Programación orientada a objetos

Capítulo 12

Manejo de errores

Tema 12: Manejo de errores. Semana 12

1- Principios del lanzamiento de excepciones. a. Lanzar una excepción. b. Las Clases Exception. c. El efecto de una excepción. d. Excepciones no comprobadas. e. Impedir la creación de un objeto. 2- Manejo de excepciones. a. Excepciones comprobadas: la cláusula throws. b. Captura de excepciones: la sentencia try. c. Lanzamiento y comprobación de excepciones. d. Propagación de excepciones. e. La cláusula finally. 3- Definición de nuevas clases de excepciones. 4- Usar aserciones. 5- Recuperarse del error y anularlo. 6- Estudio de caso: entrada/salida de texto.	1- Estudiar el capítulo 12 del libro base para la "Unidad Didáctica II". 2- Realizar los ejercicios correspondientes del libro base. 3- Incorporar el control de errores y excepciones en la práctica.

Conceptos

- Programación a la defensiva
- Anticiparse a lo que podría ir mal
- Lanzamiento y tratamiento de excepciones
- Aserciones

Causas de situaciones de error

- Implementación incorrecta
  - No se ajusta a las especificaciones
- Petición de objeto inapropiada
  - Índice inválido
  - Referencia nula
- Estado de objeto inapropiado o inconsistente
  - P.ej. debido a una extensión de la clase
- No siempre se trata de errores de programación
  - Hay errores que vienen del entorno:
    - URL incorrecta
  - Interrupción de las comunicaciones de red
  - El procesamiento de ficheros suele ocasionar errores:
    - Ficheros que no existen
    - Permisos inapropiados

12.2 Programación a la defensiva

- 12.2.1 Iteración cliente-servidor

- Pueden asumir que los objetos cliente sabrán lo que están haciendo y requerirán servicios sólo de una manera sensata y bien definida.
- Pueden asumir que el servidor operará en un ambiente esencialmente hostil, en el que se deben tomar todas las medidas posibles para prevenir que los objetos cliente usen el servidor incorrectamente.
- ¿Cuántas verificaciones de las solicitudes del cliente deben realizar los métodos del servidor?
- ¿Cómo debe informar el servidor, los errores a sus clientes?
- ¿Cómo puede un cliente anticipar un fallo en una solicitud al servidor?
- ¿Cómo puede tratar un cliente el fallo de una solicitud?

12.2.2 Comprobación de parámetros

- Los argumentos son el primer punto de vulnerabilidad de un objeto servidor
  - Los argumentos de un constructor inicializan el estado
  - Los argumentos de los métodos contribuyen a la evolución del comportamiento
- La comprobación de argumentos es una medida defensiva

Comprobación de la Clave

```
public void removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```

12.3 Generación de informes de error del servidor

- ¿Cuál es la mejor manera de que un servidor informe de un problema?
- Primera aproximación
  - **Notificar los errores al usuario**, emitiendo mensajes de error usando “System.out”
    - ¿Hay un usuario humano?
    - ¿Podría resolver el problema?
      - El usuario puede no saber que tiene que hacer antes ciertos errores
- **Notificar al objeto cliente**
  - El servidor puede usar el valor de retorno de un método para devolver una bandera que indique si fue exitosos o no la llamada al método
    - El cliente comprueba si todo ha ido bien o no según lo que devuelva el método llamado

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

- El servidor **lanza una excepción** desde el método servidor si algo anda mal

12.4 Principios de generación de excepciones

- En lenguajes de POO modernos se usan excepciones, el código se estructura:
- En el código del objeto servidor se puede indicar la ocurrencia de una excepción y esto ocasiona la finalización inmediata del método
  - Por ejemplo el incumplimiento de alguna condición necesaria para la ejecución del método
  - También pueden ocurrir excepciones del entorno (no originadas explícitamente por la aplicación)
    - Síncronas:
      - División por cero, acceso fuera de los límites de un array, puntero nulo
    - Asíncronas:
      - Algún fallo del sistema, caída de comunicaciones, falta de memoria
- El código del cliente se puede estructurar en dos partes:
  - Secuencia normal de ejecución
  - Tratamiento de excepciones:
    - Qué hacer cuando se sale del flujo normal de ejecución por la ocurrencia de alguna excepción

12.4.1 Generación de excepciones

El lanzamiento de una excepción tiene dos etapas: primero se crea un objeto excepción (en este caso un objeto `NullPointerException`) y luego se lanza el objeto excepción usando la palabra clave `throw`. Estas dos etapas se combinan casi invariablemente en una única sentencia:

```
throw new TipoDeExcepcion ("cadena opcional de diagnóstico");
```

Una **excepción** es un objeto que representa los detalles de un fallo de un programa. Se lanza una excepción para indicar que ha ocurrido un fallo.

```
/**
   * Busca un nombre o un número de teléfono y devuelve
los
   * datos del contacto correspondiente.
   * @param clave El nombre o número a buscar.
   * @return Los datos correspondientes a la clave o null
   *         si no hay coincidencias.
   * @throws NullPointerException  si la clave es null.
   */
public DatosDelContacto getContacto(String clave)
{
    if(clave == null) {
      throw new NullPointerException(
        "clave null en getContacto");
    }
    return libreta.get(clave);
}
```

Ejemplo

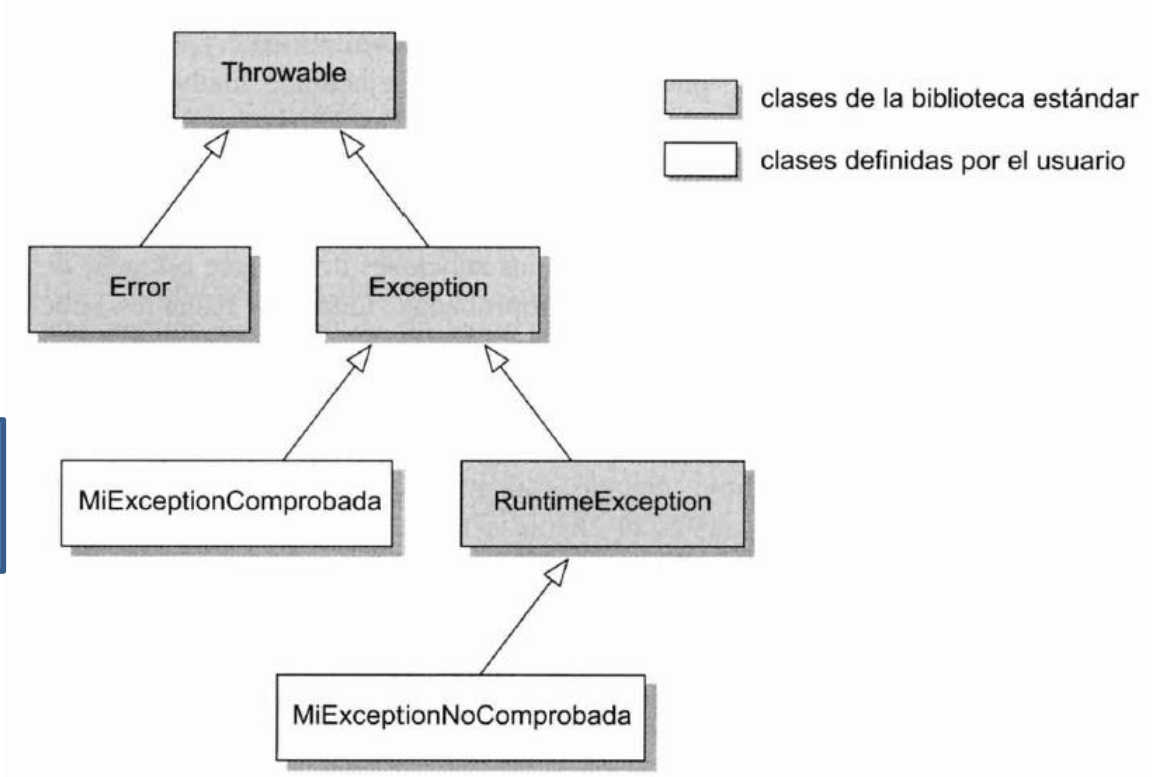
```
public void volver( ) throws Exception {

      if ( lastRoom == null) {
            // imposible... lastRoom nunca debería ser null
            throws new Exception( );
      } else {
            // El código sigue ejecutándose normalmente
            Room tmp = currentRoom;
            currentRoom = lastRoom;
            lastRoom = tmp;
      }
}
```

- Cuando ocurre una excepción se dice que se *lanza (throw)*
  throw *new Excepción();*
- La excepción puede ser capturada para tratarla *(catch)*
  catch *(Excepción e) { tratamiento(); }*

12.4.2 Excepciones comprobadas y no comprobadas: Clases “Exception”

- Toda excepción en Java es una subclase de la clase Throwable, que a su vez está dividida en:
- Error:
  - Representan fallos de tipo catastrófico generalmente no controlados, que originan la parada del programa en ejecución. Ej:**OutOfMemoryError**
- Exception:
  - Representan excepciones que deben capturarse y tratarse. Una subclase de Exception es **RuntimeException**, de la que heredan excepciones como **ArithmeticException**, **IndexOutOfBoundsException**, **NullPointerException**, etc.
- Las excepciones son objetos de la clase *Exception,* que hereda de *Throwable*
- En Java puede haber dos tipos de excepciones:
  - Excepciones que no requieren comprobarse
    - Errores y excepciones de ejecución
      - **Clase *RuntimeException***
  - Excepciones que hay que comprobar
    - Todas las demás
      - **Heredan de la clase *Exception***

Java divide las clases de excepciones en dos categorías: *excepciones comprobadas y no comprobadas*. Todas las subclases de la clase estándar de Java `RunTimeException` son excepciones no comprobadas, todas las restantes subclases de `Exception` son excepciones comprobadas.



Excepciones *comprobadas*: Cuando El cliente debe esperar que una operación falle

Excepciones *No comprobadas*: fallos del programa

Categorías de excepción

- Excepciones sin comprobación obligatoria
  - Subclases de `RuntimeException`
  - Son las que puede lanzar la MVJ
    - Normalmente representan una condición fatal del programa
    - No las comprueba el compilador y es difícil saber cuándo y por qué pueden suceder
    - Pero a veces se puede considerar que alguna condición podría ocasionarlas
    - Se suelen tratar en algún nivel superior de forma genérica
- Excepciones con comprobación
  - Subclases de `Exception`
  - Las comprueba el compilador
    - Si no se consideran en el código, el compilador indica un error
    - Estas excepciones son lanzadas por métodos que se usan en el código (por eso las reconoce el compilador)

Algunas excepciones comunes en Java

- No comprobadas, subclases de *RuntimeException*:
  - NullPointerException
    - Cuando se envía un mensaje a un objeto null
  - ArrayIndexOutOfBoundsException
    - Cuando se accede a un índice ilegal en un array
- Comprobadas:
  - IOException
    - Clase genérica para las excepciones que se producen en operaciones de E/S
  - NoSuchMethodException
    - Cuando no se encuentra un método
  - ClassNotFoundException
    - Cuando una aplicación intenta cargar una clase pero no se encuentra su definición (el fichero .class correspondiente)

12.4.3 Efectos de una excepción

¿Qué ocurre cuando se lanza una excepción? En realidad, hay dos efectos a considerar: el efecto en el método en que se lanzó la excepción y el efecto en el invocador.

```
if (key == null) {
   throw new NullPointerException("clave null en getContacto");
}
else {
   return libreta.get(clave);
}
```

La ausencia de una sentencia `return` en la ruta en que se dispara una excepción es aceptable. En su lugar, el compilador indicará un error si se han escrito sentencias a continuación de la sentencia *throw* porque podrían no ejecutarse nunca.

Si no se captura una excepción, el programa terminará indicando el problema detectado

- Efectos de una excepción
  - El método lanzado acaba prematuramente
  - No se devuelve ningún valor de retorno
  - El control no vuelve al punto de llamada del cliente
    - Con lo cual el cliente no puede despreocuparse
    - Un cliente puede capturar (“catch”) una excepción

12.4.4 Excepciones no comprobadas

Las **excepciones no comprobadas** son un tipo de excepción cuyo uso no requerirá controles por parte del compilador.

```
    /**
     * Busca un nombre o un número de teléfono y devuelve
     * los datos de contacto correspondientes.
     * @param clave El nombre o el número que a
buscar.
     * @throws   NullPointerException   si la clave es
null.
     * @throws   IllegalArgumentException   si la clave
está vacía.
     * @return   Los datos correspondientes a la clave
dato o
     *           null si no hay ninguna coincidencia.
     */
    public DatosDelContacto getContacto(String clave)
    {
   if(clave == null) {
                        throw new NullPointerException(
"clave null en getContacto");
           }
           if(clave.trim().length() == 0){
                 throw new IllegalArgumentException(
"Se pasó clave vacía a getContacto");
           }
           return libro.get(clave);
     }
```

12.4.5 Cómo Impedir la creación de un objeto

- Un uso importante de las excepciones, es impedir que se creen objetos cuando no se les puede preparar con un estado inicial válido
- Comprobamos que no sean nulos los campos fundamentales
- Si son nulos, lanzamos una excepción para impedir la creación "inconsistente" de objetos

```
/**
 * Prepara los datos del contacto. A todos los datos se
 * les elimina los espacios en blanco al comienzo y al
final.
 * El nombre y el teléfono no pueden ser simultáneamente
cadenas vacías.
 * @param nombre El nombre.
 * @param telefono El número de teléfono.
 * @param direccion La dirección.
 * @throws   Illegal StateException   Si el nombre y el
teléfono están vacíos.
 */
public DatosDeContacto(String nombre, String telefono,
String direccion)
{
      // Usa cadenas vacías si alguno de los argumentos
es null.
      if(nombre == null) {
            nombre = "";
      }
      if(telefono == null) {
            telefono = "";
      }
      if(direccion == null) {
            direccion = "";
      }
      this.nombre = nombre.trim();
      this.telefono = telefono.trim();
      this.direccion = direccion.trim();

      if(this.nombre.length() == 0 && this.telefono.length()
== 0) {
            throw new IllegalStateException(
                        "El nombre y el teléfono no
pueden estar vacíos.");
            }
      }
```

12.5 Manejo de excepciones

Las **excepciones comprobadas** son un tipo de excepción cuyo uso requiere controles adicionales del compilador. En particular, las excepciones comprobadas en Java requieren el uso de cláusulas *throws* y de sentencias *try*.

12.5.1 Manejo de excepciones comprobadas: clausula "throws"

- El manejo de las excepciones es requerida cuando se tratan de excepciones comprobadas

El primer requerimiento del compilador es que un método que lanza una excepción comprobada debe declarar que lo hace mediante una *cláusula throws* que se agrega en su encabezado. Por ejemplo, un método que lanza una `IOException` comprobada del paquete `java.io` debe tener el siguiente encabezado[3]:

```
public void grabarEnArchivo(String archivoDestino)
  throws IOException
```

12.5.2 Captura de excepciones: sentencia “try”

El segundo requerimiento es que el invocador de un método que lanza una excepción comprobada debe proveer un tratamiento para dicha excepción. Esto generalmente implica escribir un *manejador de excepción* bajo la forma de una *sentencia* `try`. Las

El código de un programa que protege sentencias que podrían lanzar una excepción se denomina **manejador de excepción**. El código proporciona información y/o código para recuperarse del error.

```
try {
   Aquí se protege una o más sentencias.
}
catch(Exception e) {
   Aquí se informa y se recupera de la excepción
}
```

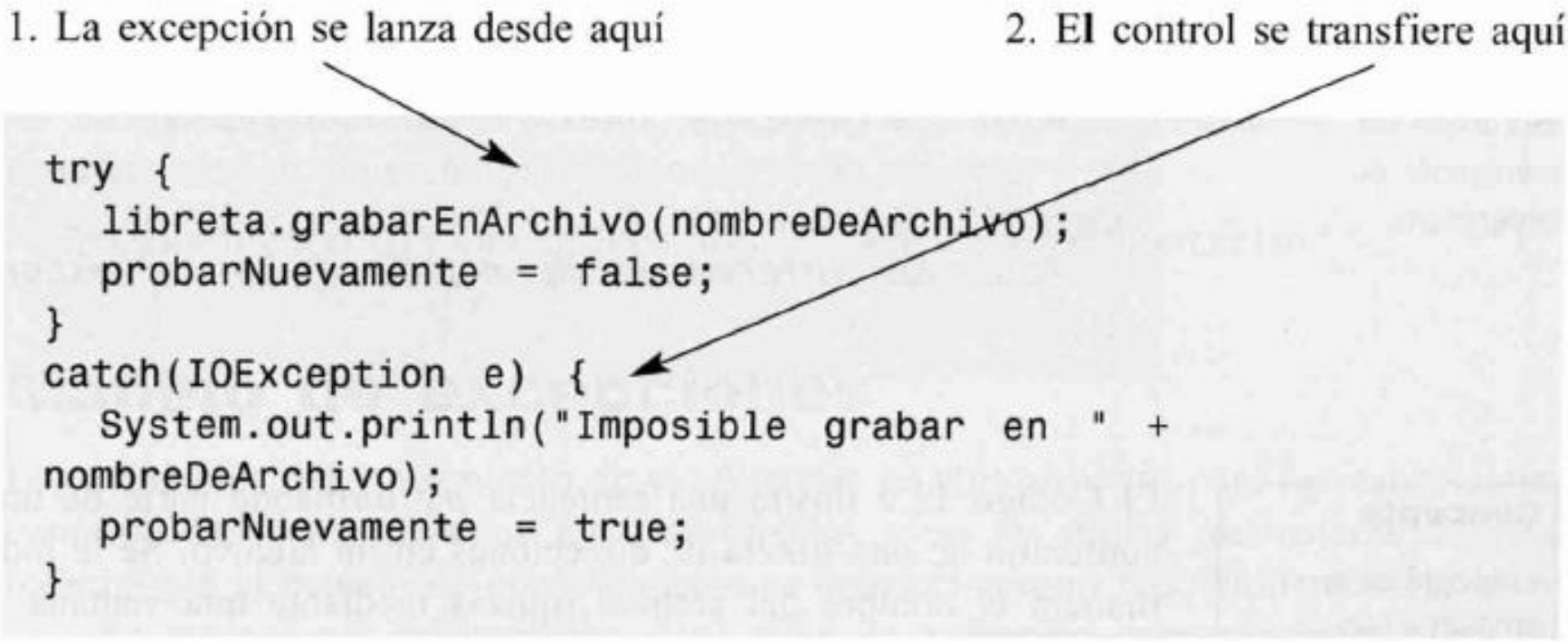
```
String nombreDeArchivo = null;
try {
   nombreDeArchivo = nombre-que-se-pide-al-usuario;
   libreta.grabarEnArchivo(nombreDeArchivo);
}
catch(IOException e) {
   System.out.println("Imposible grabar en " +
nombreDeArchivo);
}
```

Las sentencias try/catch/finally

- Las sentencias try/catch/finally permiten capturar y resolver un problema que ha generado una excepción.
  - **try** (*Intentar***)**. Define un bloque de código que se intenta ejecutar, y en el que podrían generarse excepciones.
  - **Catch** (*Capturar***)**. Define un bloque de código a ejecutar si se captura alguna excepción. Pueden existir varios bloques catch.
  - **finally** (*Finalmente***)**. Una vez ejecutado el código especificado por try y/o catch, en este bloque se incluye código que se ejecuta **siempre**, independientemente de que se haya producido una excepción o no. Este bloquees opcional, puede no incluirse un bloque finally.

Transferencia de control en una sentencia “try”

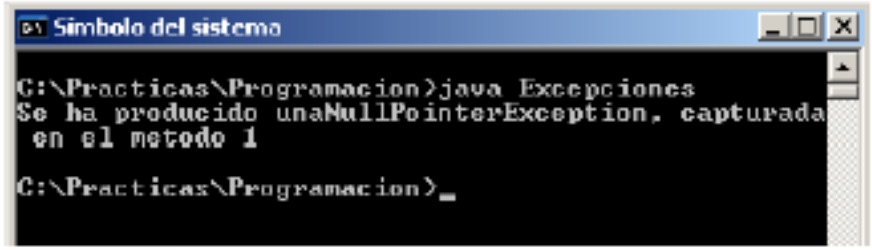
Las sentencias ubicadas dentro de un bloque *try* se conocen como *sentencias protegidas*. Si no se dispara ninguna excepción durante la ejecución de las sentencias protegidas, entonces se salteará el bloque *catch* cuando se llegue al final del bloque *try*. La ejecución continuará con cualquier sentencia que esté a continuación de la sentencia *try* completa.



Gestión excepción

- Al producirse un error en un método se genera un objeto que representa el error (**Excepción**).
- Si el error se genera en un método *m*, la JVM busca un gestor adecuado dentro del propio método.
- **Si el gestor existe**, cederá el control a dicho gestor
- **Si el gestor no existe**, buscará el gestor en el método que haya invocado al método *m*, y así sucesivamente, hasta encontrar un gestor capaz de tratar la excepción producida

```java
public class Excepciones {
    public static void main(String[] args) {
        try {
            metodo1();
        } catch(NullPointerException npe) {
            System.out.println("Se ha producido una" +
                "NullPointerException, capturada en el main");
        }
    }

    public static void metodo1() {
        try {
            metodo2();
        } catch(NullPointerException npe) {
            System.out.println("Se ha producido una" +
                "NullPointerException, capturada en el metodo 1");
        }
    }

    public static void metodo2() {
        metodo3();
    }

    public static void metodo3() {
        Object a = null;
        a.toString();   // Esto generará una NullPointerException
    }
}
```



12.5.3 Lanzar y capturar varias excepciones

Algunas veces, un método lanza más de un tipo de excepción para indicar diferentes tipos de problemas. Cuando se trate de excepciones comprobadas deben enumerarse todas en la cláusula *throws* del método, separadas por comas. Por ejemplo:

```
public void procesar()
  throws EOException, FileNotFoundException
```

```
try {
      ...
      ref.procesar();
      ...
}
catch(EOException e) {
      // Tomar las medidas apropiadas para una excepción
fin-de-archivo
      ...
}
catch(FileNotFoundException e) {
      // Tomar las medidas apropiadas para una excepción
archivo-no-encontrado
      ...
}
```

Tiene que haber un bloque catch por cada tipo de excepción

Los bloque catch se evalúan en el orden que están escritos

Multiples catch

- Se pueden utilizar múltiples bloques de sentencias **catch** en el mismo bloque de sentencias **try**,
- Cada bloque gestionará un tipo de excepción.
- El **orden** en el que se colocan las sentencias catch es relevante,
  - Los bloques catch de excepciones genéricas deberían situarse después de los de excepciones más particulares.
    - La JVM busca en orden descendente un bloque catch que sea capaz de controlar la excepción producida, una vez que lo encuentra no sigue buscando.
      - Por ejemplo, si se incluyen dos bloques catch, uno que capture Exception y otro que capture NullPointerException, este último deberá colocarse el primero porque de lo contrario nunca podría llegar a ejecutarse.

Capturar todas las excepciones en un solo bloque “catch” usando la clase “exception” que incluye a todas. En el catch el orden tiene que ser de la mas particular a la mas general

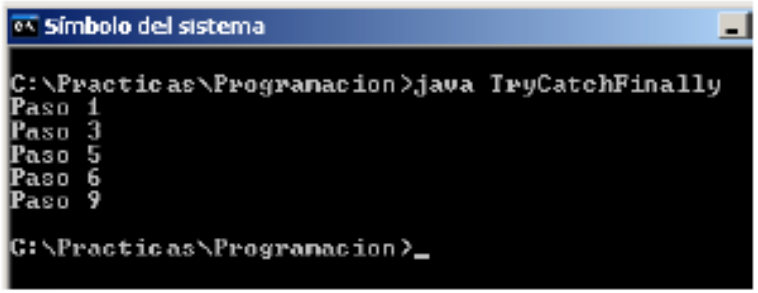
```
try {
  ...
   ref.procesar();
  ...
   }
catch(Exception e) {
  // Tomar las medidas adecuadas para todas las
excepciones
  ...
}
```

Ejercicio 12.30 ¿Qué está mal en la siguiente sentencia *try*?

```
try {
   Persona p = baseDeDatos.buscar(datos);
   System.out.println(_Los datos pertenecen a: _ + p);
}
catch(Exception e) {
   // Maneja cualquiera de las excepciones comprobadas
   ...
}
catch(RuntimeException e) {
   // Maneja cualquiera de las excepciones no comprobadas
   ...
}
```

```
try  {
      System.out.println("Vamos a ejecutar un método");
      unMetodo( );
} catch( IOException e)  {
      /* este bloque se ejecuta si se produjo una IOException */
      (...)
} catch(NullPointerException e) {
      /* este bloque se ejecuta si se produjo
       una NullPointerException */

} catch(Exception e) {
      /* por último, podríamos comparar si
      hay algún tipo de excepción que no es de ninguno
      de los tipos anteriores.  Esto funciona porque
       Exception es la excepción más general[11]... pero tiene que
      ser la última comparación */
}
```

```java
public class TryCatchFinally {
    public static void main(String[] args) {
        try {
            System.out.println("Paso 1");
            int a = 10 / 0;      // Lanza una ArithmeticException
            System.out.println("Paso 2");
        } catch(ArithmeticException ae) {
            System.out.println("Paso 3");
        } catch (Exception e) {
            System.out.println("Paso 4");
        }

        try {
            System.out.println("Paso 5");
            int a = 10 / 1;      // NO lanza una excepción
            System.out.println("Paso 6");
            Object b = null;
            b.toString();        // Lanza una NullPointerException
            System.out.println("Paso 7");
        } catch(ArithmeticException ae) {
            System.out.println("Paso 8");
        } catch (Exception e) {
            System.out.println("Paso 9");
        }
    }
}
```



12.5.5 La clausula “finally”

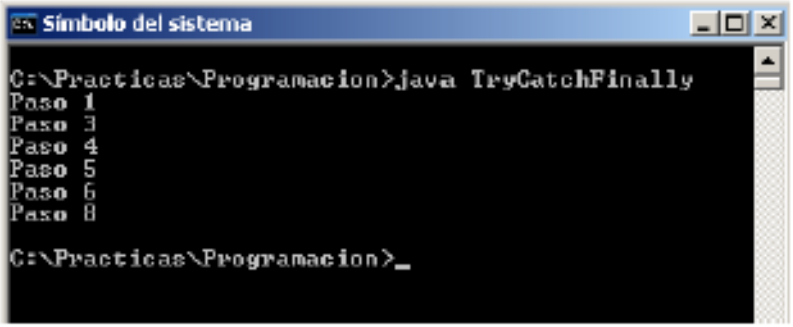
```
try {
   Aquí se protegen una o más sentencias
}
catch (Exception e) {
   Aquí se informa la excepción y se recupera de la misma
}
finally {
   Se realizan acciones comunes, se haya o no
   lanzado una excepción.
}
```

```
try {
   Aquí se protegen una o más sentencias
}
finally {
   Se realizan acciones comunes, se haya o no
   lanzado una excepción.
}
```

Si hay “excepción” se ejecuta “catch” y “filnally”

Si NO hay excepción, solo se ejecuta “finally”

```java
public class TryCatchFinally {
    public static void main(String[] args) {
        try {
            System.out.println("Paso 1");
            int a = 10 / 0;      // Lanza una excepción
            System.out.println("Paso 2");
        } catch(ArithmeticException ae) {
            System.out.println("Paso 3");
        } finally {
            System.out.println("Paso 4");
        }

        try {
            System.out.println("Paso 5");
            int a = 10 / 1;      // NO lanza una excepción
            System.out.println("Paso 6");
        } catch(ArithmeticException ae) {
            System.out.println("Paso 7");
        } finally {
            System.out.println("Paso 8");
        }
    }
}
```



12.6 Definir nuevas clases de excepción

Cuando las clases estándares de excepciones no describen satisfactoriamente la naturaleza del problema, se pueden definir nuevas clases mas descriptivas usando herencia

- Se puede extender RuntimeException para excepciones no comprobadas
- O Exception para excepciones comprobadas

```
 * @author David J. Barnes and Michael Kölling.
 * @version 2006.03.30
 */
public class NoCoincideContactoException extends Exception
{
      // La clave que no tiene coincidencias.
      private String clave;
      /**
       * Almacena los datos erróneos.
       * @param clave   La clave que no coincide.
       */
      public NoCoincidenContactoException(String clave)
      {
            this.clave = clave;
      }
      /**
       * @return La clave errónea.
       */
      public String getClave()
      {
            return clave;
      }

      /**
       * @return   Una cadena de diagnóstico que contiene
la clave errónea.
       */
      public String toString()
      {
            return "No se encontraron datos que coincidan
con : " + clave ;
      }
}
```

12.7 Usar aserciones: la sentencia “assert”

Una **aserción** es la afirmación de un hecho que debe ser verdadero en la ejecución normal del programa. Podemos usar aserciones para establecer explícitamente lo que asumimos y para detectar errores de programación más fácilmente.

- Modos de hacer comprobaciones durante el desarrollo de un proyecto y se eliminan en la versión de producto
  - Usadas para comprobaciones de consistencia interna
    - P.ej. sobre el estado de un objeto
- Sentencia de aserción en Java tiene dos formas:
  - **assert** *boolean-expression*
  - **assert** *boolean-expression* **:** *expression*
- La expresión booleana declara algo que debería ser cierto en ese punto
  - Se lanza un *AssertionError* si fuera falsa

```
/**
  * Elimina una entrada de la libreta de direcciones con
la clave dada.
  * La clave debe ser una de las que están actualmente en
uso.
  * @param clave Una de las claves de entrada a eliminar.
  * @throws IllegalArgumentException  Si la clave es null.
  */
public void eliminarContacto(String clave)
{
        if(clave == null){
              throw new IllegalArgumentException(
                                  "Se pasó clave null a
eliminarContacto.");
        }
        if(claveEnUso(clave)) {
              DatosDelContacto contacto = libreta.get(clave);
              libreta.remove(contacto.getNombre());
              libreta.remove(contacto.getTelefono());
              numeroDeEntradas--;
        }
        assert !claveEnUso(clave);
        assert tamanioConsistente() :
               "El tamaño de la libreta es inconsistente
en eliminarContacto";
}
```

La sentencia “assert” con expresión booleana

La palabra clave `assert` va seguida de una expresión booleana. El propósito de la sentencia es afirmar algo que debe ser verdadero en este punto del método. Por ejemplo, la primera sentencia `assert` en el Código 12.15 afirma que `claveEnUso` debe retornar, en este punto, un valor `false` ya sea porque la clave no estaba en uso en el primer lugar o bien porque no está más en uso pues se eliminaron de la libreta los datos asociados a ella. Esta afirmación aparentemente obvia es más importante de lo que podría parecer en un principio; observe que el proceso de eliminación no involucra realmente el uso de la clave con la libreta de direcciones.

Si la expresión booleana en una sentencia *assert* se evalúa *true*, entonces la sentencia *assert* no tiene más efecto; si la sentencia se evalúa *false* se lanzará un `AssertionError`. Este último es una subclase de `Error` (*véase* Figura 12.1) y forma parte de la jerarquía que representa errores irrecuperables: no se debe proveer ningún manejador a los clientes.

```
assert !claveEnUso(clave);
```

Segunda sentencia “assert” seguida de punto y coma

La segunda sentencia *assert* en el Código 12.15 ilustra la forma alternativa de una sentencia *assert*. La cadena seguida de un punto y coma se pasará al constructor de `AssertionError` para ofrecer una cadena de diagnóstico. La segunda expresión no tiene por qué ser una cadena explícita, puede ser cualquier expresión con un valor determinado que se convertirá en un `String` antes de ser pasada al constructor.

En la segunda forma, si no es cierto el retorno del método, genera un error con la cadena que ponemos

```
        assert tamanioConsistente() :
                "El tamaño de la libreta es inconsistente
en eliminarContacto";
}
```

Las aserciones se emplean en tareas de depuración de errores

Ejemplo de consistencia interna en la libreta de direcciones

```
/**
 * Controla que el campo numeroDeEntradas sea consistente
con
 * el número de entradas actualmente almacenadas en la
libreta.
 * @return true si el campo es inconsistente, false en
caso contrario.
 */
private boolean tamanioConsistente()
{
      Collection<DatosDelContacto> todasLasEntradas =
libreta.values();
      // Elimina los duplicados ya que se usan claves
múltiples.
      Set<DatosDelContacto> entradasUnicas =
new HashSet<DatosDelContacto>(todasLasEntradas);
      int cantidadActual = entradasUnicas.size();
      return numeroDeEntradas == cantidadActual;
}
```

12.8 Recuperarse de un error y anularlo

No es suficiente con detectar un error y emitir un mensaje, hay que tratar de solucionarlo

```
// Se intenta grabar la libreta de direcciones.
boolean exito = false;
int intentos = 0;
do {
  try {
    libreta.grabarEnArchivo(nombreDeArchivo);
    exito = true;
  }
  catch(IOException e) {
    System.out.println("Imposible grabar en " +
nombreDeArchivo);
    intentos++;
    if(intentos < MAX_INTENTOS) {
      nombreDeArchivo = un nombre de archivo
alternativo;
    }
  }
} while (!exito && intentos < MAX_INTENTOS);
if (!exito) {
  Informar el problema y rendirse.
}
```

Principios de recuperación de errores

Aunque este ejemplo ilustra la recuperación para una situación específica, los principios que ilustra son más generales:

- La anticipación de un error y la recuperación del mismo, generalmente requerirán un control de flujo más complejo que si el error no pudiera ocurrir.
- Las sentencias del bloque *catch* son la clave para preparar el intento de recuperación.
- La recuperación frecuentemente implica probar nuevamente.
- No se puede garantizar el éxito de la recuperación.
- Debe haber algunas rutas de escape que eviten que el intento de recuperación se realice desesperada y eternamente.

No siempre habrá un usuario humano al que se le pueda pedir un ingreso alternativo. Registrar el error debiera ser responsabilidad del cliente.

12.9 Entrada salida de texto

- escribir salida de texto en un archivo mediante la clase `FileWriter`;
- leer entradas de texto desde un archivo mediante las clases `FileReader` y `BufferedReader`;
- anticipar el lanzamiento de excepciones `IOException` desde las clases de E/S.

- Tutorial de E/S en JAVA
  - http://docs.oracle.com/javase/tutorial/essential/io/

12.9.3 Salida de texto con “FileWriter”

Hay tres pasos involucrados en el almacenamiento de datos en un archivo:

1. Se abre el archivo.
2. Se escriben los datos.
3. Se cierra el archivo.

El modelo básico que surge de la discusión anterior podría ser:

```
try {
   FileWriter escritor = new FileWriter("...nombre del
   archivo... ");
   while (hay más texto para escribir) {
      ...
      escritor.write(siguiente parte de texto);
      ...
   }
   escritor.close();
}
catch(IOException e) {
   algo anduvo mal al acceder al archivo
}
```

12.9.4 Entrada de texto con “FileReader”

Esto sugiere el siguiente modelo básico para leer el contenido de un archivo de texto:

```
try {
  BufferedReader lector = new BufferedReader(
    new FileReader("...nombre del archivo... "));
  String linea = lector.readLine();
  while (line != null) {
    hacer algo con la línea
    linea = lector.readLine();
  }
  lector.close();
}
catch(FileNotFoundException e) {
  no se encontró el archivo especificado
}
catch(IOException e) {
  algo anduvo mal al leer o al cerrar el archivo
}
```

12.9.5 Scanner: leer entradas desde terminal

```
Scanner lector = new Scanner(System.in);
...
String linea = lector.nextLine();
```

El método `nextLine` de `Scanner` retorna la siguiente línea completa desde la entrada estándar (sin la inclusión del carácter final `newLine`).

```
Scanner separador = new Scanner(lineaLog);
for(int i = 0; i < lineaDeDatos.length; i++) {
   lineaDeDatos[i] = separador.nextInt();
}
```

En este caso, se le aporta al `Scanner` una línea leída del archivo y la convierte en números enteros individuales.

12.9.6 Serialización de objetos

Java que se conoce como *serialización*. En términos simples, la serialización permite que se escriba un objeto completo en un archivo externo mediante una sola operación de escritura y recuperarlo en un paso posterior usando una sola operación de lectura[6]. Esto funciona con estos dos objetos simples y con objetos de múltiples componentes como son las colecciones. Es una característica importante que evita, por ejemplo, tener que leer y escribir objetos campo por campo. Es particularmente útil en el proyecto *libreta-de-direcciones* porque permite que se graben todas las entradas creadas en una sesión y luego leerlas nuevamente en otra sesión.

La **serialización** permite leer y escribir objetos completos y jerarquías de objetos en una única operación. Cada objeto involucrado debe ser de una clase que implemente la interfaz `Serializable`.

Términos introducidos en este capítulo

excepción, excepción no comprobada, excepción comprobada, manejador de excepciones, aserción, serialización

Resumen de conceptos

- **excepción** Una excepción es un objeto que representa los detalles del fallo de un programa. Se lanza una excepción para indicar que ha ocurrido un fallo.
- **excepción no comprobada** Las excepciones no comprobadas son un tipo de excepción cuyo uso no requiere controles del compilador.
- **excepción comprobada** Las excepciones comprobadas son un tipo de excepción cuyo uso requerirá controles adicionales del compilador. En particular, las excepciones comprobadas en Java requieren el uso de cláusulas *throws* y de sentencias *try*.
- **manejador de excepción** Es el código de un programa que protege las sentencias desde las cuales podría lanzarse una excepción. Proporciona código para la información y/o recuperación, una vez que se lanzó la excepción.
- **aserción** Una aserción es la afirmación de un hecho que debe ser verdadero en la ejecución normal de un programa. Podemos usar aserciones para establecer explícitamente las cuestiones que asumimos y para detectar errores de programación más fácilmente.
- **serialización** La serialización permite que se graben y lean objetos completos y jerarquías de objetos en una sola operación. Cada objeto involucrado debe ser de una clase que implemente la interfaz `Serializable`.