas mencionadas con las técnicas tradicionales. Por lo tanto, las excepciones:

- Permiten tener un código limpio, por tanto, es más fácil de mantener.

- Eliminación del código repetitivo en el manejo de una misma excepción.

- Es más fácil agregar código para el manejo de una excepción que se dejó *para después* al inicio de un programa, por tanto, es más fácil aumentar la tolerancia a fallos de éste conforme se va avanzando en su desarrollo.

- Dado que no se realizan todas las verificaciones para las excepciones en el código limpio del programa, el compilador puede optimizar su verificación, lo que reduce y en algunos casos elimina el costo de eficiencia en la ejecución del programa final.

Es importante resaltar que la técnica mostrada del manejo de excepciones es más eficiente que la tradicional cuando no ocurre una excepción, pero es mucho menos eficiente cuando sí ocurre. Por lo tanto, es fácil deducir que si un error muy frecuente es tratado como una excepción, el programa final sería menos eficiente que si sólo utilizara técnicas tradicionales para dichos errores.

Es importante tomar en cuenta las siguientes desventajas propias al manejo de excepciones:

- Al estar el código de manejo de excepciones separado del código limpio del programa, se dificulta entender el flujo de ejecución del programa cuando ocurre una excepción.

- Por el mismo motivo anterior, es más difícil relacionar un código de manejo de una excepción con la fuente del mismo.

En general, la mayoría de los errores pueden ser tratados como excepciones (existen algunos que sólo pueden ser manejados como excepciones), pero no todos los errores son buenos candidatos. Algunos criterios que pueden servir para decidir si un error debe ser tratado como una excepción son:

- El error es infrecuente.

- En el contexto donde ocurre el error no es posible darle una solución satisfactoria. Un ejemplo típico es un error dentro del constructor de una clase.

- El error está relacionado con un mal funcionamiento del sistema operativo o de alguna librería con la que interactúa el programa. Un error en el ingreso de datos por parte del usuario es un mal candidato para una excepción, pero un error en la reserva de memoria sí lo es.

- El error es síncrono.

El *manejador de excepción* es un código que comunmente se coloca en un nivel más bajo en el árbol de llamada a métodos, que el código que dispara la excepción. Eso permite abarcar las excepciones posibles desde muchos métodos. Por ejemplo, si el método C es llamado por el método B, y éste por el método A, y el método C dispara una excepción, el código "exception handler" de éste puede colocarse tanto en el método A, como en el B o en el C. Luego, las excepciones son utilizadas para tipos de errores que pueden producirse en diferentes lugares a lo largo de un programa y que se desea sean procesados de manera centralizada. Sin embargo, es importante tener en cuenta que cuanto más lejos esté el *manejador de excepción* del *punto de excepción*, más difícil podría ser manejar adecuadamente dicha excepción, dado que la misma podría ser generada por más de un *punto de excepción* y podría requerirse código especial para cada caso.

Por último, dado que las técnicas tradicionales de tratamiento de errores se realizan con las mismas estructuras utilizadas para el control del flujo de un programa, es frecuente la tendencia a querer utilizar las técnicas de manejo de excepciones para tareas diferentes al tratamiento de excepciones. Ésto debe evitarse, dado que reduciría la eficiencia del programa y la claridad de su código.

El tema de la sincronización se verá en el siguiente capítulo, *Programación Concurrente*.