

Programación Avanzada  
UD6

Autor:

P. Pablo Garrido Abenza pgarrido@umh.es

Universidad Miguel Hernández

01 de Octubre de 2012

Copyright ©2012- P. Pablo Garrido Abenza



## ÍNDICE GENERAL

<b>6 Control de excepciones y aserciones</b>	<b>1</b>
6.1. Control de excepciones ( <i>exceptions</i> ) . . . . .	1
6.2. Aserciones ( <i>asserts</i> ) . . . . .	15
<b>Glosario de acrónimos</b>	<b>19</b>

Copyright ©2012- P. Pablo Garrido Abenza

## CONTROL DE EXCEPCIONES Y ASERCIONES

En este capítulo se describen dos técnicas utilizadas para construir programas robustos: la captura de excepciones o errores en tiempo de ejecución (*exceptions*) y las aserciones (*asserts*) como mecanismo para la detección y corrección de errores (*bugs*).

### 6.1. Control de excepciones (*exceptions*)

Cuando se produce algún error durante la ejecución de un programa, el intérprete Java ejecuta el manejador de excepciones por defecto, que dependiendo de si se trata de programa en modo texto (consola) o una aplicación gráfica se comporta de modo distinto:

- Modo texto: se aborta la ejecución del programa, mostrando un mensaje.
- Modo gráfico: se muestra el mensaje pero la ejecución continua (aunque de forma inestable).

Java dispone de unas sentencias que permite capturar los errores en tiempo de ejecución, pudiendo nosotros decidir la acción a realizar en el caso de que se produzca un error, sin que se detenga la ejecución del programa. Se trata de las siguientes sentencias:

- La sentencia `try-catch` y `try-catch-finally`.
- La sentencia `try-with-resources` (IGNORAR).

### La sentencia try-catch

Los diseñadores de Java optaron por implementar un mecanismo de manejo de excepciones similar al utilizado en C++, que consiste en una sentencia try-catch-finally. El bloque try engloba el grupo de sentencias que queremos supervisar durante la ejecución. Tras el bloque try se escribe uno o más bloques catch, capturando cada uno de ellos una excepción de un tipo concreto. Si se produce una excepción no capturada en ningún bloque catch, el programa abortará. De forma opcional, puede incluirse un bloque finally, el cual se explicará en la siguiente sección.

```
try {  
    <bloque de sentencias a supervisar>  
} catch ( <TipoExcepcion1> e ) {  
    <bloque de sentencias a ejecutar en caso de error1>  
} catch ( <TipoExcepcion2> e ) {  
    <bloque de sentencias a ejecutar en caso de error2>  
    ...  
}
```

Cada bloque catch captura una excepción, especificando una clase que hereda de la clase Throwable; en la siguiente sección se mostrarán las posibles clases que podemos especificar. El objeto e que se declara de dicha clase, el cual puede tener cualquier nombre, se instancia automáticamente en el momento en que se produzca la excepción correspondiente, y contendrá toda la información sobre el error que se ha producido.

Durante la ejecución de la sentencia try-catch puede ocurrir lo siguiente:

- Que las sentencias del bloque try no generen ningún error: en ese caso, tras ejecutar esas sentencias la ejecución continuará con las sentencias que sigan a la sentencia try-catch.
- Que las sentencias del bloque try generen una excepción: la ejecución saltará al bloque catch correspondiente que captura dicho tipo de excepción, sin llegar a ejecutarse las sentencias del try que siguen a la que generó la excepción; en el caso de que no haya ningún bloque catch que la capture, el programa abortará, como si no estuviera protegido por una sentencia try-catch. También es posible que al ejecutar la sentencia dentro del try se hayan producido varias excepciones; en ese caso, es posible que se ejecuten varios bloques catch, en el mismo orden en el que se van encontrando.

Desde Java SE 7, es posible abreviar esta estructura en el caso de que varios catch vayan a realizar las mismas acciones, ya que un único bloque catch puede capturar varias excepciones. Para ello, se puede especificar una lista de excepciones separadas por una barra vertical ' | ':

```
try {  
    <bloque de sentencias a supervisar>  
} catch ( <TipoExcepcion1> | <TipoExcepcion2> | ... e ) {  
    <bloque de sentencias a ejecutar en caso de error1, error2, ...>  
}
```

## Jerarquía de excepciones

Hemos visto que cada `catch` declara un objeto, que deberá ser de una clase que hereda de la clase `Throwable`, es decir, tenemos una jerarquía de clases para las excepciones, tal y como se muestra en la figura 6.1. Pero esta jerarquía no está aislada del resto de la jerarquía de clases de Java, puesto que la clase raíz de esta jerarquía (`Throwable`) es subclase directa de la clase `Object`.

La clase `Throwable` tiene dos subclases que clasifican los posibles errores en dos grupos:

- **Exception**: son los errores habituales, capturables por el usuario. De esta clase se derivan otras subclases, e incluso podemos derivar nosotros nuestras propias clases.
- **Error**: son errores graves, errores de ejecución internos de la máquina virtual Java (JVM). Por ello, no conviene capturarlos, sino que dejaremos que aborte el programa.

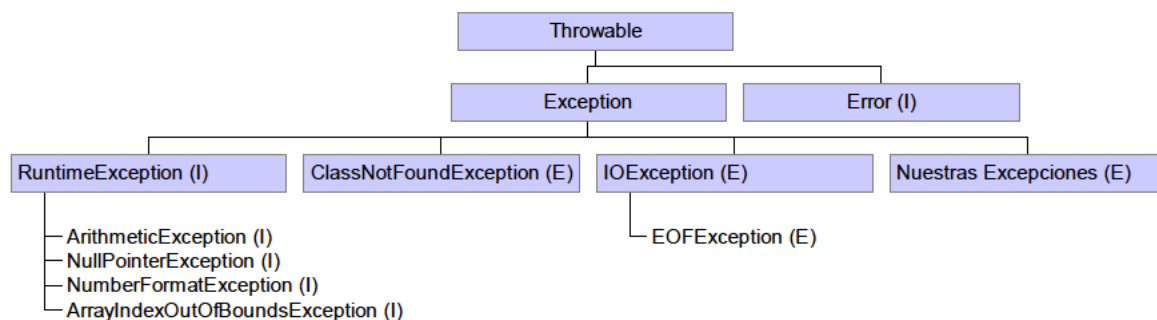


Figura 6.1: Jerarquía de clases de las excepciones

Además, las clases de esta jerarquía están marcadas con (I) o (E), indicando que nos podemos encontrar con dos tipos de excepciones:

- **Excepciones implícitas (I)**: no estamos obligados a capturarlas. Son todas las clases que heredan de `RuntimeException` y `Error`.
- **Excepciones explícitas (E)**: sí estamos obligados a capturarlas. Son `ClassNotFoundExcepction`, `IOException` y `EOFException`. Caso de que no se capturen deberán ser relanzadas (`throws`) para que otra función las capture (esto se explica más adelante).

A la vista de la jerarquía anterior, y basándonos en el concepto de herencia, podemos capturar varios tipos de excepciones con un único bloque `catch`. Por ejemplo, si capturamos la excepción `RuntimeException`, en realidad estaremos capturando una excepción de esa clase y cualquier subclase. Esto es así porque una subclase tiene una relación *es-un* con respecto su superclase, es decir, donde se espere un objeto de la clase `RuntimeException`, se aceptará un objeto también de cualquier subclase de ella (`ArithmeticException`, `NullPointerException`, `NumberFormatException`, `ArrayIndexOutOfBoundsException`). Si especificamos que la excepción a capturar es la clase `Exception`, entonces estaríamos capturando cualquier excepción que se pueda producir, escribiendo un único bloque `catch`.

```
try {
    <bloque de sentencias a
        supervisar>
} catch ( RuntimeException e ) {
    <bloque de sentencias en caso de
        error>
}
```

```
try {
    <bloque de sentencias a
        supervisar>
} catch ( Exception e ) {
    <bloque de sentencias en caso de
        error>
}
```

### El bloque finally

Anteriormente habíamos comentado que la sentencia try-catch-finally podía incluir al final un bloque finally. Este bloque siempre se ejecuta, independientemente de si se ha producido una excepción o no, incluso aunque se encuentre una sentencia return, continue o break dentro del try.

```
try {
    <bloque de sentencias a supervisar>
} catch ( <TipoExcepcion1> e ) {
    <bloque de sentencias a ejecutar en caso de error1>
} catch ( <TipoExcepcion2> e ) {
    <bloque de sentencias a ejecutar en caso de error2>
    ...
} finally {
    <bloque de sentencias a ejecutar en cualquier caso>
}
```

Por tanto, cuando se especifica un bloque finally:

- Si no se produce ningún error: después de ejecutarse las sentencias incluidas en el bloque try se ejecutarán las incluidas en el bloque finally.
- Si se produjera un error: la ejecución de las sentencias try se abortaría, ejecutándose el bloque catch correspondiente a la excepción que se haya producido, y después se ejecutarán las sentencias incluidas en el bloque finally. En el caso de que no se hubiese especificado ningún bloque catch para esa excepción, el programa no abortaría, ya que la excepción sería capturada por el bloque finally.

El bloque finally suele utilizarse para evitar la *fuga de recursos* (cerrar archivos abiertos, etc.). Sin embargo, en la versión Java SE 7 apareció una nueva sentencia try-with-resources más apropiada para estos casos, aunque no se explica para el presente curso.

### Acciones al capturar excepciones

Simplemente escribiendo un bloque try-catch, aunque dejemos los bloques catch en blanco, evitaremos que el programa aborte mostrando el mensaje de error correspondiente. Sin embargo, esto puede ocultarnos de forma silenciosa algún error que tendríamos que depurar. Por ello, dentro de los bloques catch se debería mostrar algún mensaje por pantalla o escribir algo en algún archivo (log). Podemos hacer uso del objeto `e` que cada catch declara. Como se ha dicho antes, dicho objeto se instancia automáticamente en el momento en que se produce una excepción, y contiene toda la información sobre el error producido (pila de llamadas, mensaje, etc.). Puesto que el objeto será de alguna clase subclase de `Throwable`, podremos utilizar los métodos definidos en esa clase raíz. Los métodos más habituales son los que se muestran en la tabla 6.1.

Método	Descripción
<code>getMessage()</code>	Retorna un <code>String</code> con el mensaje específico del error producido.
<code>toString()</code>	Retorna un <code>String</code> con el mensaje completo del error producido.
<code>printStackTrace()</code>	Imprime el mensaje completo y toda la pila de llamadas en el momento de producirse el error, exactamente igual que si no hubiéramos capturado el error. Por defecto utiliza el flujo de salida estándar para los errores <code>System.err</code> , aunque puede pasarse como parámetro cualquier objeto de las clases <code>PrintStream</code> (como <code>System.err</code> o <code>System.out</code> ) o <code>PrintWriter</code> (por ejemplo, un archivo en modo texto abierto por nosotros).
<code>getStackTrace()</code>	Construye un <code>array</code> en el que cada elemento contiene la información del método, archivo y línea desde el que se invoca dicho método, para poder acceder a la información de la pila de llamadas.

Cuadro 6.1: Clase `Throwable` - Algunos métodos

En siguiente fragmento de código se utiliza tres de los métodos mostrados de la clase `Throwable`. Podemos observar que mostramos ciertos mensajes por la pantalla, para que el programador pueda percibir los errores que se produzcan, y el programa continuará después su ejecución normal.

```
try {
    ...
} catch ( <TipoExcepcion1> e ) {
    System.err.println ("El mensaje es: " + e.getMessage());
} catch ( <TipoExcepcion2> e ) {
    System.out.println ("El mensaje es: " + e.toString());
} finally {
    e.printStackTrace ( System.err );
}
```

Si nos fijamos, en ocasiones se utiliza `System.out` y en otras `System.err`. Se trata de los distintos flujos de salida que ofrece Java, que junto con el flujo de entrada `System.in` forman los **flujos estándar** de Java. Aunque se hablará de ellos en un capítulo próximo, simplemente comentar que por defecto, los flujos de salida `System.out` y `System.err` están dirigidos a la pantalla, y el flujo de entrada `System.in` hace referencia al teclado. Por tanto, utilizar `System.out` y `System.err` aparentemente son idénticos, pero se explicará que, por ejemplo, el flujo `System.err` podría redirigirse a un fichero. De esa manera, los mensajes propios del programa utilizarán la pantalla con `System.out`, mientras que los mensajes de error incluidos en los blo-

ques try-catch-finally se escribirán en un archivo con `System.err`; por ello, conviene utilizar siempre `System.err` para mostrar los errores.

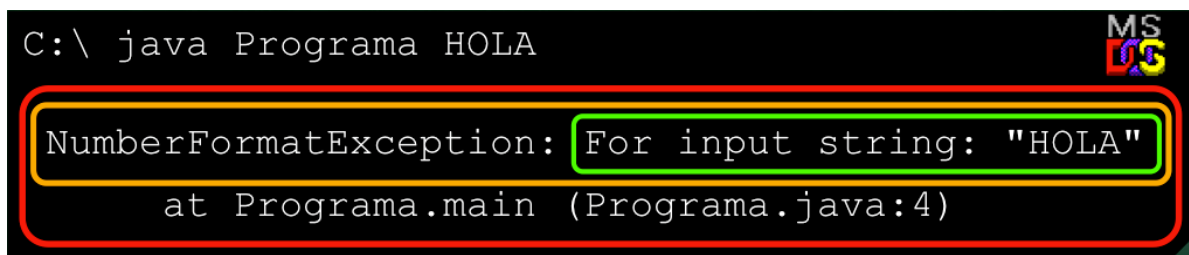
El siguiente **ejemplo** muestra un programa sencillo que puede generar dos excepciones en la misma línea: (1) `ArrayIndexOutOfBoundsException`, en el caso de que no se pase ningún argumento al programa, y (2) `NumberFormatException`, en el caso de que sí hayamos pasado un argumento, pero no pueda convertirse a entero por ser una cadena de texto con letras (por ejemplo).

```
public class ExceptionsDemo1 {  
    public static void main (String args[]) {  
        try {  
            int n = Integer.parseInt(args[0]);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            //System.err.println (e.getMessage());  
            //System.err.println (e.toString());  
            e.printStackTrace (System.err);  
        } catch (NumberFormatException e) {  
            e.printStackTrace (System.err);  
        }  
    }  
}
```

Sin invocamos al programa con un valor incorrecto que no puede convertirse a entero, el método `printStackTrace()` mostraría lo siguiente por pantalla, el mensaje de error seguido de la pila de llamadas:

```
C:\ java ExceptionsDemo1 HOLA  
java.lang.NumberFormatException: For input string: "HOLA"  
    at java.lang.NumberFormatException.forInputString (NumberFormatException.java:48)  
    at java.lang.Integer.parseInt (Integer.java:449)  
    at java.lang.Integer.parseInt (Integer.java:499)  
    at ExceptionsDemo1.main (ExceptionsDemo1.java:4)
```

De forma simplificada, y suponiendo que el programa se llama Programa, la figura 6.2 muestra el resultado utilizando los 3 métodos anteriores de la clase `Throwable`: lo de color verde sería lo que se mostraría utilizando el método `getMessage()`, lo de color naranja lo que aparecería con el método `toString()`, y si se utilizase el método `printStackTrace()`, se mostraría todo el mensaje, coincidiendo con el mensaje que se mostraría por pantalla en el caso de no haber capturado la excepción.



```
C:\ java Programa HOLA  
NumberFormatException: For input string: "HOLA"  
    at Programa.main (Programa.java:4)
```

Figura 6.2: Resultado de utilizar los 3 métodos de la clase `Throwable`



El cuarto método presentado de la clase `Throwable` era el `getStackTrace()`, el cual nos permite acceder a la información de la pila de llamadas, para mostrar mensajes personalizados. Este método nos construye un array de objetos `StackTraceElement`, el cual nos da información de cada método de la pila, así como el número de la línea del fichero. El siguiente ejemplo mostraría por pantalla la pila de llamadas, sin incluir el mensaje de error como hacía el método `printStackTrace()`. Podría también limitar el número máximo de métodos a mostrar, para el caso de que se hayan hecho muchas llamadas recursivas.

```
public class ExceptionsDemo2 {
    public static void main (String args[]) {
        try {
            int n = Integer.parseInt(args[0]);
        } catch (Exception e) {
            System.err.println (e.toString());
            StackTraceElement elements[] = e.getStackTrace();
            for (int i=0; i<elements.length; i++) {
                System.err.println ( "    Nivel=" + (elements.length-i) +
                    " -> Metodo " + elements[i].getMethodName() + "()" +
                    " (" + elements[i].getFileName() + ":" + elements[i].
                        getLineNumber() + ")" );
            }
        }
    }
}
```

En caso de invocar al programa anterior con un argumento incorrecto, el resultado será el siguiente, parecido al que se mostraba al utilizar el método `printStackTrace()`, pero personalizado. Podría modificarse también para que la pila apareciese en orden inverso, es decir, la función `main()` al principio, y se va aumentando el nivel conforme se baja, simplemente modificando los límites del bucle `for`:

```
C:\ java ExceptionsDemo2 HOLA
java.lang.NumberFormatException: For input string: "HOLA"
    Nivel=4 -> Metodo forInputString() (NumberFormatException.java:48)
    Nivel=3 -> Metodo parseInt() (Integer.java:449)
    Nivel=2 -> Metodo parseInt() (Integer.java:499)
    Nivel=1 -> Metodo main() (ExceptionsDemo2.java:11)
```

### Uso de archivos de *log* para almacenar mensajes (IGNORAR)

Para almacenar todos los mensajes de error en un archivo para su análisis posterior (archivo de *log*) tenemos varias alternativas:

- Utilizar siempre `System.err` (como hasta ahora), pero habiendo redirigido dicho flujo estándar para los errores a un archivo mediante el método `setErr()` de la clase `System`, aunque esto se explica en un capítulo posterior.
- Utilizar las clases `Logger` y `Level` de Java, definidas en el paquete `java.util.logging` (*Logging API*). Un ejemplo sería el siguiente:

```
import java.util.logging.*;
import java.io.*;

public class ExceptionsLog {

    public static void main (String args[]) {
        String nombreClase = ExceptionsLog.class.getName();
        String nombreLog = nombreClase + ".log";
        Logger logger = Logger.getLogger(nombreClase);
        try {
            Handler handler = new FileHandler (nombreLog);
            handler.setLevel (Level.ALL);
            logger.addHandler (handler);
        } catch (IOException e) {
            logger.log (Level.SEVERE, "La cosa parece grave: "+e.toString());
        }
        try {
            int n = Integer.parseInt(args[0]);
        } catch (Exception e) {
            logger.log (Level.INFO, "La cosa no es grave: "+e.toString());
        }
    }
}
```

- Utilizar el *framework* `log4j` (*Log-for-Java* de Apache), el cual ofrece el método `log.error()` y `log.warn()`, para utilizarlos según la gravedad que se estime de la excepción que se ha capturado. El uso de este *framework* se sale de los objetivos del curso; consultar el enlace proporcionado para más detalles.

## Relanzamiento de excepciones

En los casos anteriores capturábamos las excepciones justo en el lugar donde se pueden producir. Aunque esto es lo más habitual, en muchas ocasiones interesa decidir la acción a realizar en algún otro lugar. Por ejemplo, imaginemos una clase se utiliza tanto desde programas en modo texto como desde aplicaciones en modo gráfico, y que en algún punto puede generar una excepción. Al ser la clase compartida, no podemos tomar ninguna acción concreta, puesto que en cada caso mostraremos los mensajes de una forma, imprimiendo por pantalla o mostrando un cuadro de diálogo.

La solución es no capturar las excepciones justo en el método donde se producen sino en los métodos que lo han invocado, cualquiera por encima de aquel en la pila de llamadas. Para ello se utiliza la cláusula `throws`, que permite especificar una lista de excepciones separadas por comas que no serán manejadas en la función.

```
void miFuncion ( <argumentos> ) throws <lista-excepciones> {  
    ...  
}
```

Con la cláusula `throws` indicamos todas las excepciones que no serán manejadas dentro del método, sino que se delega su gestión en el método que la llama. Cuando se produce una excepción, primero se busca un gestor de excepciones en la misma función. Si no hay ninguno, se va retornando en la pila de llamadas hasta llegar a la función `main()`. Si finalmente, allí tampoco hay un manejador de excepciones para dicha excepción, se ejecuta el manejador de excepciones por defecto, que en el caso de un programa en modo texto significa que se abortará la ejecución.

Anteriormente habíamos visto que había **excepciones explícitas** (E) y **excepciones implícitas** (I). Si un método puede generar alguna excepción explícita, puesto que estamos obligados a capturar las excepciones explícitas, se debe proporcionar un gestor de excepciones `try-catch` en ese método, o especificarla en la cláusula `throws` y proporcionar dicho gestor de excepciones en cualquier otro método llamador. Un caso diferente son las excepciones implícitas, ya que como no es obligatorio su captura, no es obligatorio proporcionar un gestor de excepciones `try-catch` ni especificarlas en la cláusula `throws`; no obstante, si en tiempo de ejecución se produjera la excepción el programa abortará.

En general, las **ventajas** que nos aporta el relanzamiento de excepciones con `throws` son:

- Informar de situaciones anómalas. Si miramos la documentación de Java (API Specification), hay muchos métodos de muchas clases que informan que pueden generar varias excepciones, y es la forma de saber que al utilizarlas debemos capturar tales excepciones.
- Decidir las acciones a tomar fuera. Como se ha explicado antes, un método delega las acciones para capturar una excepción en otros métodos de otras clases, pudiendo el desarrollador de estas últimas decidir la acción a tomar: imprimir por pantalla con el método `printStackTrace()`, mostrar un cuadro de diálogo, etc.

Sin embargo, como **inconvenientes** podríamos decir que la depuración es algo más confusa, y también que la ejecución es algo más lenta.

El siguiente **ejemplo** es parecido al de la sección anterior, que simplemente convierte a entero (int) el primer argumento recibido desde línea de órdenes. En este caso no se hace todo en la función `main()`, sino que para la conversión se invoca a otra función `convertirEntero()`, la cual ya invoca a `Integer.parseInt()`. Puesto que en la función `main()` puede producirse la excepción `ArrayIndexOutOfBoundsException` al acceder a `args[0]` (igual que antes), se captura dicha excepción. Luego vemos que ya dentro de la función `convertirEntero()` se captura `NumberFormatException`, pues la llamada a `Integer.parseInt()` puede generarla.

```
public class ExceptionsDemo3 {
    public static void main (String args[]) {
        try {
            int n = convertirEntero (args[0]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println (e.toString());
        }
    }

    static int convertirEntero (String s) {
        int n = 0;

        try {
            n = Integer.parseInt (s);
        } catch (NumberFormatException e) {
            System.err.println (e.toString());
        }
        return (n);
    }
}
```

Ahora vamos a modificar de nuevo este ejemplo utilizando el relanzamiento de excepciones. Vemos ahora que en la función `convertirEntero()` ya no se captura ninguna excepción, sino que en ella se especifica mediante `throws` que la excepción `NumberFormatException` será capturada por la función llamadora (puesto que es una excepción implícita podríamos no haber escrito siquiera la cláusula `throws`). Por tanto, ahora las dos excepciones se capturan en la función principal `main()`, y puesto que las dos hacen lo mismo, tal como se explicó anteriormente, capturando la excepción `Exception` las capturamos las dos.

```
public class ExceptionsDemo4 {
    public static void main (String args[]) {
        try {
            int n = convertirEntero(args[0]);
        } catch (Exception e) {
            System.err.println (e.toString());
        }
    }
}
```

```
static int convertirEntero (String s) throws NumberFormatException {  
    int n = Integer.parseInt (s);  
    return (n);  
}  
}
```

En el ejemplo, el uso del relanzamiento de excepciones nos ha servido para simplificar algo el código, evitando tener repetidas las acciones de los bloques catch.

### Lanzamiento manual de excepciones

Las excepciones se generan automáticamente cada vez que se produce algún error, aunque también es posible crear manualmente objetos (instancias) de cualquiera de las clases de la jerarquía de Throwable, y lanzarla mediante la sentencia throw. El resultado es como si esa excepción hubiese sido generada automáticamente como cualquier otra.

Los **usos** más interesantes de para esto son:

- **Lanzar excepciones propias**, es decir, excepciones nuevas que se han añadido a la jerarquía de excepciones, lo cual se explica en el siguiente punto.
- **Relanzar una excepción que ya ha sido capturada**, si se viese que es por algún motivo grave y conviene abortar el programa, podríamos volver a relanzar de forma manual la misma excepción que se ha capturado (a la izquierda). También se puede haber capturado una excepción, y crearse una nueva excepción diferente a la capturada y lanzarse; la información de la excepción original (mensaje, línea, ...) se pierde (ya se ha consumido). Esto es lo que se conoce como **excepciones encadenadas** (derecha).

```
try {  
    ...  
} catch ( <TipoExcepcion> e ) {  
    ...  
    if (<es-grave>) {  
        throw e;  
    }  
}
```

```
try {  
    ...  
} catch ( <TipoExcepcion> e ) {  
    ...  
    BlahBlahException e2 = new  
        BlahBlahException();  
    throw e2;  
}
```

Pueden lanzarse excepciones tanto en métodos como en constructores. Las excepciones lanzadas en los constructores hacen que el objeto no llegue a crearse (seguirá valiendo null), y los objetos creados como parte del objeto que se está construyendo se marcarán para ser recolectados (null). Antes de que se recolecte la basura, se invocará el finalizador de cada uno de los objetos.

### Definición de excepciones propias

Es posible crear nuevas excepciones, simplemente escribiendo una clase específica que herede de alguna de las clases de la jerarquía `Throwable`, normalmente de `Exception` o `RuntimeException`. En resumen, los **pasos** serían los siguientes:

1. Escribir la clase correspondiente a la excepción, como subclase de `Exception` o `RuntimeException`, dependiendo de si queremos una excepción explícita o implícita, respectivamente.
2. En el punto que consideremos que se ha producido la excepción, ya sea un método o un constructor, crear un nuevo objeto de la nueva clase (`new`) y lanzarlo mediante `throw`. Si la excepción es explícita (hereda de `Exception`), el método o el constructor en el que se lanza debe especificarlo mediante `throws`; si es implícita (hereda de `RuntimeException`), no es necesario.
3. Capturar la nueva excepción como cualquier otra mediante un bloque `try-catch-finally`. En el caso de excepciones implícitas no es necesario, pero el programa abortará si se produce la excepción, de la misma forma que pasaría con cualquier otra excepción de Java.

Como **ejemplo**, vamos a crear una nueva excepción implícita, para no tener la obligación de capturarla y poder comprobar el resultado sin capturarla o capturándola. Esta excepción se lanzará cuando se intente crear un objeto de la clase `Fecha` con valores incorrectos.

```
1 // Clase para la excepcion
2 public class FechaNoValidaException extends RuntimeException {
3     // Constructor por defecto
4     FechaNoValidaException () {
5         super ();
6     }
7
8     // Constructor recibiendo mensaje opcional
9     FechaNoValidaException (String mensaje) {
10         super (mensaje);
11     }
12 }
```

La clase `Fecha` es la siguiente, la cual crea y lanza la excepción anterior en los lugares que se estime conveniente, cuando se detecte que se intenta crear una fecha incorrecta; por simplicidad, en este ejemplo consideraremos que una fecha no es válida sólo cuando el mes no esté comprendido entre 1 y 12. El punto concreto donde comprobamos este caso es en el constructor, aunque también debería hacerse en los métodos `setXXX()` si estuvieran, es decir, en cualquier lugar que pudiera modificar el valor de la fecha. En el constructor utilizamos `throws` para informar de que puede generarse `FechaNoValidaException`, pero como es implícita no sería obligatorio.

```
1 // Clase Fecha
2 import java.text.DecimalFormat;
```

```
3
4 public class Fecha {
5     private int dd, mm, aa;
6
7     // Constructor por defecto
8     public Fecha () {
9     }
10
11    // Constructor recibiendo mensaje opcional
12    public Fecha (int dd, int mm, int aa) throws FechaNoValidaException {
13        if ( (mm<1) || (mm>12) ) {
14            // Fecha no valida: creamos y lanzamos excepcion
15            FechaNoValidaException e = new FechaNoValidaException("Mes="+mm);
16            throw e;
17        }
18        this.dd = dd;
19        this.mm = mm;
20        this.aa = aa;
21    }
22
23    public String toString () {
24        DecimalFormat f00 = new DecimalFormat ("00");
25        String s = f00.format(dd) + "/" +
26                  f00.format(mm) + "/" +
27                  f00.format(aa);
28        return (s);
29    }
30 }
```

Finalmente, escribimos un programa de prueba de todo lo anterior, que simplemente crea un objeto de la clase Fecha, capturando la excepción FechaNoValidaException.

```
1 // Clase de test
2 public class ExceptionsPropiaDemo {
3     public static void main (String args[]) {
4         Fecha f=null;
5         try {
6             f = new Fecha(1,13,2012);
7         } catch (FechaNoValidaException e) {
8             System.err.println (e.toString());
9         }
10        System.out.println ("Fecha: " + f);
11    }
12 }
```

Cuando el programa crea una fecha correcta (por ejemplo, `f = new Fecha(1,12,2012);`), el resultado es el siguiente:

```
C:\ java ExceptionsPropiaDemo
Fecha: 01/12/2012
```

Cuando el programa crea una fecha incorrecta (por ejemplo, `f = new Fecha(1,13,2012);`), se genera la excepción `FechaNoValidaException`, pero como la hemos capturado se ejecuta el bloque `catch` correspondiente, el cual nos muestra el mensaje propio de Java, pero incluyendo la cadena adicional que hemos añadido justo en el momento de la generación de la excepción, para aportar más información al mensaje (sin la pila de llamadas). Después continúa con el resto del programa, que, en este caso, muestra el valor del objeto que ha intentado crear, pero es `null`, es decir, no ha llegado a crearlo porque se ha generado la excepción dentro del constructor.

```
C:\ java ExceptionsPropiaDemo
libro.FechaNoValidaException: Mes=13
Fecha: null
```

¿Qué ocurriría si en el caso anterior no hubiéramos capturado la excepción? El efecto es el mismo que para cualquier otra excepción de Java, es decir, el programa muestra el resultado del método `printStackTrace()` (mensaje y pila de llamadas), y aborta.

```
C:\ java ExceptionsPropiaDemo
Exception in thread "main" FechaNoValidaException: Mes=13
    at Fecha.<init>(Fecha.java:16)
    at ExceptionsPropiaDemo.main(ExceptionsPropiaDemo.java:6)
```

Para terminar, si la clase `FechaNoValidaException` hubiera heredado de `Exception`, sería una excepción explícita, y el compilador nos daría un error si no capturamos obligatoriamente la excepción en el programa principal. Salvo esto, el resto del código sería el mismo.

#### CONSEJOS:



- Para facilitar la lectura de código, utilizar siempre nombres de clase que terminen en `Exception`.
- Se recomienda crear excepciones explícitas, es decir, que hereden de `Exception`, ya que fuerza a que se tengan que capturar.



## 6.2. Aserciones (*asserts*)

Java dispone de una sentencia que nos permite asegurarnos de que se cumplan ciertas condiciones que seguro que deben cumplirse: la sentencia `assert`. Este mecanismo es muy efectivo para detectar posibles errores en los programas, al tiempo que sirven como documentación del código.

Para especificar una aserción, se utiliza la sentencia `assert` seguida de una condición booleana. En caso de que no se cumpla dicha condición, se lanzará un error `AssertionError` en tiempo de ejecución; si no se produce el error querrá decir que el programa funciona correctamente. Como cualquier excepción que se produce, si no se captura el programa abortará, mostrando la pila de llamadas, fichero y número de línea donde se ha producido. Si se quiere mostrar algún mensaje específico para aportar alguna otra información, éste se puede especificar de forma opcional a continuación de la condición en la misma sentencia `assert`, separado por dos puntos `:` (a la derecha):

```
assert <condicion>;
```

```
assert <condicion> : <mensaje o valor>;
```

El siguiente **ejemplo** debería imprimir los días de la semana, pero por un lapsus del programador, como límite del bucle se ha escrito el valor 8. Por fortuna, en la función `diaSemana()` que retorna el día de la semana para día especificado (valor 0..7), la sentencia `switch` ha escrito la cláusula `default`, evitando que el programa aborte. Para el caso de que el día especificado sea el 8, el programa no abortará, pero podría continuar silenciosamente y pasar el error desapercibido al programador. Por ello, se ha escrito una aserción en el `default`, para que nos avise de tal situación.

```
1 public class AssertDemo1 {
2
3     public static void main (String args[]) {
4         for (int i=1; i<=8; i++) {
5             System.out.println ("Dia semana " + i + " = " + diaSemana(i));
6         }
7     }
8
9     // Dia de la semana (d=1..7)
10    static String diaSemana (int d) {
11        String dia = "";
12
13        switch (d) {
14            case 1:
15                dia = "Lunes";
16                break;
17            case 2:
18                dia = "Martes";
19                break;
20            case 3:
21                dia = "Miercoles";
22                break;
23            case 4:
```

```
24         dia = "Jueves";
25         break;
26     case 5:
27         dia = "Viernes";
28         break;
29     case 6:
30         dia = "Sabado";
31         break;
32     case 7: // Diciembre
33         dia = "Domingo";
34         break;
35     default:
36         assert false : "Dia semana incorrecto (" + d + ")";
37     }
38     return (dia);
39 }
40 }
```

Ejecutando el programa desde la línea de órdenes obtenemos el siguiente resultado.

```
C:\> java AssertDemo1
Dia semana 1 = Lunes
Dia semana 2 = Martes
Dia semana 3 = Miercoles
Dia semana 4 = Jueves
Dia semana 5 = Viernes
Dia semana 6 = Sabado
Dia semana 7 = Domingo
Dia semana 8 =
C:\> _
```

Viendo el resultado, no parece que el programa haya mostrado ningún mensaje para el caso en que el día de la semana sea el 8, ni siquiera ha abortado mostrando un mensaje de error. Esto es así porque la comprobación de aserciones está deshabilitada por defecto, ignorándose las sentencias `assert`. En el siguiente punto se explica como habilitar o deshabilitar la comprobación de las aserciones.

### Habilitar y deshabilitar las aserciones

Puesto que comprobar las condiciones de las sentencias `assert` requiere tiempo de computación, por defecto la comprobación de aserciones está deshabilitada. Cuando se quieran utilizar, normalmente en tiempo de desarrollo, es necesario habilitarlas, por medio de unos parámetros a la hora de invocar al programa desde línea de órdenes. Por ejemplo, para ejecutar el programa anterior `AssertDemo1` con las aserciones habilitadas o no utilizaríamos las siguientes órdenes:

- Habilitar aserciones: `java -enableassertions AssertDemo1` (o `java -ea AssertDemo1`).
- Deshabilitar aserciones: `java -disableassertions AssertDemo1` (o `java -da AssertDemo1`), o simplemente, no especificar nada, puesto que están deshabilitadas por defecto.

Ejecutando el programa anterior con las aserciones habilitadas generaría la siguiente salida. Podemos comprobar que ahora sí se genera la excepción `AssertionError`, mostrando además el mensaje opcional que habíamos especificado que nos informa que el valor 8 es incorrecto en esta función. Como en cualquier excepción, nos muestra la pila de llamadas y el punto exacto del error.

```
C:\ java -enableassertions AssertDemo1
Dia semana 1 = Lunes
Dia semana 2 = Martes
Dia semana 3 = Miercoles
Dia semana 4 = Jueves
Dia semana 5 = Viernes
Dia semana 6 = Sabado
Dia semana 7 = Domingo
Exception in thread "main" java.lang.AssertionError: Dia semana incorrecto (8)
    at libro.AssertDemo1.diaSemana(AssertDemo1.java:36)
    at libro.AssertDemo1.main(AssertDemo1.java:5)
Java Result: 1
C:\ _
```

De esta forma, el *bug* cometido en la función `main()` ya no pasa desapercibido.

Si utilizásemos el IDE NetBeans, para especificar estos argumentos abriremos las propiedades del proyecto, y en el apartado Ejecutar (o Run), además de poder especificar la clase principal (o Main class) y sus argumentos, justo debajo tenemos otra línea para pasar opciones a la máquina virtual (ver figura 6.3).

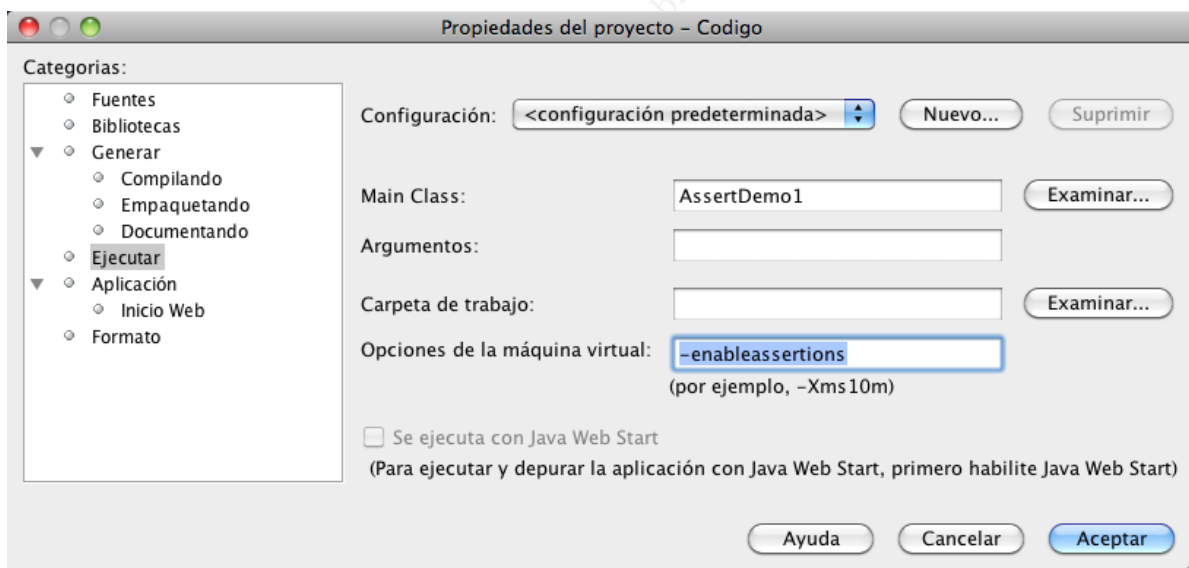


Figura 6.3: NetBeans - opción para habilitar las aserciones

Finalmente, comentar que es posible habilitar o deshabilitar las aserciones en todo el programa, como hemos hecho antes, o en ciertos paquetes, o sólo en ciertas clases. Para abreviar, en los siguientes ejemplos utilizaremos las opciones `-ea` y `-da`, que son equivalentes de las opciones `-enableassertions` y `-disableassertions`, respectivamente, siendo posible especificar varias de estas opciones:

- Habilitar las aserciones en todo el programa MiProg: `java -ea MiProg`.
- Habilitar las aserciones en el paquete `es.umh.p1`: `java -ea:es.umh.p1... MiProg` (notar los 3 puntos suspensivos tras el nombre del paquete).
- Habilitar las aserciones en la clase principal MiProg, y deshabilitarlas en la clase `Circulo` del paquete `es.umh.p1` (si la utilizase): `java -ea:MiProg -da:es.umh.p1.Circulo MiProg`

Copyright © 2012- P. Pablo Garrido Abenza

## GLOSARIO DE ACRÓNIMOS

Siglas	Significado
ANSI	American National Standards Institute
AOT	Ahead-Of-Time (técnica de compilación)
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CISC	Complex Instruction Set Computing
DBMS	DataBase Management System
DLL	Dynamic Linked Library
ETSI	European Telecommunications Standards Institute
GNU	GNU Not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineer
ISO	International Standards Organization
JAR	Java ARchive
J2SE	Java 2 Platform, Standard Edition (~ Java SE)
J2EE	Java 2 Platform, Enterprise Edition (~ Java EE)
J2ME	Java 2 Platform, Micro Edition (~ Java ME)
JCP	Java Community Process
JDBC	Java DataBase Conectivity
JDK	Java Development Kit (~ Java 2 SDK y J2SDK)
JFC	Java Foundation Classes
JIT	Just-In-Time (técnica de compilación)
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KNI	K Native Interface (~ JNI)
ODBC	Open Database Connectivity
OSI	International Organization of Standarization
PC	Personal Computer

PDA	Personal Digital/Data Assistant
RDBMS	Relational DataBase Management System
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SDK	Standard Development Kit
SQL	Structured Query Language
SRAM	Static Random Access Memory
URL	Uniform Resource Locator
UTF-16	Unicode Transformation Format (16-bit)
WORA	Write Once, Run Anywhere