Tema 4

Excepciones en Java

Manejo de Problemas de Ejecución

Si en la ejecución de una pieza de código se prevé que pueda ocurrir un error, hay dos formas de tratar ese posible error:

- Escribir código para evitar que el error ocurra. Por ejemplo supongamos que en un programa se desea hacer una división. Si existe la posibilidad de que el divisor tome el valor de 0, podemos evitar la posible división entre 0, verificando el valor del divisor y tomando una acción si este vale cero, como en el siguiente ejemplo:

```
if(divisor != 0) cociente = dividendo / divisor;
else System.out.println(“División entre cero”);
```

- Escribir código que maneje el error cuando éste ocurra. Para esto se requiere que el lenguaje provea de un mecanismo que detecte el error al tiempo de ejecución y nos notifique de ello. Java, al igual que otros lenguajes orientados a objetos, posee ese mecanismo de detección y notificación llamado Excepción.

Excepciones

Cuando ocurre un error en la ejecución de un programa, Java genera una excepción y activa una alarma para que el sistema de tiempo de ejecución emprenda una acción urgente.

El sistema de ejecución detendrá su modo normal de ejecución y buscará un código que capture la excepción y la maneje, esto es, le de un tratamiento al error.

Luego se apaga la alarma y el sistema continúa la ejecución después del bloque que contiene la instrucción que causó el problema.

Si no existe un código que capture y maneje las excepciones, el sistema terminará la ejecución del programa.

La Jerarquía de Clases de Excepciones

Una excepción es una instancia de la clase `Exception` o de una de sus subclases. La clase `Exception` hereda de la clase `Throwable`. El número de subclases de la clase `Exception` en la API de Java es mayor de 80. Algunas de las subclases se muestran en la figura 4.1:

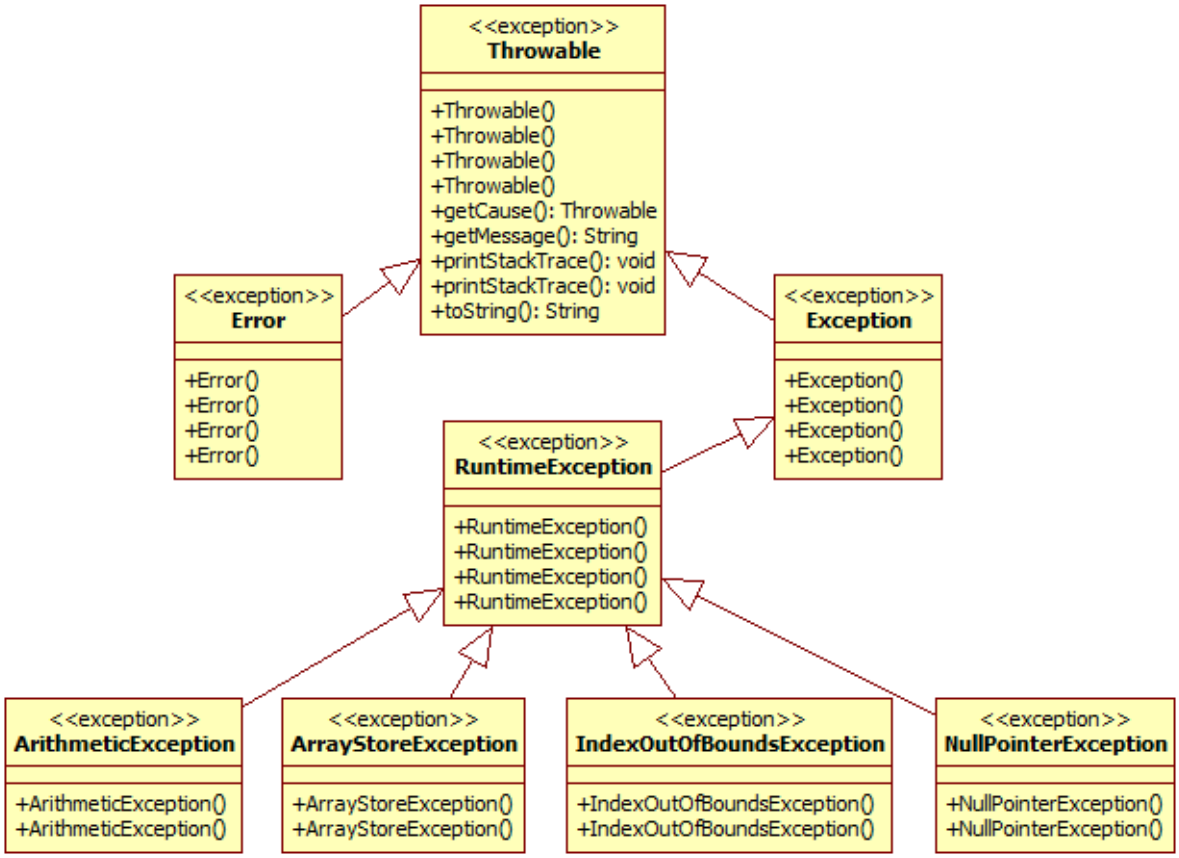

Figura 4.1

La mayoría de las clases de esta jerarquía sólo difieren en el nombre de la clase. Y la decisión de usar una u otra es hacer que el nombre de la clase refleje el tipo de error al que se le está dando un tratamiento. El uso de las clases de la figura 4.1 se muestra en la tabla 4.1:

Un objeto de la clase `Throwable`, igual que uno de la clase `Exception` o cualquiera de sus subclases tiene las siguientes características:

- Contiene una fotografía de la pila de ejecución de su hilo al momento de que fue creado. Esto es, la secuencia de invocaciones a métodos hasta llegar al método en el que se lanzó la excepción.

Tabla 4.1 Algunas Clases de la *Jerarquía de Clases de Excepciones*

Clase de Exception	Descripción
`Throwable`	La clase padre de las clases Exception y Error
`Error`	Indican algunos errores graves que las aplicaciones comunes no deben atrapar. Por lo general son errores que no deben ocurrir.
`Exception`	Es la clase padre de todas las excepciones que las aplicaciones pueden querer atrapar. Es la clase padre de todas las excepciones llamadas Excepciones Verificadas, exceptuando la clase `RuntimeException` y sus descendientes.
`RuntimeException`	Es la clase padre de todas las excepciones llamadas Excepciones No Verificadas. Son las excepciones que pueden ocurrir en la ejecución normal de un programa.
`ArithmeticException`	Error al evaluar una expresión aritmética, como por ejemplo una división entre 0.
`ArrayStoreException`	Se intentó insertar un elemento de tipo incorrecto en el arreglo.
`IndexOutofBoundsException`	Se trató de tener acceso a un elemento de un arreglo o una cadena con un índice inválido (menor que 0 o mayor o igual al tamaño del arreglo o cadena).
`NullPointerException`	Indica que una instancia de un objeto que actualmente es null llama a un método de clase.

- Puede contener un mensaje que dé más información acerca del error.
- Puede contener una causa, una excepción que hizo que se lanzara esta excepción.

Los constructores y algunos de los métodos de la clase `Throwable` se muestran en la siguiente tabla.

Tabla 4.2 Constructores y Métodos de la clase Throwable

`public Throwable()` Construye un objeto del tipo `Throwable` con su mensaje de error igual a null.
`public Throwable(String` *`s`***`)`** Construye un objeto del tipo `Throwable` con su mensaje de error igual al especificado por su parámetro.
`public` **`Throwable(String`** *`s`***`, Throwable`** *`cause`***`)`** Construye un objeto del tipo `Throwable` con su mensaje de error y la causa igual a los especificados por sus parámetros.
`public Throwable (Throwable` *`cause`***`)`** Construye un objeto del tipo `Throwable` con su causa igual a la especificada por su parámetro y un mensaje de error igual a `cause.toString()`, que normalmente contiene el nombre de la clase y mensaje de la causa.
`public String getMessage()` Regresa una cadena con el mensaje de error.

Tabla 4.2 Constructores y Métodos de la clase Throwable (Continuación)

`public Throwable getCause()` Regresa la causa del error o null si la causa no existe o no se conoce.
`public void printStackTrace()` Escribe la representación de este objeto (mediante el método toString()) y la secuencia de invocaciones a métodos hasta llegar al método en el que se lanzó la excepción, en orden inverso, en el flujo estándar de error (la ventana de salida).
`public void printStackTrace(PrintWriter pw)` Escribe la representación de este objeto (mediante el método toString()) y la secuencia de invocaciones a métodos hasta llegar al método en el que se lanzó la excepción, en orden inverso, en el flujo del tipo `PrintWriter` dado por el parámetro.
`public String toString()` Regresa una cadena con la representación de este objeto que normalmente contiene el nombre de la clase y mensaje detallado de la causa.

La clase `Exception` y sus subclases tienen dos o cuatro constructores con los mismos parámetros que la clase `Throwable` y que hacen lo mismo que ellos. Por ejemplo la clase `Exception` tiene los siguientes constructores:

```
public Exception()
public Exception(String s)
public Exception(String s, Throwable cause)
public Exception(Throwable cause)
```

mientras que la clase `ArithmeticException` tiene los siguientes constructores:

```
public ArithmeticException()
public ArithmeticException(String s)
```

La mayoría de las clases de excepción no tiene métodos propios. Sin embargo heredan de la clase `Throwable` su conjunto de métodos.

Manejo de Excepciones

Para manejar un posible error mediante excepciones se utilizan las sentencias `try`, `catch` y `finally` que tienen la siguiente sintaxis:

```
try {
   código que puede generar uno o más errores
}
catch(TipoExcepcion1 varTipoExcepcion1) {
      código para manejar el error tipo 1
}
[catch(TipoExcepcion2 varTipoExcepcion2) {
      código para manejar el error tipo 2
```

```
}]...
[finally {
      código que se ejecutará tanto si se generó el error o no
}]
```

Bloque try

Un bloque `try` encierra el código que puede generar una o más excepciones y que se tiene la intención de manejarlas.

Si no ocurren excepciones mientras se ejecutan las instrucciones del bloque `try` el control del programa se brinca todos los bloques `catch` y pasa al código del bloque `finally` (si hay).

Bloques catch

Cada bloque catch tiene un parámetro, en el recibe la excepción que atrapa. El tipo del parámetro establece que ese bloque catch atrapa las excepciones de esa clase o las excepciones de alguna de sus subclases.

Cuando ocurre una excepción en una sentencia del bloque try, el sistema compara el tipo de la excepción ocurrida con los tipos de excepciones que atrapan cada bloque catch. El control del programa pasa al bloque catch que atrapa la excepción generada y se ignoran los demás bloques catch.

Por último, el control del programa pasa al código del bloque finally (si hay).

Bloque finally

El bloque `finally` contiene el código que debe ejecutarse sin importar que se genere y capture una excepción o no.

Un bloque `try` sólo puede tener un bloque `finally` que debe aparecer inmediatamente después del último bloque `catch`.

Un bloque `try` que tiene un bloque `finally` no necesita bloques catch.

El bloque `finally` siempre se ejecuta aún si los bloque `try` o `catch` contienen sentencias `break` o `return`.

Orden de los Bloques catch

Supongamos que en un bloque `try` se pueden generar dos tipos de excepciones: `Excepcion1` y `Excepcion2`. Suponga también que `Excepción2` es una subclase de `Excepción1`. Si las codificamos como sigue:

```
try {
   ...
}
catch(Excepcion1 e1) {
   ...
}
catch(Excepcion2 e2) {
   ...
}
```

¿Qué pasaría si ocurre una excepción del tipo `Exception2`?

El bloque `catch` al que pasaría el control del programa sería el bloque con el argumento `e1`, ya que como el sistema compara cada bloque `catch` por compatibilidad de asociación y como `Excepcion1` es una superclase de `Excepcion2` sí existe compatibilidad. El bloque `catch` con el argumento `e2` nunca capturaría la excepción. En lugar debemos codificar como sigue:

```
try {
   ...
}
catch(Excepcion2 e2) {
   ...
}
catch(Excepcion1 e1) {
   ...
}
```

Propagación de Excepciones

Cuando ocurre una excepción en el bloque `try`, el sistema busca un bloque `catch` compatible por asignación, es decir, del mismo tipo de excepción o una superclase, que pueda manejarla, en el siguiente orden:

- En los bloques `catch` que siguen al bloque try.

```
try {
   // La excepción se genera aquí
}
// Primero se prueba si el siguiente bloque catch
// captura la excepción
catch(Excepcion1 e1) {
  ...
```

```
}
// Si la excepción no fue capturada en el bloque Catch anterior,
// se prueba si el siguiente bloque catch captura la excepción
catch(Excepcion2 e2) {
   ...
}
// etc.
```

- En un bloque `try...catch` delimitante.

```
try {
   ...
   try {
     // La excepción se genera aquí
   }
   // Primero se prueba si el siguiente bloque catch
   // captura la excepción
   catch(Excepcion1 e1) {
      ...
   }
}
// Si la excepción no fue capturada en el bloque Catch anterior,
// se prueba si el siguiente bloque catch captura la excepción
catch(Excepcion2 e2) {
   ...
}
```

La búsqueda continúa hasta llegar al bloque que delimita todo el cuerpo del método.

- En el bloque que contiene la llamada al método.

```
void metodo1() {
   ...
   try {
     // La excepción se genera aquí
   }
   // Primero se prueba si el siguiente bloque catch
   // captura la excepción
   catch(Excepcion1 e1) {
      // Si no se captura aquí
   }
}

void metodo2() {
   ...
   try {
      ...
      metodo1()  // La excepción se generó dentro de éste método
      ...
   }
   // Si la excepción no fue capturada en el método donde se generó,
   // se prueba si el siguiente bloque catch captura la excepción
   catch(Excepcion2 e2) {
      ...
```

```
            }
            ...
        }
```

- Si todavía no se detecta un bloque `catch` compatible, el proceso continúa hacia fuera, en los bloques y hacia arriba, en las llamadas a métodos y los bloques limitantes respectivos hasta encontrar un bloque `catch` idóneo.
- Si no se proporciona un manejador de la excepción del programa, es el ambiente Java el que maneja la excepción deteniendo la ejecución del programa.

Ejemplos Sobre Manejo de Excepciones

La clase que implementa una pila, `Stack`, tiene dos métodos `pop()` y `peek()` que lanzan una excepción del tipo `EmptyStackException` cuando queremos extraer un elemento o inspeccionar el tope de una pila vacía. Para mostrar el manejo de excepciones se muestran dos programas que ilustran el comportamiento de una pila. En el primer programa no se maneja la excepción. Esto es posible ya que `EmptyStackException` es una excepción no verificada ya que hereda de la clase `RuntimeException`.

DemoStack.java

```
/*
 * DemoStack.java
 *
 * Creada el 15 de octubre de 2005, 12:36 PM
 */

package queue;

import java.util.Stack;

/**
 * Esta clase muestra que pasa cuando no se captura una excepción,
 * la excepción EmptyStackException, lanzada por algunos
 * métodos de la clase Stack
 *
 * @author mdomitsu
 */
public class DemoStack {

  public static void main(String[] args) {
    DemoStack demoStack1 = new DemoStack();

    // Crea una pila vacía
    Stack stack = new Stack();
    Object o;
    // Despliega el contenido inicial de la pila
    System.out.println(stack);

    // Se inserta una cadena en la pila
```

```
    stack.push("hola");
    // Despliega el contenido de la pila
    System.out.println(stack);

    // Se inserta el entero 10 a la pila
    stack.push(new Integer(10));
    // Despliega el contenido inicial de la pila
    System.out.println(stack);

    // Se extrae el contenido del tope de la pila.
    // No se requiere encerrar la instrucción en un bloque try - catch
    o = stack.pop();
    Integer n = (Integer)o;
    // Despliega el contenido inicial de la pila
    System.out.println(stack + " == > " + n);

    // Se extrae el contenido del tope de la pila.
    // No se requiere encerrar la instrucción en un bloque try - catch
    o = stack.pop();
    String s = (String)o;
    // Despliega el contenido inicial de la pila
    System.out.println(stack + " == > " + s);

    // Se extrae el contenido del tope de la pila.
    // No se requiere encerrar la instrucción en un bloque try - catch
    // La pila está vacía, se lanzará una excepción del tipo
    // EmptyStackException que al no ser capturada será capturada por
    // la máquina virtual, abortando el programa.
    o = stack.pop();
    // Despliega el contenido inicial de la pila
    System.out.println(stack + " == > " + o);  }
}
```

La salida del programa anterior es:

```
[]
[hola]
[hola, 10]
[hola] == > 10
[] == > hola
java.util.EmptyStackException
      at java.util.Stack.peek(Stack.java:79)
      at java.util.Stack.pop(Stack.java:61)
      at queue.DemoStack.main(DemoStack.java:55)
Exception in thread "main"
```

En este caso podemos ver que como la excepción no es atrapada en el método main(), la máquina virtual de Java atrapa la excepción abortando el programa y desplegando el mensaje de error, seguido de la traza del error:

En el siguiente programa, las sentencias en las que se extrae un elemento de la pila se encierran en un bloque `try-cach` para atrapar la excepción:

DemoStack2.java

```
/*
 * DemoStack2.java
 *
 * Creada el 15 de octubre de 2005, 12:36 PM
 */

package queue;

import java.util.Stack;
import java.util.EmptyStackException;

/**
 * Esta clase muestra que pasa cuando se captura una excepción,
 * la excepción EmptyStackException, lanzada por algunos
 * métodos de la clase Stack
 *
 * @author mdomitsu
 */
public class DemoStack2 {

  public static void main(String[] args) {
    DemoStack2 demoStack2 = new DemoStack2();
    // Crea una pila vacía
    Stack stack = new Stack();
    Object o;

    // Despliega el contenido inicial de la pila
    System.out.println(stack);

    // Se inserta una cadena en la pila
    stack.push("hola");
    // Despliega el contenido de la pila
    System.out.println(stack);

    // Se inserta el entero 10 a la pila
    stack.push(new Integer(10));
    // Despliega el contenido de la pila
    System.out.println(stack);

    // Se encierran las instrucciones que pueden lanzar una excepción
    // en un bloque try
    try {
      // Se extrae el contenido del tope de la pila.
      o = stack.pop();
      Integer n = (Integer) o;
      // Despliega el contenido de la pila
      System.out.println(stack + " == > " + n);

      // Se extrae el contenido del tope de la pila.
      o = stack.pop();
      String s = (String) o;
      // Despliega el contenido de la pila
      System.out.println(stack + " == > " + s);

      // Se extrae el contenido del tope de la pila.
```

```
      // La pila está vacía, se lanzará una excepción del tipo
      // EmptyStackException
      o = stack.pop();
      // Despliega el contenido de la pila
      System.out.println(stack + " == > " + o);
    }
    catch(EmptyStackException ese) {
      // Se despliega un mensaje de error como tratamiento a la excepción
      System.out.println("Error: Pila vacía");
    }
  }
}
```

La salida del programa anterior es:

```
[]
[hola]
[hola, 10]
[hola] == > 10
[] == > hola
Error: Pila vacía
```

El ejemplo anterior aunque ilustrativo no es la forma correcta de tratar con el posible error. Lo mejor es tratar de evitar que ocurra la excepción cuando tratamos de extraer un elemento del tope de la pila. Para ello podemos probar si la pila está vacía antes de intentar el elemento mediante el siguiente código:

```
// Se extrae el contenido del tope de la pila.
if(!stack.empty()) {
  String s =  = stack.pop();
  // Despliega el contenido de la pila
  System.out.println(stack + " == > " + o);
}
else {
  // Se despliega un mensaje de error
  System.out.println("Pila vacía");
}
```

Lanzamiento de Excepciones

Si no se desea o no se sabe como manejar una o más excepciones generadas dentro de un método, podemos declarar que el método lanza la excepción o excepciones que no desea o puede manejar para que el método llamante las capture y les dé tratamiento. Para ello se utiliza la sentencia `throws`, cuya sintaxis es la siguiente:

```
public void metodo() throws Excepcion1, Excepcion2, ...{
   ...
   // Aquí se genera una de las excepciones
   ...
}
```

Por ejemplo:

```
public void agrega(Vector v, Object x, int n)
   throws arrayIndexOutOfBoundsException {
   // La siguiente instrucción genera una excepción si n está fuera de
   // los límites permitidos para el vector v: n < 0 o n >= v.size().
   v.insertElementAt(x, n);
}
```

Un método al llamar a otro método puede recibir una excepción lanzada por el método llamado. Si el primer método no desea manejar la excepción, puede a su vez relanzarla para que se capture en otra parte utilizando la sentencia `throws`. Por ejemplo:

```
public void almacena(int pos) throws ArrayIndexOutOfBoundsException {
   Vector v = new Vector();
   int x = 10;

   // La siguiente instrucción genera una excepción si pos está fuera
   // de los límites permitidos para el vector v: pos < 0 o
   // pos >= v.size().
   agrega(v, new Integer(x), pos);
}
```

Lanzamiento de Excepciones en Métodos Abstractos

Si la declaración de un método abstracto, ya sea de una clase abstracta o de una interfaz, establece que el método puede lanzar una excepción, entonces en una implementación de ese método en una clase concreta, la declaración de que lanza una excepción puede omitirse si en dicha implementación no existe la posibilidad de que ocurra dicha excepción o no se desea lanzarla. La razón de ello es que la sentencia throws establece que el método concreto podrá lanzar una excepción, pero esa posiblilidad también acepta el hecho de que no se lance la excepción.

Por ejemplo considera las siguientes fraciones de código de una clase abstracta y de una interfaz:

```
public abstract class Abstracta {
    public abstract void metodoAbstracto1() throws Exception1;

}

public interface Interfaz {
    public void metodoAbstracto2() throws Exception2;

}
```

En la siguiente clase, la implementación de los dos métodos de la clase abstracta y de la interfaz pueden lanzar las excepciones declaradas en los métodos abstractos:

```
public class Implementacion2 extends Abstracta implements Interfaz {
    public void metodoAbstracto1() throws Exception1 {
```

```
            // Aquí puede ocurrir una excepción del tipo Exception1
        }

        public void metodoAbstracto2() throws Exception2 {
            // Aquí puede ocurrir una excepción del tipo Exception2
        }

    }
```

En la siguiente clase, la implementación de los dos métodos de la clase abstracta y de la interfaz no lanzar las excepciones declaradas en los métodos abstractos:

```
    public class Implementacion2 extends Abstracta implements Interfaz {
        public void metodoAbstracto1() {
            // Aquí no ocurre o no se lanza una excepción del tipo Exception1
        }

        public void metodoAbstracto2() {
            // Aquí no ocurre o no se lanza una excepción del tipo Exception2
        }

    }
```

Excepciones Verificadas y No Verificadas

Las excepciones se clasifican en dos grupos: Las excepciones verificadas y las excepciones no verificadas. Una instrucción que puede lanzar una **excepción verificada** debe encerrarse en un bloque `try` o el método que contiene a la instrucción debe declarar que puede lanzar la excepción utilizando la sentencia `throws`. Las excepciones verificadas son aquellas instancias de la clase `Exception` o de sus subclases exceptuando la clase `RuntimeException` o sus subclases.

Una instrucción que puede lanzar una **excepción no verificada** no requiere encerrarse en un bloque `try`. Tampoco se requiere que el método que contiene a la instrucción deba declararse que puede lanzar la excepción utilizando la sentencia `throws`. Las excepciones no verificadas son aquellas instancias de la clase `Error` o sus subclases o instancias de la clase `RuntimeException` o sus subclases. Las excepciones no verificadas no deben tratar de atraparse usando un bloque `try` ya que o no pueden dársele un tratamiento (Clas clase `Error` o sus subclases) o deben de prevenirse (Clase `RuntimeException` o sus subclases).

Generación de Excepciones

Podemos generar una condición de excepción dentro de un método utilizando la instrucción `throw`, cuya sintaxis es:

```
throw instanciaExcepcion
```

El método debe lanzar la excepción mediante la cláusula `throws`. Por ejemplo:

```
public void agrega(Vector v, Object x, int n)
  throws ArrayIndexOutOfBoundsException {
  if(n < 0)
    throw new ArrayIndexOutOfBoundsException("Indice negativo");
  else v.insertElementAt(x, n);
}
```

Creación de Excepciones

Si ninguna de las clases de excepción predefinidas satisface nuestros requerimientos, podemos crear nuestras propias excepciones creando una subclase de alguna clase de excepción.

Se puede obtener una excepción verificada heredando de una excepción verificada

Se puede obtener una excepción no verificada heredando de una excepción no verificada.

Ejemplos Sobre Creación de Excepciones

1. Por ejemplo, el siguiente código crea una excepción no verificada a partir de la excepción `RuntimeException`. Esta excepción será lanzada por la clase Queue que implementa una cola, al querer extraer un elemento de una cola vacía :

EmptyQueueException.java

```
/*
 * EmptyQueueException.java
 *
 * Creada el 15 de octubre de 2005, 12:36 PM
 */

package queue;

/**
 * Esta excepción es lanzada por los métodos de la clase Queue
```

```
 * para indicar que la cola está vacía.
 *
 * @author mdomitsu
 */
public class EmptyQueueException extends RuntimeException {

  /**
   * Construye una excepción del tipo EmptyQueueException sin mensaje de
   * error.
   */
  public EmptyQueueException() {
    super();
  }

  /**
   * Construye una excepción del tipo EmptyQueueException con s como mensaje
   * de error.
   * @param s Cadena con el mensaje de error
   */
   public EmptyQueueException(String s) {
     super(s);
   }
}
```

La clase Queue, que hereda de la clase Vector, implementa una cola. Su código es el siguiente:

Queue.java

```
/*
 * Queue.java
 *
 * Creada el 15 de octubre de 2005, 12:36 PM
 */

package queue;

import java.util.Vector;

/**
 * La clase Queue representa una cola (Primero en entrar - primero en salir)
 * de objetos. Extiende la clase Vector con cinco operaciones que permiten
 * tratar a un vector como una cola. Se tienen las operaciones usuales de
 * enqueue y dequeue así como un método para inspeccionar al elemento en el
 * inicio de la cola, un método para probar si la cola está vacía, y un
 * método para buscar la cola por un elemento y descubrir qué tan lejos está
 * del inicio.
 *
 * Cuando se crea la cola, se crea vacía.
 *
 * @author mdomitsu
 */
public class Queue extends Vector {

  /**
   * Crea una cola vacía
```

```
   */
  public Queue() {
  }

  /**
   * Prueba si la cola está vacía
   * @return true si y sólo sí la cola no tiene elementos; false en caso
   * contrario.
   */
  public boolean empty() {
    return isEmpty();
  }

  /***
   * Inserta un elemento al final de la cola
   * @param item El elemento a insertarse en la cola.
   */
  public void enqueue(Object item) {
    addElement(item);
  }

  /**
   * Extrae un elemento del principio de la cola.
   * @return El elemento extraído
   * @throws EmptyQueueException Cuando la cola está vacía
   */
  public Object dequeue() throws EmptyQueueException {
    Object item;

    if(empty()) throw new EmptyQueueException("Cola vacía");
    else {
      // Obtiene una copia del primer elemento
      item = firstElement();

      // Elimina el primer elemento de la cola
      removeElementAt(0);

      return item;
    }
  }

  /**
   * Inspecciona el elemento del principio de la cola sin extraerlo.
   * @return El elemento inspeccionado
   * @throws EmptyQueueException Cuando la cola está vacía
   */
  public Object peek() throws EmptyQueueException {
    Object item;

    if(empty()) throw new EmptyQueueException("Cola vacía");
    else {
      // Obtiene una copia del primer elemento
      item = firstElement();

      return item;
    }
  }
```

```
  /**
   * Regresa la posición (base 1) de la primera ocurrencia del elemento o en
   * la cola. Si el elemento o existe en la cola, este método regresa la
   * distancia del inicio de la cola de la primera ocurrencia más cercana al
   * principio de la cola.  El elemento al inicio de la cola está a una
   * distancia de 1.
   *
   * @param o Elemento a buscar
   * @return La posición (base 1) del elemento desde el inicio de la cola.
   * Si el elemento no está en la cola el método regresa el valor de -1
   */
  public int search(Object o) {
    return indexOf(o) + 1;
  }
}
```

Para mostrar el manejo de la excepción `EmptyQueueException` se muestra un programa que ilustra el comportamiento de una cola:

DemoQueue.java

```
/*
 * DemoQueue.java
 *
 * Creada el 15 de octubre de 2005, 12:36 PM
 */

package queue;

/**
 * Esta clase prueba la clase Queue y la excepción EmptyStackException
 * lanzada por algunos métodos de la clase Queue
 *
 * @author mdomitsu
 */
public class DemoQueue {
  public static void main(String[] args) {
    DemoQueue demoQueue = new DemoQueue();

    // Crea una cola vacía
    Queue queue = new Queue();
    Object o;

    // Despliega el contenido inicial de la cola
    System.out.println(queue);

    // Se inserta una cadena en la cola
    queue.enqueue("hola");
    // Despliega el contenido de la cola
    System.out.println(queue);

    // Se inserta el entero 10 a la cola
    queue.enqueue(new Integer(10));
    // Despliega el contenido de la cola
    System.out.println(queue);
```

```
    // Se encierran las instrucciones que pueden lanzar una excepción
    // en un bloque try
    try {
      // Se extrae el contenido del frente de la cola.
      o = queue.dequeue();
      String s = (String) o;
      // Despliega el contenido de la cola
      System.out.println(queue + " == > " + s);

      // Se extrae el contenido del frente de la cola.
      o = queue.dequeue();
      Integer n = (Integer) o;
      // Despliega el contenido de la cola
      System.out.println(queue + " == > " + n);

      // Se extrae el contenido del frente de la cola.
      // La cola está vacía, se lanzará una excepción del tipo
      // EmptyQueueException
      o = queue.dequeue();
      // Despliega el contenido de la cola
      System.out.println(queue + " == > " + o);
    }
    catch(EmptyQueueException eqe) {
      // Se despliega un mensaje de error como tratamiento a la excepción
      System.out.println("Error: " + eqe.getMessage());
    }
  }
}
```

La salida del programa anterior es:

```
[]
[hola]
[hola, 10]
[10] == > hola
[] == > 10
Error: Cola vacía
```

2. En el siguiente ejemplo se modificará el ejemplo sobre el fabricante de silos del tema 1: Programación Orientada a objetos, para que los constructores de las clases `SiloCilindro4`, `SiloCono4` y `SiloEsfera4` lancen una excepción del tipo `DimensionNegativaException` si alguna de las dimensiones de los silos es negativa. El siguiente código muestra la implementación de la excepción `DimensionNegativaException`.

DimensionNegativaException.java

```
/*
 * DimensionNegativaException.java
 *
 * Creada el 8 de octubre de 2005, 12:33 PM
 */

package silos;
```

```
/**
 * Esta clase permite excepciones del tipo DimensionesNegativas.
 *
 * @author mdomitsu
 */
public class DimensionNegativaException extends NumberFormatException {
  /**
   * Constructor por omisión.
   */
  public DimensionNegativaException() {
    super();
  }

  /**
   * Constructor que inicializa el atributo s de la clase padre.
   * @param s Mensaje de error
   */
  public DimensionNegativaException(String s) {
    super(s);
  }
}
```

El siguiente código muestra la clase `Silo4` que es la clase padre de las clases `SiloCilindro4`, `SiloCono4` y `SiloEsfera4`. Note que a diferencia de la clase `Silo3` del ejemplo del tema 1, el constructor de la clase no incrementa el atributo estático `contadorSilos` ni inicializa el atributo `numSilo`. Esto se debe a que no queremos que se incremente el contador de silos si queremos crear un silo con alguna dimensión negativa. Como el constructor de la clase padre es lo primero que se invoca en la invocación del constructor de la clase hija, se realizaría el incremento del contador de silos aún cuando en el constructor de la clase hija se lance la excepción al determinar que una de las dimensiones sea negativa. Por eso el incremento del atributo estático `contadorSilos` y la inicialización del atributo `numSilo` deben hacerse hasta que se determine que las dimensiones sean no negativas y por lo tanto su código se pasa a las clases hijas.

Silo4.java

```
/*
 * Silo4.java
 *
 * Creada el 8 de octubre de 2005, 12:36 PM
 */

package silos;

/**
 * Esta clase es la clase padre de las clases SiloCilindro4,
 * SiloCono4 y SiloEsfera4
 *
 * @author mdomitsu
 */
```

```
public class Silo4 {
  protected String tipoSilo;
  protected double superficie;
  protected double volumen;
  protected double costo;
  protected int grosorLamina;
  protected int numSilo;
  protected static int contadorSilos = 0;

  /**
   * Constructor. Inicializa el atributo tipoSilo e incrementa el
   * contador de silos en uno cada vez que se crea un silo.
   * @param tipoSilo Tipo de silo: "Cilindro", "Cono", "Esfera"
   * @param grosorLamina Grosor de la lámina del silo cilíndrico
   */
  public Silo4(String tipoSilo, int grosorLamina) {
    // Inicializa los atributos
    this.tipoSilo = tipoSilo;
    this.grosorLamina = grosorLamina;
  }

  /**
    * Regresa una cadena con la representación de esta clase
    * @return Una cadena con la representación de esta clase
    */
   public String toString() {
     return numSilo + ": " + tipoSilo + ", Grosor lámina: " + grosorLamina;
   }
}
```

Las siguientes tres clases son las clases `SiloCilindro4`, `SiloCono4` y `SiloEsfera4` que heredan de la clase `Silo4` e implementan la interfaz `ISilo` vista en el Tema 1: programación Orientada a Objetos. Note que en los constructores de las tres clases se lanza una excepción del tipo `DimensionNegativaException` si alguna de las dimensiones de los silos es negativa y que en caso contrario inicializa los atributos de la clase, incrementa el atributo estático `contadorSilos`, inicializa el atributo `numSilo`, etc.

SiloCilindro4

```
/*
 * SiloCilindro4.java
 *
 * Creada el 8 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCilindro4
 *
 * @author mdomitsu
 */
public class SiloCilindro4 extends Silo4 implements ISilo {
  private double radio;
```

```
  private double altura;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo cilíndrico
   * @param altura Altura del silo cilíndrico
   */
  public SiloCilindro4(double radio, double altura, int grosorLamina) {
    // Llama al constructor de la clase padre: Silo4
    super("Silo Cilíndrico", grosorLamina);

    // Verifica que las dimensiones sean positivas
    if(radio < 0.0 || altura < 0.0) throw new
               DimensionNegativaException("Error: Radio o altura negativas");

    // Inicializa los atributos
    this.radio = radio;
    this.altura = altura;

    // Incrementa el contador de silos creados
    contadorSilos++;

    // Le asigna un número a este silo
    numSilo = contadorSilos;
  }

  /**
   * Calcula la superficie del cilindro
   */
  public void calculaSuperficie() {
    superficie = 2*Math.PI*radio*(radio+altura);
  }

  /**
   * Calcula el volumen del cilindro
   */
  public void calculaVolumen() {
    volumen = Math.PI*radio*radio*altura;
  }

  /**
   * Calcula el precio del silo cilíndrico
   */
  public void calculaCosto() {
    costo = superficie * Costos.get(grosorLamina);
  }

  /**
    * Regresa una cadena con la representación de esta clase
    * @return Una cadena con la representación de esta clase
   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", altura = " + altura
        + ", superficie = " + superficie + ", volumen = " + volumen
        + ", costo: " + costo;
  }
}
```

SiloCono4

```
/*
 * SiloCono4.java
 *
 * Creada el 8 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCono4
 *
 * @author mdomitsu
 */
public class SiloCono4 extends Silo4 implements ISilo {
  private double radio;
  private double altura;
  private int tipoBase;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo coníco
   * @param altura Altura del silo coníco
   * @param tipoBase Tipo de la base del silo cónico
   * @param grosorLamina Grosor de la lámina del silo cónico
   */
  public SiloCono4(double radio, double altura, int tipoBase,
                   int grosorLamina) {
    // Llama al constructor de la clase padre: Silo4
    super("Silo Cónico", grosorLamina);

    // Verifica que las dimensiones sean positivas
    if(radio < 0.0 || altura < 0.0) throw new
               DimensionNegativaException("Error: Radio o altura negativas");

    // Inicializa los atributos
    this.radio = radio;
    this.altura = altura;
    this.tipoBase = tipoBase;

    // Incrementa el contador de silos creados
    contadorSilos++;

    // Le asigna un número a este silo
    numSilo = contadorSilos;
  }

  /**
   * Calcula la superficie del cono
   */
  public void calculaSuperficie() {
    superficie = Math.PI*radio*(radio + Math.sqrt(radio*radio +
                 altura*altura));
  }

  /**
```

```
   * Calcula el volumen del cono
   */
  public void calculaVolumen() {
    volumen = Math.PI*radio*radio*altura/3;
  }

  /**
   * Calcula el precio del silo cónico
   */
  public void calculaCosto() {
    costo = Costos.get(tipoBase) + superficie * Costos.get(grosorLamina);
  }

  /**
    * Regresa una cadena con la representación de esta clase
    * @return Una cadena con la representación de esta clase
   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", altura = " + altura
        + ", superficie = " + superficie + ", volumen = " + volumen
        + ", costo: " + costo;
  }
}
```

SiloEsfera4

```
/*
 * SiloEsfera4.java
 *
 * Creada el 8 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloEsfera4
 *
 * @author mdomitsu
 */
public class SiloEsfera4 extends Silo4 implements ISilo {
  private double radio;
  private int tipoBase;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo esférico
   */
  public SiloEsfera4(double radio, int tipoBase, int grosorLamina) {
    // Llama al constructor de la clase padre: Silo4
    super("Silo Esférico", grosorLamina);

    // Verifica que el radio sea positivo
    if(radio < 0.0) {
      throw new DimensionNegativaException("Error: Radio negativo");
    }

    // Inicializa los atributos
```

```
    this.radio = radio;
    this.tipoBase = tipoBase;

    // Incrementa el contador de silos creados
    contadorSilos++;

    // Le asigna un número a este silo
    numSilo = contadorSilos;
  }

  /**
   * Calcula la superficie de la esfera
   */
  public void calculaSuperficie() {
    superficie = 4*Math.PI*radio*radio;
  }

  /**
   * Calcula el volumen de la esfera
   */
  public void calculaVolumen() {
    volumen = 4*Math.PI*radio*radio*radio/3;
  }

  /**
    * Calcula el precio del silo esférico
    */
   public void calculaCosto() {
     costo = Costos.get(tipoBase) + superficie * Costos.get(grosorLamina);
   }

  /**
    * Regresa una cadena con la representación de esta clase
    * @return Una cadena con la representación de esta clase
    */
   public String toString() {
     return super.toString() + ", radio = " + radio + ", superficie = "
         + superficie + ", volumen = " + volumen + ", costo: " + costo;
   }
}
```

Para probar las clases anteriores se implementa la siguiente clase de prueba:

PruebaSilo4

```
/*
 * PruebaSilo4.java
 *
 * Creada el 7 de octubre de 2005, 12:45 PM
 */

package silos;

/**
 * Esta clase se utiliza para probar las clases SiloCilindro4,
 * SiloCono4 y SiloEsfera4
 *
```

```
 * @author mdomitsu
 */
public class PruebaSilo4 {

  /**
   * Método main() en el que se invocan a los métodos de las clases
   * SiloCilindro4, SiloCono4 y SiloEsfera4 para probarlos
   * @param argumentos Los argumentos en la línea de comando
   */
  public static void main(String[] args) {
    PruebaSilo4 pruebaSilo4 = new PruebaSilo4();
    ISilo silo[] = new ISilo[5];
    int i = 0;

    // Intenta crear el primer silo
    try {
      silo[i]=new SiloCilindro4(1.0, 1.0, Costos.LAMINA_CAL12);
      i++;
    }
    catch(DimensionNegativaException dne) {
      System.out.println(dne.getMessage());
    }

    // Intenta crear el segundo silo
    try {
      silo[i]=new SiloEsfera4(-1.0, Costos.BASE_SIMPLE, Costos.LAMINA_CAL12);
      i++;
    }
    catch(DimensionNegativaException dne) {
      System.out.println(dne.getMessage());
    }

    // Intenta crear el tercer silo
    try {
      silo[i]=new SiloCono4(1.0, 1.0, Costos.BASE_SIMPLE,
                            Costos.LAMINA_CAL12);
      i++;
    }
    catch(DimensionNegativaException dne) {
      System.out.println(dne.getMessage());
    }

    // Intenta crear el cuarto silo
    try {
      silo[i]=new SiloCilindro4(-2.0, 1.0, Costos.LAMINA_CAL14);
      i++;
    }
    catch(DimensionNegativaException dne) {
      System.out.println(dne.getMessage());
    }

    // Intenta crear el quinto silo
    try {
      silo[i]=new SiloEsfera4(2.0, Costos.BASE_REFORZADA,
                              Costos.LAMINA_CAL16);
      i++;
    }
```

```
    catch(DimensionNegativaException dne) {
      System.out.println(dne.getMessage());
    }

    // Para cada silo creado
    for (i = 0; i < Silo4.contadorSilos; i++) {
        // Calcula la superficie del silo
        silo[i].calculaSuperficie();

        // Calcula el volumen del silo
        silo[i].calculaVolumen();

        // Calcula el costo del silo
        silo[i].calculaCosto();
    }

    // Para cada silo creado
    for(i = 0; i < Silo4.contadorSilos; i++)
       // Escribe los valores de sus atributos
      System.out.println(silo[i]);

    // Escribe el número de silos creados
    System.out.println("Silos creados: " + Silo4.contadorSilos);
  }
}
```

Envolvimiento de Excepciones y Encadenamiento de Excepciones

Una práctica común en el desarrollo de aplicaciones es construirlas en capas donde la capa superior utiliza a la capa inferior para llevar a cabo su trabajo. Por ejemplo, la capa superior de una publicación podría ser la interfaz con el usuario, la capa intermedia podría contener la lógica del programa, mientras que la capa inferior podría ser la encargada de almacenar los datos permanentemente. En estos casos, si ocurre un error en el código de la capa inferior y deseamos que el usuario se entere del error, podemos hacer que el código de la capa inferior lance una excepción que será relanzada por el código de la capa intermedia al código de la capa superior para su tratamiento. Un problema que tiene esta solución, sin embargo, es que con frecuencia los mensajes de error de las excepciones lanzadas por el código de la capa inferior sean muy técnicos y que aunque provea información útil al desarrollador que esté depurando la aplicación, no sean claros ni útiles para el usuario final de la aplicación.

Una solución en estos casos, llamada **Envolvimiento de Excepciones**, es hacer que los métodos que llamen al código de la capa inferior capturen las excepciones lanzadas por éste y relancen una nueva excepción con un mensaje de error más amigable a la capa superior, tal como se ilustra en el siguiente fragmento de código:

```
// Esta clase contiene el código de la capa inferior de la aplicación
public class Inferior {
  ...

  // Este método contiene el código que genera la excepción
  public void metodoInferior() throws ConfusaException {
    ...
    // Aquí se genera la excepción con un mensaje de error confuso
    // para el usuario final
    ...
  }
  ...
}

// Esta clase contiene el código de la capa intermedia de la aplicación
public class Intermedia {
  Inferior inferior = new Inferior();
  ...

  // Este método llama a un método de la capa inferior
  public void metodoIntermedio() throws new AmistosaException {
    ...

    try {
      // Aquí se llama al método de la capa inferior que lanzó
      // la excepción
      inferior.metodoInferior();
    }
    catch(ConfusaException ce) {
      // Aquí se captura la excepción generada en la capa inferior
      // y se relanza como una excepción más amigable
      throw new AmistosaException(mensajeAmistoso);
    }
    ...
  }
  ...
}

// Esta clase contiene el código de la capa superior de la aplicación
public class Superior {
  Intermedia intermedia = new Intermedia();
  ...

  // Este método llama a un método de la capa intermedia
  public void metodoSuperior() {
    ...

    try {
      // Aquí se llama al método de la capa intermedia que relanzó
      // la excepción generada en la capa inferior
      intermedia.metodoIntermedio();
    }
    catch(AmistosaException ae) {
      // Aquí se le dará tratamiento al error ocurrido en la capa
      // inferior, desplegando el mensaje amistoso en lugar del confuso
      System.out.println(ae.getMessage());
    }
```

```
    ...
  }
  ...
}
```

El código anterior muestra que el usuario final ve un mensaje de error más amistoso que el de la excepción generado original.

La solución anterior, aunque adecuada para el usuario final, no es conveniente para el desarrollador que depura la aplicación ya que si ocurre un error en la capa inferior y si trata de determinar dónde ocurrió el error, por ejemplo mediante el método de la excepción `printStackTrace()`, la traza de las llamadas a métodos por las que siguió la aplicación hasta llegar al momento de generarse la excepción termina en el método de la capa intermedia donde se envolvió la excepción y no donde ocurrió el error.

La solución a este problema, llamada **Encadenamiento de Excepciones**, consiste en incluir la causa del error en la excepción envolvente utilizando un constructor que aparte de recibir como argumento el mensaje de error, recibe también un objeto del tipo Throwable, la excepción original como se ilustra en el siguiente código:

```
// Esta clase contiene el código de la capa inferior de la aplicación
public class Inferior {
  ...

  // Este método contiene el código que genera la excepción
  public void metodoInferior() throws ConfusaException {
    ...

    // Aquí se genera la excepción con un mensaje de error confuso
    // para el usuario final
    ...
  }
  ...
}

// Esta clase contiene el código de la capa intermedia de la aplicación
public class Intermedia {
  Inferior inferior = new Inferior();
  ...

  // Este método llama a un método de la capa inferior
  public void metodoIntermedio() throws AmistosaException {
    ...

    try {
      // Aquí se llama al método de la capa inferior que lanzó
      // la excepción
      inferior.metodoInferior();
    }
    catch(ConfusaException ce) {
      // Aquí se captura la excepción generada en la capa inferior
      // y se relanza como una excepción más amigable más la
```

```
        // causa del error, la excepción original
        throw new AmistosaException(mensajeAmistoso, ce);
      }
      ...
    }
    ...
  }

  // Esta clase contiene el código de la capa superior de la aplicación
  public class Superior {
    Intermedia intermedia = new Intermedia();
    ...

    // Este método llama a un método de la capa intermedia
    public void metodoSuperior() {
      ...

      try {
        // Aquí se llama al método de la capa intermedia que relanzó
        // la excepción generada en la capa inferior
        intermedia.metodoIntermedio();
      }
      catch(AmistosaException ae) {
        // Aquí se le dará tratamiento al error ocurrido en la capa
        // inferior, desplegando el mensaje amistoso en lugar del
        // confuso.
        System.out.println(ae.getMessage());

        // El desarrollador puede depurar el código desplegando la traza
        // de las llamadas de los métodos hasta en punto en que ocurrió
        // el error, ya que el método printStackTrace está diseñado para
        // inspeccionar la información de una excepción encapsulada en
        // otra excepción.
        System.out.println(ae.printStackTrace());
      }
      ...
    }
    ...
  }
```

El código anterior muestra que el usuario final ve un mensaje de error más amistoso que el de la excepción generado original, pero que el desarrollador tiene acceso a la información de la excepción original.

Consideraciones sobre el Uso de Excepciones

Si no existe forma posible de manejar un posible error, genérese una excepción.
Si es factible dar un manejo sencillo a un error, debe hacerse sin generar una excepción.

Las excepciones deben reservarse para circunstancias excepcionales ya que el tiempo requerido para generar y manejar las excepciones es mayor al requerido para dar manejo local a los problemas con instrucciones if.