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.

Principios de lanzamiento 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");
```

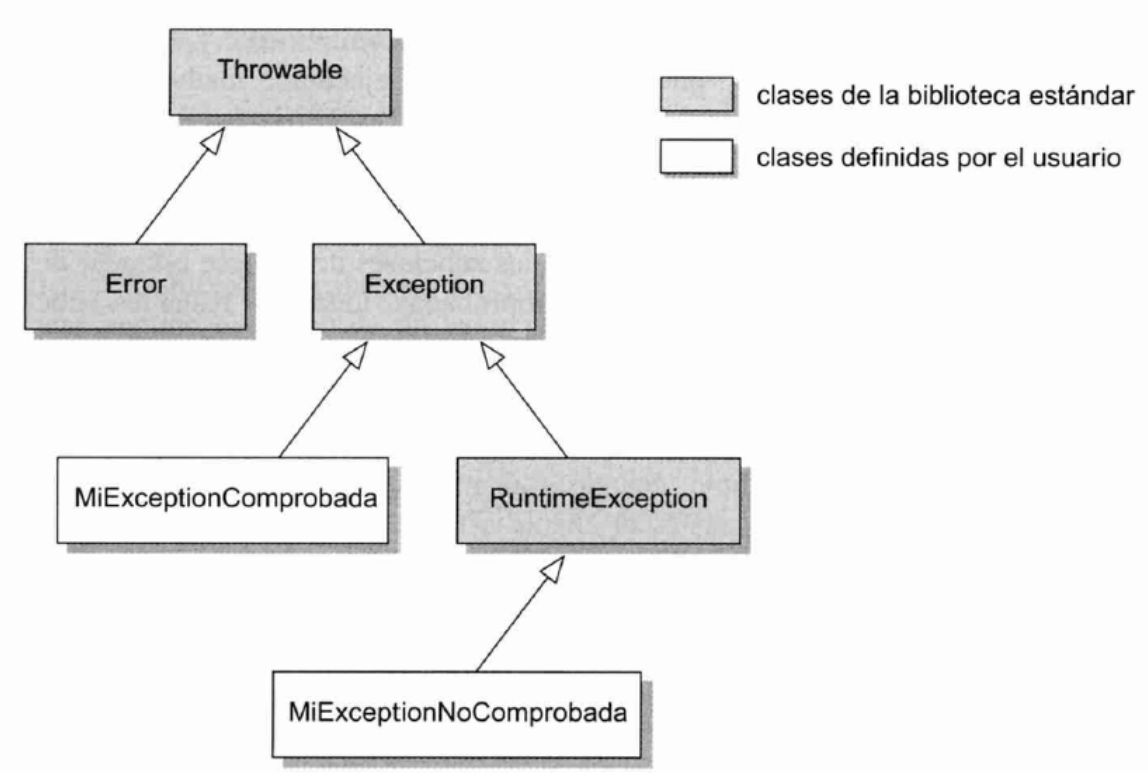
Una **excepción** es un objeto que representa los detalles de un fallo de un programa. Se lanza una excepción para indicar que ha ocurrido un fallo.

```
/**
   * 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);
}
```

Clases 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.

No comprobadas: del programa



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

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);
    }
```

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

```
/**
 * 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.");
          }
     }
```

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
```

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*.

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
}
```

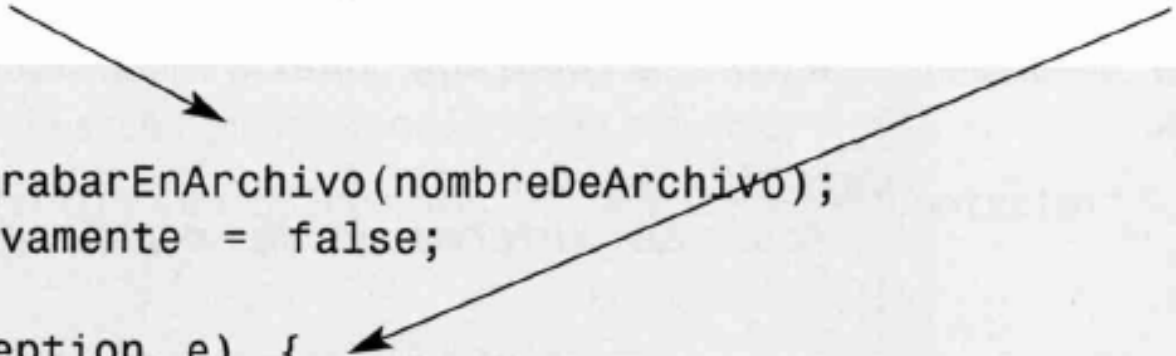
```
String nombreDeArchivo = null;
try {
   nombreDeArchivo = nombre-que-se-pide-al-usuario;
   libreta.grabarEnArchivo(nombreDeArchivo);
}
catch(IOException e) {
   System.out.println("Imposible grabar en " +
nombreDeArchivo);
}
```

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.

1. La excepción se lanza desde aquí

2. El control se transfiere aquí



```
try {
   libreta.grabarEnArchivo(nombreDeArchivo);
   probarNuevamente = false;
}
catch(IOException e) {
   System.out.println("Imposible grabar en " +
nombreDeArchivo);
   probarNuevamente = true;
}
```

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
      ...
}
```

Capturar todas las excepciones en un solo bloque “catch”

```
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
   ...
}
```

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.
}
```

Definir nuevas clases de excepción

```
 * @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 ;
      }
}
```

Usar aserciones: la sentencia “assert”

- Modos de hacer comprobaciones durante el desarrollo de un proyecto
- El compilador las incluirá si se lo pedimos

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.

```
/**
 * 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";
}
```

Recuperarse de un error y anularlo

```
// 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.

Entrada salida de texto

- http://docs.oracle.com/javase/tutorial/essential/io/

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
}
```

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
}
```

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.

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`.