TEMA 5. EXCEPCIONES EN JAVA

Introducción:

Las excepciones son el medio que ofrecen algunos lenguajes de programación para tratar situaciones anómalas que pueden suceder cuando ejecutamos un programa. Algunos casos de situaciones anómalas que se pueden citar, son, por ejemplo, invocar a un método sobre un objeto “null”, intentar dividir un número por “0”, intentar abrir un fichero que no existe para leerlo, quedarnos sin memoria en la JVM al ejecutar (por ejemplo, debido a un bucle infinito), intentar crear un fichero en una carpeta en la que no hay permisos de escritura, tratar de recuperar información de Internet cuando no se dispone de conexión, o simplemente sufrir un apagón eléctrico al ejecutar nuestro código.

Como se puede observar en los ejemplos anteriores, las excepciones no son inherentes a los lenguajes de POO, pero sí que lo son a Java como lenguaje de programación (también se pueden encontrar excepciones en otros lenguajes como Ada, ML, Eiffel, Prolog, Ruby o el propio C++). Hay que tener en cuenta que muchos de los métodos de la propia librería de Java hacen uso de excepciones, e incluso los programas que hemos desarrollado hasta ahora nos han permitido observar algunas de ellas (como, por ejemplo, la muy común *NullPointerException*). Por este motivo, cualquier programador en Java tiene que ser capaz de crear, lanzar y gestionar excepciones de una forma adecuada.

C++ también permite la programación con excepciones, y se puede hacer uso de las mismas en código programado por los usuarios, pero en la propia librería del lenguaje no son utilizadas y el uso de las mismas es completamente opcional, así que no profundizaremos en su estudio.

El Tema estará dividido de la siguiente forma: en la Sección 5.1 introduciremos la noción de excepción en programación así como alguna de las consecuencias de su uso con respecto al flujo normal de los programas y algún ejemplo sencillo de uso de las mismas. En la Sección 5.2 presentaremos algunos de los errores o excepciones más comunes de los que podemos encontrar en la librería de Java. En la Sección 5.3 veremos las distintas formas de gestión y propagación de excepciones que se nos ofrecen en Java, así como su sintaxis particular. Finalmente, en la Sección 5.4 incluiremos por una parte la forma de programar excepciones propias en Java, y en segundo lugar un ejemplo un poco más elaborado de programación con excepciones en el lenguaje en el que entrarán en juego diversas librerías del mismo con sus propios métodos y excepciones.

5.1 DEFINICIÓN DE EXCEPCIONES EN PROGRAMACIÓN

Definición: una excepción es una situación anómala que puede tener lugar cuando ejecutamos un determinado programa. La forma en que el programador

trate la misma es lo que se conoce generalmente como manejo o gestión de la excepción.

Las excepciones son una forma de intentar conseguir que, si un código fuente no se ejecuta de la forma “prevista” inicialmente, el programador sea capaz de controlar esa situación y decir cómo ha de responder el programa.

Conviene destacar un malentendido muy común sobre las excepciones. Las excepciones no sirven para “corregir” errores de programación. Por ejemplo, si un programa, durante su ejecución, no encuentra un determinado archivo, por medio de las excepciones no vamos a conseguir que el archivo “aparezca”. O si en un determinado programa intentamos acceder a una posición de un “array” mayor que la longitud del mismo (o a una posición negativa), las excepciones no van a conseguir que dicha posición del “array” exista. Las excepciones servirán para (1) alertarnos de dicha situación y (2) dejarnos decidir el comportamiento de nuestro programa en dicho caso. También resultan de gran utilidad en la depuración de grandes proyectos, ya que permiten recuperar la traza de los errores fácilmente, evitando que tengamos que “bucear” en el código buscando potenciales errores.

En realidad, los resultados que podemos conseguir por medio del uso de excepciones también se pueden conseguir por medio de una detallada atención a los fallos que pueden surgir durante la ejecución de un programa. Trataremos de verlo con el siguiente ejemplo, en el que en primer lugar gestionaremos los potenciales errores por medio de métodos tradicionales y ya conocidos, para dejar luego paso al uso de excepciones:

Supongamos que queremos escribir un método auxiliar que devuelva la división real de dos números reales (en Java), y que para lo mismo hacemos uso del operador de la librería “/”. Un primer código para dicho método podría ser:

```
public static double division_real (double dividendo, double divisor){
        return (dividendo / divisor);
}
```

El código es correcto, y el resultado devuelto por el método se corresponde con la división real de dos números reales. Por ejemplo, lo podemos comprobar con un sencillo ejemplo:

```
public static void main (String [] args){
                double x = 15.0;
                double y = 3.0;
                System.out.println ("El resultado de la division real de "
                        + x + " entre " + y + " es " + division_real (x, y));
}
```

El resultado de compilar y ejecutar el anterior programa sería:

El resultado de la division real de 15.0 entre 3.0 es 5.0

El método “division_real(double, double): double” está correctamente programado y produce la división real de los dos argumentos que recibe.

Sin embargo, hay, al menos, una situación “excepcional” que no hemos tenido en cuenta al programar el mismo. ¿Qué sucedería si al realizar la división de ambos números el denominador de la misma fuese igual a cero?

Lo comprobamos con el siguiente fragmento de código:

```
public static void main (String [] args){
        double x = 15.0;
        double y = 0.0;
        System.out.println ("El resultado de la division real de "
            + x + " entre " + y + " es " + division_real (x, y));
}
```

Al ejecutar la anterior operación se obtiene como resultado:

El resultado de la division real de 15.0 entre 0.0 es Infinity

Vemos que Java ha sido capaz de proporcionarnos una respuesta, en este caso “Infinity”, que se corresponde con el valor que Java asigna a las divisiones por cero (puedes comprobar en http://java.sun.com/javase/6/docs/api/java/lang/Double.html la existencia de constantes en Java representando los valores de más y menos infinito, entre otras).

Puede haber situaciones en las que dicho comportamiento no sea el deseado por nosotros, ya que un valor “Infinity” en nuestro programa podría provocar más adelante respuestas inesperadas, así que puede que un usuario quiera tratar dicho caso “excepcional” de una forma diferente. La forma natural para ello es definir una estructura condicional (un bloque “if ... else ...”) que nos permita separar dicho caso de los casos que el programa puede tratar de forma convencional. La estructura sería la siguiente:

```
public static double division_real (double dividendo, double divisor){
    double aux;
    if (divisor != 0){
        aux = dividendo/divisor;
    }
    else {
        aux = ¿?;
    }
    return aux;
}
```

Como siempre que plantemos la anterior situación, la pregunta que debemos responder ahora es, ¿qué comportamiento esperaremos de nuestro método en caso de que el divisor sea igual a cero? ¿Qué valor de “aux” debería devolver el método?

Lo primero que debemos tener claro es que el método debe devolver un valor "double", con lo cual no podemos devolver, por ejemplo, un "mensaje de error" advirtiendo al usuario de que está intentando dividir por cero. Una alternativa es devolver lo que se suele denominar como un "valor señalado", que nos permita distinguir que en el método ha sucedido algo anómalo. Un "candidato" a ser devuelto sería el propio cero, pero cero también es el resultado de dividir "0.0" entre "3.0", por lo que los clientes del método "division_real (double, double): double" difícilmente podrían distinguir entre una situación y la otra. Con cualquier otro valor real encontraríamos el mismo problema. Optaremos por devolver "0.0" en caso de que el divisor sea igual a cero:

```
public static double division_real (double dividendo, double divisor){
        double aux;
        if (divisor != 0){
                aux = dividendo/divisor;
        }
        else {
                aux = 0.0;
        }
        return aux;
}
```

Ahora el resultado de ejecutar el anterior fragmento de código sería:

El resultado de la division real de 15.0 entre 0.0 es 0.0

La forma anterior de tratar el caso excepcional, aunque pueda parecer un poco deficiente, es ampliamente utilizada por múltiples lenguajes de programación. Por ejemplo, en las librerías de C y C++ podemos encontrar las siguientes especificaciones de funciones encargadas de abrir o cerrar ficheros. En particular, nos vamos a preocupar de la función "fclose (FILE *): int", que es propia de la librería "cstdio". La especificación de la función ha sido extraída de http://www.cplusplus.com/reference/clibrary/cstdio/fclose.html:

int fclose (FILE * stream);

Return Value:

If the stream is successfully closed, a zero value is returned.
On failure, EOF is returned.

La función "fclose (FILE *): int" tiene por cometido cerrar el fichero que se le pasa como parámetro. Como se puede observar, el valor de retorno de la misma es un entero. Al leer detenidamente la especificación de su "Return Value" podemos observar que la misma devolverá cero en caso de que el fichero haya sido correctamente cerrado, o el valor "EOF", que es un entero negativo, en caso de que haya sucedido algún error al cerrar el mismo. Es decir, si sucede algo "anómalo" que no nos permita completar la acción, la

función nos avisará por medio del valor “EOF” (en lugar de, por ejemplo, mandarnos un mensaje o una señal advirtiendo de tal situación).

Si el programador que hace uso de la función “fclose (FILE *): int” tiene en cuenta que dicha función puede devolver un valor negativo, es probable que sea capaz de reaccionar ante tal hecho, y proponer un comportamiento específico para este caso “alternativo”. En caso contrario, el flujo del programa seguirá siendo el mismo, si bien el fichero no estará cerrado, tal y como nosotros esperábamos.

Aquí es donde las excepciones suponen una diferencia sustancial con respecto a la forma de gestionar los errores por medio de “valores señalados” (como un valor negativo, en el caso de la función “fclose (FILE *): int”, o devolver “0.0” en el caso de una división entre “0.0” con la función “division_real(double, double): double”).

Las excepciones tienen dos características principales a las cuales debemos prestar atención:

1) En primer lugar, las excepciones nos van a permitir “enriquecer” los métodos, en el sentido de que un método no sólo va a poder devolver un valor de un determinado tipo (por ejemplo, un “double” o un “int” en las funciones anteriores), sino que nos van a permitir que un método “lance” una excepción.

2) En segundo lugar, nos van a permitir alterar el flujo natural de un programa y que éste no sea único. Siguiendo con la propiedad que hemos citado en 1), si un método va a poder devolver un valor de un tipo determinado o lanzar una excepción, el programa cliente de dicho método debe estar preparado para ambos comportamientos, y eso hace que la ejecución de un programa no vaya a ser lineal, tal y como la entendíamos hasta ahora, sino que puede variar dependiendo de los valores o excepciones que produzca cada llamada a un método.

Veamos en primer lugar la característica que mencionamos en 1). Vamos a hacer que nuestro método “division_real(double, double): double” sea capaz de enviar una señal de que se ha producido una situación “anómala” a sus clientes (en este caso, sólo el método “main”) cuando el divisor sea igual a “0.0”. El código del método auxiliar “division_real(double, double): double” pasa ahora a ser:

```
public static double division_real (double dividendo, double divisor) throws Exception{
		double aux;
		if (divisor != 0){
			aux = dividendo/divisor;
		}
		else {
			throw new Exception();
		}
		return aux;
}
```

Veamos los cambios que han tenido lugar en el método con respecto a su versión anterior:

1) En primer lugar, la cabecera del mismo ha sufrido una modificación, pasando ahora a ser:

public static double division_real (double dividendo, double divisor) throws Exception{…}

Dicha cabecera expresa el siguiente comportamiento:

El método “division_real” (que es público, estático, y tiene como parámetros dos valores de tipo “double”) puede, o bien devolver un valor “double”, o bien lanzar un objeto de la clase “Exception”.

Vemos aquí una primera gran diferencia derivada de trabajar con excepciones. El flujo del código puede adoptar distintas direcciones. O bien el método “division_real (double, double): double” puede devolver un número real, o bien puede “lanzar” una excepción, en la forma de un objeto de la clase “Exception”.

Los clientes de dicho método deben ser capaces de “soportar” ambos comportamientos, lo cual veremos un poco más adelante.

2) La segunda diferencia sustancial del método “division_real(double, double): double” con respecto a su versión anterior la encontramos, precisamente, al estudiar el caso excepcional:

```
double aux;
if (divisor != 0){
        aux = dividendo/divisor;
}
else {
        throw new Exception();
}
return aux;
```

Como se puede observar en la parte correspondiente al “else {…}” (es decir, cuando el divisor es igual a “0”), lo que se hace no es darle un valor “señalado” (o arbitrario) a la variable “aux”, sino que lo que hacemos es, primero construir un objeto de la clase “Exception” (propia de Java, y de la que luego daremos detalles), y posteriormente “lanzarlo” por medio del comando “throw” (insistimos una vez más, lo que hacemos es “lanzar” la excepción a nuestro cliente, en este caso el método “main”).

Al lanzar dicha excepción, el flujo de nuestra aplicación vuelve al cliente de la función “division_real(double, double): double”, y ya no se llega a ejecutar la orden “return aux;”.

Nota: cabe señalar que ésta no es la única forma de gestionar las excepciones, más adelante veremos algunas formas distintas.

Después de ver los cambios que hemos debido introducir en el método “division_real (double, double): double” para poder decidir el caso excepcional y crear y lanzar la excepción, pasamos ahora a ver los cambios que debemos realizar en los clientes de dicho método para que sean capaces de “capturar” la excepción que lanza el mismo.

El comportamiento de los clientes del método “division_real (double, double): double” debe estar preparado para este nuevo escenario, en el que puede recibir un “double”, pero también puede que le llegue un objeto de la clase “Exception”. De nuevo, hay varias formas de prepararse para gestionar la excepción. Nos quedamos en este ejemplo con una de ellas que consiste en “capturar” la excepción y gestionarla. Eso se podría hacer por medio de los siguientes cambios en nuestro código para el método “main”:

```
public static void main (String [] args){

        double x = 15.0;
        double y = 3.0;
        try{
                System.out.println ("El resultado de la division real de " + x +
                                        " entre " + y + " es " + division_real (x, y));
        }
        catch (Exception mi_excepcion){
                System.out.println ("Has intentado dividir por 0.0;");
                System.out.println ("El objeto excepcion lanzado: " +
                                        mi_excepcion.toString());
        }
}
```

Veamos los cambios principales que hemos tenido que realizar en nuestro código:

1) En primer lugar, con respecto a la versión anterior, podemos observar cómo la llamada al método que lanza la excepción, “division_real (double, double): double” está ahora “encerrada” en un bloque “try {...}” de la siguiente forma:

```
try{
        System.out.println ("El resultado de la division real de " + x +
                                " entre " + y + " es " + division_real (x, y));
}
```

El bloque “try {...}” puede ser entendido como una forma de decirle a nuestro código “intenta ejecutar las órdenes dentro de este bloque, de las cuales alguna podría lanzar una excepción” (en nuestro caso, “division_real(double, double): double”).

2) La segunda modificación que debe ser incluida en nuestro código es el bloque “catch (...) {...}”. Lo primero que debemos notar es que un bloque “try {...}” siempre debe ir acompañado de un (o varios) bloque “catch (...) {...}”. La

función del bloque “catch (...) {...}” es precisamente la de “capturar” (o coger) las excepciones que hayan podido surgir en el bloque previo “try{...}”. En nuestro caso, el mismo tenía el siguiente aspecto:

```
catch (Exception mi_excepcion){
        System.out.println ("Has intentado dividir por 0.0;");
        System.out.println ("El objeto excepcion lanzado: " +
                                mi_excepcion.toString());
}
```

Veamos detalladamente la sintaxis del mismo:

a) En primer lugar, la orden “catch” es la que comienza el bloque.

b) Seguido de ella podemos encontrar la indicación “catch (Exception mi_excepcion) {...}”. La parte situada entre paréntesis indica el tipo de excepción que vamos a “capturar” en este bloque. La misma debería coincidir con alguna de las que se han lanzado en el bloque “try {..}”, es decir, en nuestro caso “Exception”. Además de eso, le damos un identificador local a la misma (en nuestro caso, “mi_excepcion”) que será válido dentro del bloque que va a continuación. Por tanto, el bloque “catch (Exception mi_excepcion) {...}” captura una excepción de la clase “Exception” y la etiqueta con el nombre “mi_excepcion”. En la parte correspondiente al bloque “{...}” introduciremos los comandos que queremos realizar en caso de que haya tenido lugar la excepción de tipo “Exception”. En nuestro caso, la única orden que hemos incluido en dicho bloque sería:

```
System.out.println ("Has intentado dividir por 0.0;");
System.out.println ("El objeto excepcion lanzado: " + mi_excepcion.toString());
```

Lo cual quiere decir que si la excepción de tipo “Exception” tiene lugar, se mostrarán por pantalla los mensajes señalados.

La acción a realizar depende del contexto en el que nos encontremos. Si, por ejemplo, la excepción detectada fuese un error de escritura a un fichero, podríamos optar por escribir a un fichero distinto. Si fuese un error del formato de los datos de entrada, podríamos pensar en pedirlos de nuevo al usuario, o en finalizar la ejecución del programa advirtiendo de dicho extremo.

En cualquier caso, debe quedar claro que si pretendemos gestionar una excepción lanzada por un método al que invoquemos, debemos facilitar un bloque “try {...}” en el que se incluya la llamada a dicho método, y un método “catch (TipoExcepcion nombre_excepcion) {...}” que indique, precisamente, lo que se debe hacer cuando surja tal excepción.

Veamos ahora el resultado de ejecutar el anterior código.

Si lo ejecutamos con una entrada “no excepcional”, por ejemplo, con los valores “15.0” y “3.0” que utilizamos inicialmente:

```
public static void main (String [] args){
        double x = 15.0;
        double y = 3.0;
        try{
                System.out.println ("El resultado de la division real de " + x +
                                        " entre " + y + " es " + division_real (x, y));
        }
        catch (Exception mi_excepcion){
                System.out.println ("Has intentado dividir por 0.0;");
                System.out.println ("El objeto excepcion lanzado: "
                                                + mi_excepcion.toString());
        }
}
```

Al encontrarnos fuera del caso excepcional, la ejecución es la esperada, y el resultado de compilar y ejecutar el programa sería:

El resultado de la division real de 15.0 entre 3.0 es 5.0

Si, por el contrario, la entrada que le damos al programa se encuentra dentro del caso excepcional, es decir, el divisor es igual a cero:

```
public static void main (String [] args){
        double x = 15.0;
        double y = 0.0;
        try{
                System.out.println ("El resultado de la division entera de " + x +
                                        " entre " + y + " es " + division_real (x, y));
        }
        catch (Exception mi_excepcion){
                System.out.println ("Has intentado dividir por 0.0;");
                System.out.println ("El objeto excepcion lanzado: "
                                        + mi_excepcion.toString());
        }
}
```

El resultado sería el siguiente:

Has intentado dividir por 0.0;
El objeto excepcion lanzado: java.lang.Exception

Podemos observar que el caso excepcional ha tenido lugar, y por tanto el método “division_real (double, double): double” ni siquiera ha llegado a ejecutar su orden “return aux”, sino que se ha salido de él por medio de la orden “throw new Exception();”, y eso ha provocado que en el programa cliente “main” el flujo de ejecución se haya introducido en la cláusula “catch (Exception mi_excepcion) {...}”, alterando el flujo normal de ejecución del programa.

Lo anterior nos ha servido para mostrar las características principales de las excepciones. Es importante retener la idea de que las excepciones en Java nos

permiten enriquecer el comportamiento de los métodos, ya que por el uso de excepciones podemos conseguir que los métodos devuelvan un valor determinado de un tipo cualquiera o que también “lancen” una excepción. También es importante tener en cuenta que el programador, cuando define un método, decide cuáles van a ser los casos “anómalos” o excepcionales que va a considerar su método. Como consecuencia de esto, cuando utilicemos métodos de la API de Java, deberemos comprobar siempre cuáles son las excepciones que los mismos pueden lanzar, para así gestionarlas de forma conveniente.

5.2 TIPOS DE EXCEPCIONES / ERRORES Y CÓMO TRATARLOS

Lo primero que debe quedar claro es que en Java, todas las excepciones que podamos usar o crear propias, deben heredar de la clase “Exception” propia de la librería de Java (http://java.sun.com/javase/6/docs/api/java/lang/Exception.html).

Existe una superclase de “Exception”, llamada “Throwable” (http://java.sun.com/javase/6/docs/api/java/lang/Throwable.html) que sirve para representar la clase de la que deben heredar todas las clases que queramos “lanzar”, por medio del comando “throw”. Sólo existen dos subclases directas de “Throwable” en la API de Java. Una es la clase “Error” (http://java.sun.com/javase/6/docs/api/java/lang/Error.html) que se utiliza para representar algunos errores poco comunes que pueden tener lugar cuando ejecutamos una aplicación en Java (por ejemplo, que la JVM se quede sin recursos). Java da la opción de que el usuario los gestione, pero en general no es recomendable hacerlo. La otra subclase directa de “Throwable” es “Exception”, y esta clase sirve para representar, citando a la propia API de Java, “condiciones que una aplicación razonable quizá quiera capturar”.

Sería conveniente comprobar algunos de los métodos y características de la clase “Exception” para poder trabajar con ella de un modo adecuado. En primer lugar, con respecto a su lista de métodos, podemos observar que cuenta con diversos constructores:

1) “Exception()”: un primer constructor sin parámetros, que es del cual hemos hecho uso en nuestro ejemplo anterior.

2) “Exception (String message)”: un segundo constructor, con un parámetro de tipo “String”, que nos permite crear la excepción con un determinado mensaje. Este mensaje puede ser accedido por medio del método “getMessage(): String” que mostraremos posteriormente.

Existen otros dos constructores para la clase “Exception”, pero por lo general no haremos uso de ellos.

Por lo demás, la clase no cuenta con métodos propios, pero sí con algunos métodos heredados de sus superclases que nos serán de utilidad. Especialmente útiles nos resultarán:

1) “toString(): String” que como ya conocemos pertenece a la clase “Object”. De todos modos, su comportamiento es redefinido (tal y como también hacíamos nosotros en el Tema 3) en la clase “Throwable” para que su comportamiento sea el siguiente:

Si hemos creado la excepción con un “message”, el resultado de invocar al método “toString(): String” será la siguiente cadena de texto:

NombreDeLaClase: message

En caso contrario, como ya comprobamos en nuestro ejemplo anterior de excepciones, el resultado será la siguiente cadena de texto:

NombreDeLaClase

2) “getClass(): Class”, método que está heredado también de la clase “Object”, y que lo que hace es devolver la clase del objeto sobre el que se invoca. Para nosotros, con conocer el nombre de dicha clase será suficiente.

3) “getMessage(): String”, es un método que pertenece a la clase “Throwable” y que nos devuelve una “String” que contiene el mensaje original con el que fue creado el objeto.

4)“printStackTrace():void” es un método que pertenece a la clase “Throwable” y que directamente imprime a la salida estándar de errores (en nuestro caso, la consola MSDOS) el objeto “Throwable” desde el que se invoca, así como la traza de llamadas a métodos desde el que se ha producido la excepción. Resulta de utilidad sobre todo para depurar programas, ya que nos ayuda a saber el punto exacto de nuestro código en el que surge la excepción, y el método que la ha producido.

Por supuesto, el comportamiento de todos estos métodos puede ser variado (redefinido) por nosotros al declarar nuestras propias clases que hereden de “Exception”.

Si seguimos observando la especificación que de la clase “Exception” se da en la API de Java (http://java.sun.com/javase/6/docs/api/java/lang/Exception.html) y prestamos atención a la sección “Direct Known Subclasses”, podemos observar ya algunas de las clases que en Java se van a considerar como excepciones de programación. No vamos a enumerar todas ellas, sólo algunas de las más destacadas o que ya hemos visto en nuestro trabajo con Java:

-ClassNotFoundException (http://java.sun.com/javase/6/docs/api/java/lang/ClassNotFoundException.html): Esta excepción tiene lugar cuando intentamos ejecutar un proyecto y, por ejemplo, la clase que contiene la función “main” no ha sido añadida al mismo o no es encontrada (una causa muy común para lo mismo es el haber configurado mal el proyecto).

-RuntimeException (http://java.sun.com/javase/6/docs/api/java/lang/RuntimeException.html): la clase RuntimeException representa el conjunto de las excepciones que pueden tener lugar durante el proceso de ejecución de un programa sobre la JVM, con la peculiaridad de que el usuario no tiene que prestar atención al hecho de capturarlas (todas las demás excepciones deberán ser capturadas o gestionadas en algún modo por el usuario).

Dentro de la especificación de la clase RunTimeException (http://java.sun.com/javase/6/docs/api/java/lang/RuntimeException.html) podemos encontrar un numeroso grupo de excepciones, con algunas de las cuales ya estamos familiarizados, y que conviene citar:

-ClassCastException (http://java.sun.com/javase/6/docs/api/java/lang/ClassCastException.html): excepción que tiene lugar cuando intentamos hacer un "cast" de un objeto a una clase de la que no es subclase. Un ejemplo sencillo está en la propia API de Java:

```
Object x = new Integer(0);
System.out.println((String)x);
```

Al no ser la clase "Integer" una subclase de "String", no podemos hacer un "cast" de forma directa. La situación anterior produciría una "ClassCastException" en tiempo de ejecución, no antes, que no es necesario que el usuario gestione.

-IndexOutOfBoundsException (http://java.sun.com/javase/6/docs/api/java/lang/IndexOutOfBoundsException.html): excepción que tiene lugar cuando intentamos acceder a un índice de un "array", "String" o "vector" mayor que el número de elementos de dicha estructura. Veamos también un ejemplo sencillo:

```
int array_enteros [] = new int [50];
System.out.println (array_enteros [67]);
```

Al intentar acceder a una posición del "array" mayor que la dimensión del mismo, podemos observar como Java lanza una excepción para advertirnos de que dicha dirección de memoria no ha sido reservada. De nuevo, estamos ante una excepción que aparecerá en tiempo de ejecución, y que no es necesario que el usuario gestione. ¿Qué habría sucedido en C++ ante esta misma situación?

-NegativeArraySizeException (http://java.sun.com/javase/6/docs/api/java/lang/NegativeArraySizeException.html): excepción que tiene lugar cuando intentamos crear un "array" con longitud negativa. Un ejemplo de una situación donde aparecería tal excepción sería el siguiente:

```
int array_enteros [] = new int [-50];
```

De nuevo es una excepción que el usuario no debe gestionar (pero debe ser consciente de que puede surgir en sus programas). Repetimos la pregunta que lanzábamos con la excepción anterior, ¿qué habría pasado en C++ ante esa situación?

-InputMismatchException (http://java.sun.com/javase/6/docs/api/java/util/InputMismatchException.html): excepción que lanzan varios métodos de la librería "Scanner" de Java cuando están recorriendo una entrada en busca de un dato y éste no existe en la entrada. Por ejemplo, si tenemos la siguiente entrada abierta:

Hola
Esto es un scanner
Que sólo contiene texto

Y sobre el mismo tratamos de ejecutar el método "nextDouble(): double", obtendremos dicha excepción. Una vez más, no es obligatorio gestionarla, aunque dependiendo de nuestra aplicación puede ser importante hacerlo.

-NumberFormatException (http://java.sun.com/javase/6/docs/api/java/lang/NumberFormatException.html): esta excepción tiene lugar cuando intentamos convertir un dato de tipo "String" a algún tipo de dato numérico, pero el dato de tipo "String" no tiene el formato adecuado.

Es usada, por ejemplo, en los métodos "parseDouble(): double", "parseInt(): int", "parseFloat(): float", ...

-NullPointerException (http://java.sun.com/javase/6/docs/api/java/lang/NullPointerException.html): quizá ésta sea la excepción que más comúnmente aparece cuando se trabaja con Java. Surge siempre que intentamos acceder a cualquier atributo o método de un objeto al que le hemos asignado valor "null". Un caso sencillo donde aparecería sería el siguiente:

```
Object ds = null;
ds.toString();
```

Como se puede observar, el objeto "ds" ha sido inicializado (con el valor "null"), lo que nos previene de obtener un error de compilación en Java, y al intentar acceder a cualquiera de sus métodos (por ejemplo, el método "toString(): String"), al invocar a un método sobre la referencia nula, obtenemos la excepción "NullPointerException". Similar excepción se obtiene también en la siguiente situación:

```
Object array_ob [] = new Object [25];
array_ob[1].toString();
```

Declaramos y creamos un “array” de objetos de la clase “Object”, pero no inicializamos cada uno de dichos objetos; esto quiere decir que cada componente del “array” contiene el objeto “null”. Al intentar acceder a cualquier propiedad (por ejemplo, “toString(): String”) de cualquiera de ellos, obtenemos una excepción de tipo “NullPointerException”.

Esta excepción tampoco es necesario gestionarla, pero sí que conviene conocer su origen ya que puede ser de gran utilidad a la hora de depurar y corregir programas.

Veamos ahora algunas nuevas excepciones, que heredan de la clase “Exception” en Java, pero no de “RunTimeException”, y que por tanto los usuarios deben gestionar, al hacer uso de los métodos que las lancen:

-IOException (http://java.sun.com/javase/6/docs/api/java/io/IOException.html): ésta es otra de las excepciones más comúnmente usadas en la librería de Java. Los métodos de Java la lanzan siempre que encuentren un problema en cualquier operación de lectura o escritura a un medio externo (lectura o escritura a un fichero, por ejemplo).

Puedes observar la lista de métodos de la API de Java que la pueden lanzar en http://java.sun.com/javase/6/docs/api/java/io/class-use/IOException.html, así que cada vez que hagas uso de alguno de esos métodos deberás gestionarla de algún modo.

También deberías observar todas sus subclases (es decir, las excepciones que heredan de ella), porque algunas son también muy comunes (http://java.sun.com/javase/6/docs/api/java/io/IOException.html). Gracias a los mecanismos de herencia aplicados a excepciones, siempre que gestiones una “IOException” estarás capturando también cualquiera de las excepciones que heredan de la misma.

También puedes observar en la página anterior que la lista de constructores y métodos de la misma es equivalente al de la clase “Exception“.

-FileNotFoundException (http://java.sun.com/javase/6/docs/api/java/io/FileNotFoundException.html): esta excepción hereda de la clase “IOException” que acabamos de introducir. En general, la lanzan diversos métodos de la API de Java cuando intentan abrir algún fichero que no existe o no ha sido encontrado.

Aparte de estas excepciones, cada usuario puede definir las suyas propias, tal y como veremos en la Sección 5.4, simplemente declarando una clase que herede de la clases “Exception” o “RunTimeException” (si preferimos que sea una excepción que pueda no ser gestionada) de Java.

5.3 TRABAJANDO CON EXCEPCIONES: DECLARACIÓN, CONSTRUCCIÓN, LANZAMIENTO Y GESTIÓN DE EXCEPCIONES

Una vez conocemos algunas de las excepciones más comunes con las que nos podemos encontrar al programar en Java, en esta Sección pasaremos a ver cómo debemos trabajar con las mismas, es decir, cómo debemos gestionarlas.

En general, cuando dentro de un método aparece una excepción es porque el propio método la ha creado y lanzado, o porque dicha excepción ha venido lanzada de algún otro método que ha sido invocado desde éste.

A partir de esa situación, y como norma general se puede enunciar que, un método cualquiera, cuando se encuentra con una excepción (por cualquiera de los dos motivos anteriores) que no sea de tipo “RunTimeException”, puede propagar la misma, o puede capturarla y gestionarla (si es de tipo “RunTimeException”, el hecho de gestionarla es opcional, depende del programador).

En la Sección 5.1 ya hemos visto un sencillo ejemplo en el cual hemos lanzado y capturado una excepción, que es una de las formas habituales de gestionarlas, pero no la única. En esta Sección veremos tres formas distintas de gestionar una excepción que se podrían declarar como “propagar” la misma (Sección 5.3.1), “capturar y gestionar” la misma (Sección 5.3.2), y, por último, una combinación de los dos métodos anteriores, “capturar, gestionar y propagar” la excepción.

Para ello vamos a escoger un ejemplo sencillo. Imagina que tenemos un método auxiliar que recibe como parámetro un “array” de números reales y un entero. Dicho método devolverá la posición del “array” indicada por el entero. Como ya hemos indicado antes, Java cuenta con excepciones propias (“IndexOutOfBoundsException”) que se encargan de esta misión, pero dicha excepción es de tipo “RunTimeException”, por lo cual no es obligatorio que sea gestionada por el usuario. Lo que haremos será gestionar el caso excepcional anterior (que no exista el índice que se nos solicita) por medio de una excepción de la clase “Exception”, que sí debe ser gestionada por el usuario.

5.3.1 PROPAGACIÓN DE EXCEPCIONES

Veamos en primer lugar qué aspecto podría tener el método auxiliar que nos solicita el problema:

```
public static double acceso_por_indice (double [] v, int indice){
        return v [indice];
}
```

Un sencillo programa cliente que hace uso del método auxiliar podría ser el siguiente:

```
public static void main (String [] args){
                        double array_doubles [] = new double [500];
                        for (int i = 0; i < 500; i++){
                                array_doubles [i] = 7 * i;
```

```
                    }
                    for (int i = 0; i < 500; i = i + 25){
                            System.out.println ("El elemento en " + i + " es "
                                    + acceso_por_indice (array_doubles, i));
                    }
}
```

Presentaremos en primer lugar la forma de hacer que el método auxiliar “acceso_por_indice (double [], int): double“ sea capaz de lanzar una excepción cuando no se cumple la condición que hemos exigido.

```
public static double acceso_por_indice (double [] v, int indice) throws Exception{
        if ((0<=indice) && (indice <v.length)){
                return v [indice];
        }
        else {
                //Caso excepcional:
                throw new Exception (“El indice ” + indice +
                                        “ no es una posicion valida”);
        }
}
```

Veamos los cambios que hemos introducido en el método al declarar y propagar la excepción:

1. En primer lugar, hemos declarado una estructura condicional, por medio de un “if ... else ...” que nos permite separar los casos “buenos” de los casos excepcionales. En el mismo puedes observar que hemos hecho uso de “length”, que es un atributo de instancia con el que cuentan todos los “arrays” y que nos permite saber cuántas componentes han sido reservadas para el mismo.

Si se verifica la condición bajo el “if”, el método auxiliar se comportará como debe y su resultado será el esperado.

2. En caso de que el valor de “indice” se encuentre fuera del rango especificado, es cuando nos encontramos con el caso excepcional. Como se puede observar, lo que hacemos es crear un objeto por medio del constructor “Exception (String)”, y lanzarlo por medio del comando “throw”.

3. Lo tercero que debemos observar es que el objeto que hemos lanzado por medio de “throw” no ha sido capturado dentro del método “acceso_por_indice (double [], int): double”, lo cual quiere decir que esa excepción está siendo propagada por el método. Todos los métodos clientes de este método deberán tenerlo en cuenta a la hora de ser programados, gestionando la excepción que les podría llegar. Cabe recordar que las dos opciones que existen ante una excepción que ha sido lanzada son capturar la misma y gestionarla, o en su defecto propagarla (al método que haya invocado este método). En este caso, nos hemos decantado porque nuestro método “acceso_por_indice (double [],

int): double” propague la excepción, y para ello debemos indicar en la cabecera del método dicha situación.

De ese modo la cabecera queda como:

public static double acceso_por_indice (double [] v, int indice) throws Exception{…}

Por medio del comando “throws Exception” estamos avisando de que nuestro método lanza una excepción, y por tanto cualquier usuario que invoque al mismo deberá capturar dicha excepción, o, si lo prefiere, de nuevo propagarla.

Veamos ahora por ejemplo cómo queda la llamada al mismo que hacíamos desde el cliente “main” de nuestro método “acceso_por_indice (double [], int)”:

```
public static void main (String [] args){
        double array_doubles [] = new double [500];
        for (int i = 0; i < 500; i++){
                array_doubles [i] = 7 * i;
         }

         for (int i = 0; i < 600; i = i + 25){
              try {
                   System.out.println ("El elemento en " + i + " es "
                       + acceso_por_indice (array_doubles, i));
               }
               catch (Exception e){
                   System.out.println (e.toString());
               }
        }
}
```

Como se puede observar, el hecho de que “acceso_por_indice (double [], int): double” lance una excepción ha hecho que, al hacer uso del mismo, tengamos que capturarla y gestionarla dentro de nuestro método “main”, por medio de una estructura “try {...} catch (...) {...}”.

De nuevo, nuestro método “main” podría haber “propagado” la excepción, por medio de la cabecera:

public static void main (String [] args) throws Exception{..}

Aunque esta solución no resulta muy elegante, ya que nos hace perder la información de dónde y cómo se han producido las excepciones.

Por último, podemos prestar atención a la salida que produce la ejecución del fragmento de código anterior. En este caso sería:

El elemento en 0 es 0.0
El elemento en 25 es 175.0
El elemento en 50 es 350.0

```
El elemento en 75 es 525.0
El elemento en 100 es 700.0
El elemento en 125 es 875.0
El elemento en 150 es 1050.0
El elemento en 175 es 1225.0
El elemento en 200 es 1400.0
El elemento en 225 es 1575.0
El elemento en 250 es 1750.0
El elemento en 275 es 1925.0
El elemento en 300 es 2100.0
El elemento en 325 es 2275.0
El elemento en 350 es 2450.0
El elemento en 375 es 2625.0
El elemento en 400 es 2800.0
El elemento en 425 es 2975.0
El elemento en 450 es 3150.0
El elemento en 475 es 3325.0
java.lang.Exception: El indice 500 no es una posicion valida
java.lang.Exception: El indice 525 no es una posicion valida
java.lang.Exception: El indice 550 no es una posicion valida
java.lang.Exception: El indice 575 no es una posicion valida
```

Como se puede observar, todas y cada una de las veces que hemos llamado al método “acceso_por_indice (double [], int): double” con un valor fuera del rango del “array” la respuesta que hemos obtenido es la excepción correspondiente que nos avisa de que se ha producido dicha situación excepcional (y el método “acceso_por_indice (double[], int): double” no ha tenido que devolver un valor “double”, sino que simplemente ha lanzado la excepción mostrada).

5.3.2. CAPTURA Y GESTIÓN DE EXCEPCIONES

La segunda opción que hay ante una excepción es capturar la misma y gestionarla (como ya hemos visto en algunos ejemplos anteriores). Lo que haremos ahora será programar el método “acceso_por_indice (double [], int): double” de tal forma que él mismo lance y capture su propia excepción. Una posible solución para lo mismo sería:

```
public static double acceso_por_indice (double [] v, int indice){
    try {
        if ((0<=indice) && (indice <v.length)){
            return v [indice];
        }
        else {
            //Caso excepcional
            throw new Exception ("El indice " + indice +
                                 " no es una posicion valida");
        }
      }
    catch (Exception mi_excepcion){
```

```
                System.out.println(mi_excepcion.toString());
                System.out.println(mi_excepcion.getMessage());
        }
        finally {
                return 0.0;
        }
}
```

Veamos algunas peculiaridades sobre la sintaxis de la captura y gestión de la excepción:

1. En primer lugar, la estructura “try {...} catch(...){...}” con la que ya nos hemos encontrado con anterioridad.

Dentro de la parte “try{...}” es donde debe surgir la excepción, en este caso, donde debe estar el “throw”. En este caso, la excepción que puede surgir de este fragmento de código es de tipo “Exception”, y ése debe ser el tipo de excepciones que capturemos dentro de la estructura “catch(...){...}”.

En la estructura “catch (...){...}” hay dos partes diferenciadas que debemos destacar:

- En primer lugar, entre paréntesis, como si fuera la cabecera de un método, debemos situar el tipo de excepción que queremos capturar (en nuestro caso, “Exception”), seguido de un identificador para el mismo, por ejemplo “mi_excepcion”, que será el identificador por el que nos podamos referir a la misma en el bloque entre llaves:

```
catch (Exception mi_excepcion){...}
```

- En segundo lugar, entre llaves, nos podemos encontrar con la parte en la que gestionamos la excepción. ¿Qué queremos hacer en caso de que surja la excepción? Por ejemplo, en nuestro caso, lo que vamos a hacer es simplemente mostrar un mensaje por pantalla que nos permita saber de qué tipo es la excepción generada, y mostrar el mensaje que le hemos asignado a la misma:

```
catch (Exception mi_excepcion){
                System.out.println(mi_excepcion.toString());
                System.out.println(mi_excepcion.getMessage());
}
```

2. En segundo lugar, podemos observar el bloque “finally {...}” también propio del trabajo con excepciones (debe ir siempre a continuación de un bloque “try {...} catch (...) {...}”), que nos permite realizar ciertas acciones que nos permitan salir del método “acceso_por_indice (double [], int): double” de una forma más “correcta” o “elegante”. Por ejemplo, dicho bloque se podría utilizar para cerrar la conexión con un fichero que tuviéramos abierto, o con una base de datos, o para vaciar los búferes, ...

En nuestro caso, y ya que nuestro método ha de devolver un “double”, lo hemos usado para situar la sentencia “return 0.0;” como valor por defecto del método.

3. Por último, cabe también destacar que la cabecera del método ya no lanza ninguna excepción, sino que la gestiona por sí mismo, y por tanto ha vuelto a ser:

```
public static double acceso_por_indice (double [] v, int indice){…}
```

Los clientes de este método ahora no deberán preocuparse de que el método “acceso_por_indice (double[], int): double” lance una excepción, ya que el propio método las captura y gestiona. Veamos entonces ahora un posible cliente “main” para el mismo:

```
public static void main (String [] args){
        double array_doubles [] = new double [500];
        for (int i = 0; i < 500; i++){
                array_doubles [i] = 7 * i;
        }
        for (int i = 0; i < 600; i = i + 25){
                System.out.println ("El elemento en " + i + " es "
                                        + acceso_por_indice (array_doubles, i));
        }
}
```

Como podemos observar, el mismo no presta atención a la captura y gestión de excepciones. El resultado de ejecutar ahora el mismo será:

```
El elemento en 0 es 0.0
El elemento en 25 es 0.0
El elemento en 50 es 0.0
El elemento en 75 es 0.0
El elemento en 100 es 0.0
El elemento en 125 es 0.0
El elemento en 150 es 0.0
El elemento en 175 es 0.0
El elemento en 200 es 0.0
El elemento en 225 es 0.0
El elemento en 250 es 0.0
El elemento en 275 es 0.0
El elemento en 300 es 0.0
El elemento en 325 es 0.0
El elemento en 350 es 0.0
El elemento en 375 es 0.0
El elemento en 400 es 0.0
El elemento en 425 es 0.0
El elemento en 450 es 0.0
El elemento en 475 es 0.0
java.lang.Exception: El indice 500 no es una posicion valida
```

```
El indice 500 no es una posicion valida
El elemento en 500 es 0.0
java.lang.Exception: El indice 525 no es una posicion valida
El indice 525 no es una posicion valida
El elemento en 525 es 0.0
java.lang.Exception: El indice 550 no es una posicion valida
El indice 550 no es una posicion valida
El elemento en 550 es 0.0
java.lang.Exception: El indice 575 no es una posicion valida
El indice 575 no es una posicion valida
El elemento en 575 es 0.0
```

En la salida anterior del programa se puede observar como cada vez que ha ocurrido una excepción la misma ha sido gestionada por el propio método “acceso_por_indice (double [], int): double”. Esto tiene una contrapartida, y es que el método “acceso_por_indice (double [], int): double” debe devolver siempre un “double” (no existe la posibilidad de que el método lance una excepción), y por tanto debemos devolver un valor “comodín”, que en nuestro caso será “0.0”.

El hecho de gestionar la excepción en el propio método ha hecho que los clientes del mismo no sean conscientes de que tal excepción ha sucedido y por tanto no pueden actuar en consecuencia. En cierto modo, el hecho de capturar la excepción de forma prematura ha hecho que se pierda parte de la utilidad de la misma, que es “avisar” a los clientes de nuestro método de que una situación excepcional se ha producido en el mismo. Este tipo de gestión de excepciones puede resultar más útil para depurar nuestro propio código, detectar errores en el mismo, ...

Veamos por último la posibilidad de capturar y gestionar la excepción y además propagarla, que no deja de ser una combinación de los dos métodos anteriores.

5.3.3 CAPTURA, GESTIÓN Y PROPAGACIÓN DE EXCEPCIONES

Al programar el método “acceso_por_indice (double [], int): double” podemos decidir que el mismo capture la excepción y además la propague. La solución es una combinación de las dos que hemos visto hasta ahora. Una posible programación del método para que se comporte de dicho modo sería:

```
public static double acceso_por_indice (double [] v, int indice) throws Exception{
        try {
                if ((0<=indice) && (indice <v.length)){
                        return v [indice];
                }
                else {
                        //Caso excepcional:
                        throw new Exception ("El indice " + indice
                                                + " no es una posicion valida");
                }
            }
```

```
                catch (Exception mi_excepcion){
                        System.out.println(mi_excepcion.toString());
                        System.out.println(mi_excepcion.getMessage());
                        throw mi_excepcion;
                }
}
```

En realidad el método no añade nada que no hayamos visto en los anteriores casos. En el bloque “else {...}” se lanza una excepción en todos aquellos casos que nos encontremos fuera del rango del “array”. La misma se captura en el bloque “catch (Exception mi_excepcion){...}”.

Como se puede observar, en el mismo, la excepción se gestiona (por medio de mostrar por pantalla un mensaje que nos informa de la misma, gracias a los métodos “toString(): String” y “getMessage (): String”), y, como se puede observar, la volvemos a lanzar (la propagamos), por medio del comando “throw mi_excepcion;”. Esto hace que la excepción sea “lanzada” a los clientes de nuestro método “acceso_por_indice (double [], int): double”, y que los mismos deban capturarla y gestionarla.

Por ejemplo, podríamos recuperar el “main” que presentamos en la Sección 5.3.1, que en este caso también funcionaría de forma satisfactoria:

```
public static void main (String [] args){
        double array_doubles [] = new double [500];

        for (int i = 0; i < 500; i++){
                array_doubles [i] = 7 * i;
        }
        for (int i = 0; i < 600; i = i + 25){
                try {
                        System.out.println ("El elemento en " + i + " es "
                                + acceso_por_indice (array_doubles, i));
                }
                catch (Exception e){
                        System.out.println (e.toString());
                }
        }
}
```

Cada llamada al método “acceso_por_indice (double [], int): double” está “encerrada” en un bloque “try {..} catch (...) {...}” que nos permite capturar y gestionar la excepción que nos puede mandar “acceso_por_indice (double [], int): double”.

5.3.4 CÓMO TRABAJAR CON EXCEPCIONES

Después de presentar las tres formas anteriores de trabajar con excepciones ahora correspondería indicar las ventajas y desventajas de cada una de ellas.

En general es difícil dar recetas universales sobre cómo trabajar con excepciones. Aún más, depende de si estamos programando métodos auxiliares que deben lanzar dichas excepciones, o aplicaciones que hacen uso de otros métodos que lanzan dichas excepciones.

Lo que sí podemos hacer es señalar cómo se hace, por lo general, en la librería (o API) de Java. En la API de Java generalmente los métodos existentes siguen la estrategia presentada en la Sección 5.3.1, es decir, la de generar excepciones y lanzarlas para que los métodos que los invoquen las gestionen como consideren oportuno. En cierto modo, los métodos de la API de Java siguen la "estrategia" de dejar que el usuario de tales métodos decida qué quiere hacer ante una situación excepcional, obligándole a programar bloques "try {...} catch (...) {...}" cuando haga uso de métodos de la librería que lanzan excepciones, o, en su defecto, a propagarlas indefinidamente (en Java existe incluso la posibilidad de propagarlas en la cabecera del método "main", con lo cual no deberíamos gestionarla; de nuevo, esta práctica no es muy recomendable, ya que delegamos en lo que el sistema quiera hacer con dicha excepción).

Aun así, la propia librería de Java también ha definido gran parte de sus excepciones como "RunTimeException", o herederas de dicha clase, de tal modo que las aplicaciones finales pueden evitar gestionar o capturar dichas excepciones (aunque pueden hacerlo si así lo consideran necesario).

En la siguiente Sección veremos cómo programar nuestras propias excepciones, así como un caso más general de captura y gestión de excepciones.

5.4 PROGRAMACIÓN DE EXCEPCIONES EN JAVA. UTILIZACIÓN DE EXCEPCIONES DE LA LIBRERÍA Y DEFINICIÓN DE EXCEPCIONES PROPIAS

Esta Sección estará dividida en dos partes. En la Sección 5.4.1 explicaremos cómo programar excepciones propias en Java. En la Sección 5.4.2 veremos un ejemplo un poco más elaborado que los anteriores donde presentaremos un caso de uso donde pueden aparecer un rango más amplio de excepciones.

5.4.1 PROGRAMACIÓN DE EXCEPCIONES PROPIAS EN JAVA

Aparte de todas las excepciones propias que existen en la librería de Java, que puedes encontrar a partir de http://java.sun.com/javase/6/docs/api/java/lang/Exception.html y explorando todas las subclases de las mismas, y de las cuales hemos presentado algunas en la Sección 5.2, en Java también es posible definir excepciones propias.

La metodología para lo mismo es sencilla. Sólo debemos declarar una clase que herede de la clase "Exception" en Java. De este modo, habremos creado una clase de objetos que pueden ser lanzados (o sea, que podemos utilizar con el comando "throw") y que por tanto pueden ser utilizados para señalar una situación anómala de cualquier tipo.

Generalmente, a la hora de declarar una nueva excepción, buscaremos un nombre que resulte descriptivo de la misma. Veamos cómo definir una clase de excepciones que nos sirva para representar la excepción que hemos lanzado en la Sección 5.3. El nombre que elegimos para la misma podría ser "IndiceFueraDeRangoExcepcion" (por cierto, la API de Java cuenta con una excepción que se puede usar en situaciones similares, llamada "IndexOutOfBoundsException").

Una decisión relevante que debemos tomar a la hora de definirla es si la misma debe heredar de la clase "Exception" o de la clase "RunTimeException". Como ya dijimos al introducir las excepciones en la Sección 5.2, caso de que la hagamos heredar de "Exception" tendremos la obligación de capturar y gestionar la misma (o de propagarla), mientras que si la hacemos heredar de "RunTimeException" el gestionar la misma pasará a ser opcional.

En este caso, la haremos heredar de "Exception". Veamos una posible definición para la misma:

```
class IndiceFueraDeRangoExcepcion extends Exception{

        public IndiceFueraDeRangoExcepcion (){
                super();
        }

        public IndiceFueraDeRangoExcepcion (String s){
                super(s);
        }
}
```

Como se puede observar, la definición de la clase es bastante sencilla. Lo más reseñable de la cabecera es la declaración de herencia que hemos realizado en la cabecera, "class IndiceFueraDeRangoExcepcion extends Exception{...}".

Posteriormente, hemos definido dos constructores para la misma, siguiendo el estilo de la definición de la clase "Exception" en Java (http://java.sun.com/javase/6/docs/api/java/lang/Exception.html). Dichos constructores únicamente invocan al constructor de la clase base. El resto de métodos de la clase "IndiceFueraDeRangoExcepcion" serán los heredados de la clase "Exception".

A partir de ahora, podremos hacer uso de la clase anterior como si fuese una excepción más del sistema. Por ejemplo, el método anterior "acceso_por_indice (double [], int): double" se podría programar ahora como:

```
public static double acceso_por_indice (double [] v, int indice) throws IndiceFueraDeRangoExcepcion{

        try {
                if ((0<=indice) && (indice <v.length)){
                        return v [indice];
```

```
                }
                else {
                        //Caso excepcional:
                        throw new IndiceFueraDeRangoExcepcion ("El indice " +
                                        indice + " no es una posicion valida");
                        }
        }
        catch (IndiceFueraDeRangoExcepcion mi_excepcion){
                System.out.println(mi_excepcion.toString());
                System.out.println(mi_excepcion.getMessage());
                throw mi_excepcion;
        }
}
```

Como podemos observar, el método no ha cambiado, salvo en que donde antes utilizábamos “Exception” ahora hemos pasado a usar “IndiceFueraDeRangoExcepcion”.

Por supuesto, al definir la clase “IndiceFueraDeRangoExcepcion” podíamos haber redefinido alguno de los métodos de la misma, o haber añadido métodos nuevos. Por ejemplo, podíamos haber redefinido el método “toString (): String” en la misma para que mostrase un mensaje distinto al que muestra en su definición por defecto:

```
class IndiceFueraDeRangoExcepcion extends Exception{
        public IndiceFueraDeRangoExcepcion (){
                super();
        }
        public IndiceFueraDeRangoExcepcion (String s){
                super(s);
        }
        public String toString (){
                return ("Se ha producido la excepcion " +
                                this.getClass().getName() + "\n" +
                        "Con el siguiente mensaje: " + this.getMessage() + "\n");
        }
}
```

Cuando ahora invoquemos al método ”toString (): String” sobre algún objeto de la clase “IndiceFueraDeRangoExcepcion” obtendremos una cadena como la siguiente:

```
Se ha producido la excepcion IndiceFueraDeRangoExcepcion
Con el siguiente mensaje: El indice 575 no es una posicion valida
```

Por último, recordamos de nuevo que si hubiéramos declarado la clase por herencia de la clase “RunTimeException” no hubiera sido necesario gestionar o capturar la misma.

En la siguiente Sección veremos un ejemplo más complejo de programación en Java con excepciones, relacionado con la salida y entrada de información desde ficheros.

5.4.2 UN EJEMPLO DESARROLLADO DE TRABAJO CON EXCEPCIONES DE LA API DE JAVA

En la siguiente Sección vamos a ver un ejemplo un poco más elaborado de trabajo con excepciones. No nos vamos a reducir a gestionar y capturar una simple excepción, sino que desarrollaremos un programa completo que haga la gestión de las mismas. Tampoco pretende ser una receta de cómo debe ser la gestión de excepciones, aunque sí que puede contener ideas útiles para la gestión de las mismas.

El programa resulta sencillo. Lo que hace es crear un fichero, de nombre "entrada_salida.txt", y después volcar varias cadenas de caracteres al mismo (objetos de tipo "String"), que luego el mismo programa se encargará de recuperar.

Las principales clases envueltas en el proceso serán la clase "File" (http://java.sun.com/javase/6/docs/api/java/io/File.html), que nos permite crear y gestionar ficheros, la clase "FileWriter" (http://java.sun.com/javase/6/docs/api/java/io/FileWriter.html), que nos permite abrir un fichero para realizar escritura en el mismo, la clase "BufferedWriter" (http://java.sun.com/javase/6/docs/api/java/io/BufferedWriter.html) que permite asociarle un "búfer" de escritura a un flujo de escritura (en este caso, al "FileWriter"), facilitando así las operaciones de escritura sobre el mismo, la clase "Scanner" (http://java.sun.com/javase/6/docs/api/java/util/Scanner.html) que ya conocemos, y que nos permitirá hacer lectura de distintos tipos de datos básicos desde un dispositivo de entrada (por ejemplo, un fichero, o la consola de MSDOS), y por último, las clases "wrapper" o envoltorio de alguno de los tipos básicos de Java, como "Double" (http://java.sun.com/javase/6/docs/api/java/lang/Double.html) o como "Integer" (http://java.sun.com/javase/6/docs/api/java/lang/Integer.html).

Las demás clases que aparecerán en el programa serán las excepciones propias de cada uno de los métodos que utilicemos de cada una de las clases anteriores:

```
import java.io.*;
import java.util.Scanner;
import java.util.NoSuchElementException;

public class Ejemplo_Ficheros{

        public static void main (String [] args){

                try{
                        File file = new File("entrada_salida.txt");
```

```
//Lanza NullPointerException, si la cadena es vacía

// Crea el fichero si no existe
boolean success = file.createNewFile();
//Lanza IOException o SecurityException

if (success) {
        // El fichero no existe y se crea:
        System.out.println("El fichero no existe y se crea");

        //Comprueba que el fichero se puede escribir y leer:
        System.out.println ("El fichero se puede escribir "
                            + file.canWrite());
        System.out.println ("El fichero se puede leer "
                            + file.canRead());

        //Le asociamos al fichero un búfer de escritura:
        BufferedWriter file_escribir =
                new BufferedWriter (new FileWriter (file));
        //Lanza IOException

        //Escribimos cadenas de caracteres en el fichero
        //Separadas por saltos de líneas:
        file_escribir.write("Una primera sentencia:");
        //Lanza IOException
        file_escribir.newLine();
        file_escribir.write("8.5");
        //Lanza IOException
        file_escribir.newLine();
        file_escribir.write("6");
        //Lanza IOException
        file_escribir.newLine();
        file_escribir.flush();//Lanza IOException
        file_escribir.close();//Lanza IOException

        //Abrimos ahora el fichero para lectura
        //por medio de la clase Scanner:
        Scanner file_lectura =
                new Scanner (file);
        //Lanza FileNotFoundException

        //Leemos cadenas de caracteres del mismo:
        String leer = file_lectura.nextLine();
        //Lanza IllegalStateException
        //o NoSuchElementException
        String leer2 = file_lectura.nextLine();
        //Lanza IllegalStateException
        //o NoSuchElementException
        String leer3 = file_lectura.nextLine();
        //Lanza IllegalStateException
```

```
                        //o NoSuchElementException

                        //Intentamos convertir cada cadena
                        //a su tipo de dato original:
                        double leer_double;
                        int leer_int;
                        leer_double = Double.parseDouble(leer2);
                        //Lanza NumberFormatException
                        leer_int = Integer.parseInt (leer3);

                        //Mostramos por la consola las diversas cadenas:
                        System.out.println ("La cadena leida es " + leer);
                        System.out.println ("El double leido " + leer_double);
                        System.out.println ("El entero leido " + leer_int);

                }
                else {
                    // El fichero ya existe:
                    System.out.println("El fichero ya existe y no se creo");
                }
        }
catch (FileNotFoundException f_exception) {
        //Excepcion que surge si no encontramos el fichero
        //al crear el Scanner
        System.out.println ("Las operaciones de lectura no
                            se han podido llevar a cabo,");
        System.out.println ("ya que ha sucedido un problema
                            al buscar el fichero para lectura");
        System.out.println (f_exception.toString());
}
catch (IOException io_exception){
        //Excepcion que puede surgir en
        //alguna de las operaciones de escritura
                System.out.println("Ocurrio algun error de entrad y salida");
                System.out.println (io_exception.toString());
}
catch (NumberFormatException nb_exception){
        //Excepcion que ocurre al realizar una conversion de una cadena
        //de caracteres a un tipo numerico
        System.out.println ("Ha ocurrido un error de
                            conversión de cadenas a numeros");
        System.out.println (nb_exception.toString());
}
catch (NoSuchElementException nse_exception){
                //Excepcion que ocurre cuando el metodo
                //"nextLine(): String" no encuentra una cadena
                System.out.println ("No se ha podido encontrar el
                                    siguiente elemento del Scanner");
                System.out.println (nse_exception.toString());
}
```

```
        catch (Exception e_exception){
                //Un ultimo bloque que nos permita
                //capturar cualquier tipo de excepcion
                System.out.println (e_exception.toString());
        }
}
}
```

Algunas consideraciones se pueden hacer sobre el anterior fragmento de código:

1. En primer lugar, con respecto a la estructura del mismo, se puede observar como hemos creado un bloque “try {...}” en el que hemos incluido la mayor parte de los comandos del programa. El mismo está seguido de una secuencia de comandos “catch (...) {...}” que capturan y gestionan cada una de las excepciones que pueden surgir del bloque “try {...}”.

2. Tal y como hemos programado los bloques anteriores, en cuanto surja una excepción cualquiera en un método, el flujo del programa se dirigirá a los bloques “catch (...) {...}” que finalizan el mismo, por lo que el programa habrá terminado su ejecución (frente a esto, podíamos haber optado por ir gestionando las excepciones junto a las llamadas a los métodos, haciendo que el flujo del programa continuara aun después de tener lugar una situación excepcional). Como ya hemos mencionado con anterioridad, tal decisión depende del programador de la aplicación y del diseño que hagamos de la misma.

A modo de conclusión del Tema, cabe decir que el uso de excepciones no evita los errores en programación. Su utilidad está relacionada más bien con informar sobre los errores o situaciones excepcionales que se producen durante la ejecución de un programa, y por eso su uso depende en gran medida del programador de cada aplicación, y de cómo el mismo quiera tener constancia de esos errores. Por lo tanto, deben ser entendidas como un sistema para informar sobre errores que han ocurrido en una aplicación, y no como una solución a los errores en la programación.