INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Curso online: **PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA (TELEFORMACIÓN - ON LINE)**

Tema 5: EXCEPCIONES Y ENTRADA/SALIDA EN JAVA

Autores

Jorge Molinero Muñoz   Miguel Sallent Sánchez

ÍNDICE

1. Excepciones 3

1.1. Clasificación de Excepciones 3

1.2. Jerarquía de Excepciones 4

1.3. Tratamiento de errores 5

1.4. Captura y manejo de Excepciones 6

1.5. Generación de Excepciones 7

1.6. Creación de clases de Excepciones 7

2. Entrada/Salida 8

2.1. El concepto de Stream 8

2.2. Streams de bytes 9

2.3. Streams de caracteres 10

3. Ejercicios prácticos 12

3.1. Ejercicio Resuelto: Lectura por teclado y tratamiento de excepciones 12

1. Excepciones

Una excepción es un evento, que ocurre durante la ejecución de un programa y rompe el flujo normal de instrucciones de dicho programa.

Gracias a las excepciones, el posible gestionar los posibles errores que se pueden producir en un programa, controlando el comportamiento de este en dichas situaciones.

Cuando se produce un error dentro de un método o programa, se crea un objeto Exception, que contiene información sobre el error, y se propaga dicho objeto al sistema, rompiendo el flujo natural de ejecución. Este objeto Exception, puede ser capturado por nuestro programa o en el caso de no ser capturado, provocar que el programa termine.

1.1. Clasificación de Excepciones

Existen tres tipos de excepciones

- Excepciones controladas ("Checked Exception") Son errores previstos y que están controlados por el código, de modo que el sistema puede recuperarse ante dichas situaciones.

  Todas estas excepciones deberán ser capturadas o propagadas, es decir, cuando en un punto de un programa se pueda generar una excepción de este tipo, dicha sección de código deberá estar delimitada por un bloque try-catch o deberá pertenecer a un método que indique explícitamente que puede generar excepciones.

- Errores. Situaciones de error excepcionales debido a elementos externos al programa y que no pueden ser previstas. Este tipo de excepción no tiene sentido que sea controlada, ya que se dan cuando se producen situaciones graves frente a las cuales el programa no puede actuar.

  Por ejemplo: Errores de la máquina virtual de Java y errores de falta de memoria.

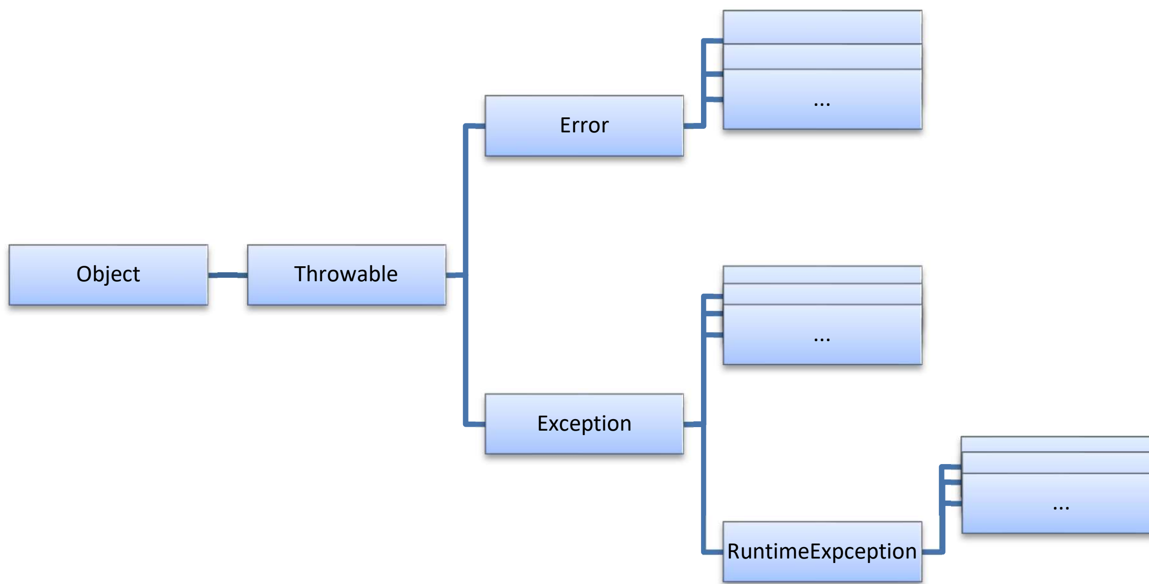
- Excepciones en tiempo de ejecución ("Runtime Exception"). Son situaciones de error excepcionales e internas al sistema. Generalmente son provocadas por errores en la programación. Por lo tanto este tipo de excepciones deberían evitarse por medio de una correcta programación. No es necesario declarar o capturar estas excepciones, ya que es mucho más costoso su gestión a través de excepciones que evitar dichas situaciones por código.

Excepciones Controladas	• Situaciones previstas • Deben ser capturadas o especificadas • Controlados por el programa
Errores	• Situaciones no previstas por factores externos • No deben ser capturados • No es posible controlarlos
Excepciones en tiempo de ejecución	• Situaciones motivadas por mala programación • No deben ser capturadas • Deben evitarse programáticamente

1.2. Jerarquía de Excepciones

Como ya se ha mencionado anteriormente, una excepción es un objeto de un tipo particular de clase, que permite romper la ejecución normal del código. Por lo tanto, Java representa las excepciones como clases, las cuales tienen su propia jerarquía para los distintos tipos de excepciones.

A continuación se presenta un diagrama con parte de la jerarquía de clases:



Tal y como se observa, la jerarquía de clases representa los tipos de excepciones anteriormente mencionados:

- Excepciones controladas: Aquellas que heredan de Exception
- Errores: Heredan de Error
- Excepciones en tiempo de ejecución: Heredan de RuntimeExcepcion

Todas las excepciones y errores, son subclases de *Throwable*. Esta clase indica que puede ser lanzada, rompiendo el flujo de ejecución normal del programa. *Throwable* contiene una foto de la pila del sistema en el momento que se crea y un mensaje con la información de aquello que ha provocado el error.

La clase *Throwable* sólo puede ser lanzada por la máquina virtual de Java o a través de la sentencia *throw*, tal y como se verá más adelante.

1.3. Tratamiento de errores

Cuando en una sección de código se puede producir una excepción, existen dos alternativas:

• Tratar la excepción, capturándola y actuando oportunamente. Mediante un bloque try...catch.

• Propagarla al método que invocó dicho código. A través de la sentencia throws.

1.4. Captura y manejo de Excepciones

Cuando se prevé que una sección de código puede generar una excepción y se desea tratar, se delimita dicho código a través de la sentencia ***try***, seguida de al menos una cláusula ***catch*** (que encierra el código manejador) o una cláusula ***finally***.

La sintaxis es:

```
try
{
      // código que puede provocar una excepción
}
catch( TipoExcepcion  variable)
{
      // código manejador de ese tipo de excepción
}
finally
{
      // sentencias
}
```

El sistema ejecuta el código delimitado por llaves en el bloque *try.*

- Si el código no provoca ninguna excepción, no se ejecuta la cláusula catch.
- Si el código provoca alguna excepción, se abandona la ejecución del bloque *try* y se pasa a ejecutar el código de la cláusula *catch*.

La cláusula *catch* especifica el tipo de excepciones que captura, cuando se produce una excepción dentro del *try*, se comprueba si dicha excepción es del tipo indicado o es una excepción hija de esta, y si es así, se ejecuta el código de la cláusula catch.

Después de un bloque *try*, pueden aparecer varias cláusulas *catch* (al menos una). El orden en el cual van colocadas es importante, ya que en el caso de que se produzca una excepción, se evalúan en el mismo orden en que aparecen, ejecutando el código de la primera cláusula *catch* que capture la excepción.

Para escribir varias cláusulas *catch* se utilizará la siguiente estructura:

```
try {
      sentencias;
}
catch  (TipoExcepcion1) {
      sentencias;
}
catch  (TipoExcepcion2) {
      sentencias;
}
...   // más cláusulas catch
/*opcional*/ finally { ... }
```

Todas las excepciones que no pertenezcan a la jerarquía *RuntimeException* deberán ser capturadas o en su caso propagadas.

La cláusula *finally*, es opcional, el código contenido en dicha cláusula se ejecutará siempre al final, se produzca o no una excepción en el bloque *try*. Si se produjera una excepción, se ejecutaría la cláusula *catch* que capture la excepción y finalmente la cláusula *finally*. Por ejemplo, en el bloque finally se podría realizar el cierre de una conexión a la base de datos, para que nunca se queden las conexiones abiertas, aunque se haya producido un error en la ejecución.

1.5. Generación de Excepciones

Si durante la ejecución de un método, se desea producir un error, que queremos notificar en forma de excepción. Se debe:

1. Crear un objeto excepción de la clase apropiada.

2. Lanzar dicho objeto.

La sintaxis es la siguiente:

throw new ConstructorExcepción(parametros);

Por ejemplo, si se está leyendo un fichero y se recibe el fin de fichero antes de lo esperado. Podemos lanzar una excepción de tipo IOException, y buscando en su árbol de herencia descubrimos la clase *EOFException* (Excepción utilizada para indicar que se alcanzado el final de fichero inesperado)

El código sería:

```
throw new EOFException( );
```

o también:

```
EOFException e = new EOFException( );

throw e;
```

1.6. Creación de clases de Excepciones

Gracias a que las excepciones son clases, Java permite crear clases de Excepciones a medida para nuestro sistema, para ello se debe implementar una clase que herede de *Excepcion* o alguna de sus clases hijas.

Es conveniente que la clase presente al menos dos constructores, uno por defecto y otro que reciba un mensaje detallado de lo que ocurrió.

```
class ExcepcionAMedida extends Exception // o cualquier clase hija
{
   public ExcepcionAMedida ( ){
   }
   public ExcepcionAMedida(String descripcion) {
      super(descripcion);
   }
}
```

2. Entrada/Salida

Hasta ahora hemos visto cómo crear programas en Java, pero sería poca la funcionalidad que obtendríamos de ellos si no fuésemos capaces de procesar datos externos. La heterogeneidad de las fuentes y destinos de datos (no es en absoluto lo mismo tratar con los datos procedentes de un teclado que con los procedentes de una red) haría muy compleja la programación de no ser por-que Java dispone de una jerarquía de clases que abstrae todo lo referente a la entrada/salida.

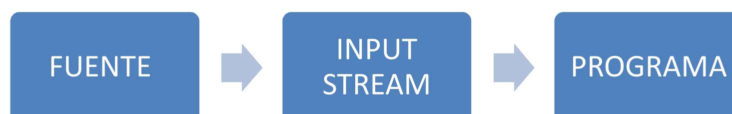
Esta abstracción se concreta en lo que se denominan ***streams*** (flujos), que veremos a continuación.

2.1. El concepto de Stream

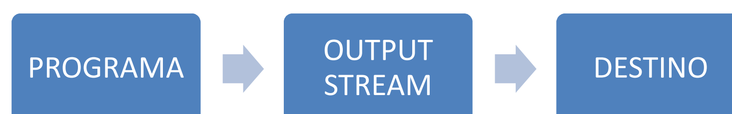
Un ***stream*** representa una fuente o un destino de datos, como por ejemplo ficheros en un disco, dispositivos de entrada como un teclado, una conexión http, etc.

Hay dos tipos de *streams*:

- *Stream* de entrada: el programa **lee** los datos del *stream*. Por tanto el *stream* es el origen y el programa es el destino.



- *Stream* de salida: el programa **escribe** los datos en el *stream*. Por tanto el programa es el origen y el *stream* el destino.



Los *streams en Java* son un mecanismo poderoso porque proporcionan una interfaz de acceso a las operaciones de entrada/salida a través de una serie de métodos que ocultan los detalles de manejo de los diferentes dispositivos. Los *streams* soportan distintos tipos de datos, como bytes, caracteres de texto, tipos primitivos e incluso objetos.

2.2. Streams de bytes

Los *streams* de bytes se utilizan para leer o escribir *bytes.* Todos los *streams* de bytes heredan de las clases InputStream y OutputStream. Por ejemplo, cuando queremos leer de un fichero en disco utilizaremos FileInputStream, y cuando queramos escribir, FileOutputStream.

Los métodos principales son *read* para la lectura y *write* para la escritura. Veremos cómo funcionan en el siguiente ejemplo: el programa lee los bytes de un fichero y los escribe en otro fichero:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class EjemploStreams {
     public static void main(String[] args) throws IOException {

          FileInputStream in = null;
          FileOutputStream out = null;

          try {
              in = new FileInputStream("prueba.txt");
              out = new FileOutputStream("copia.txt");
              int c;

              while ((c = in.read()) != -1) {
                   out.write(c);
              }
          } finally {
              if (in != null) {
                   in.close();
              }
              if (out != null) {
                   out.close();
              }
          }
     }
}
```

Este programa va leyendo los bytes del fichero de origen de uno en uno, y a medida que va leyendo va escribiendo en el fichero destino.

Es importante cerrar los *streams* cuando ya no van a ser utilizados, mediante el método *close()*. Esto sirve para liberar los recursos reservados cuando tenemos el *stream* abierto.

La ventaja de usar *streams* es que como todos heredan de InputStream y OutputStream, todos implementan los mismos métodos, de forma que si cambia la fuente de datos podemos sustituir un *stream* por otro y el resto del programa no necesitaría ser modificado.

2.3. Streams de caracteres

Imaginemos que queremos leer un fichero de texto como el del ejemplo anterior para luego procesar de alguna manera el texto que contiene. Si utilizamos FileInputStream, nos encontraremos con que leemos bytes, pero lo que necesitamos para procesarlo son caracteres. Nuestro programa se vería obligado a convertir de *byte* a *char*. Para evitar esto disponemos de los *streams* de caracteres, que nos permiten trabajar directamente con caracteres de texto en lugar de con bytes. Todos los *streams* de caracteres heredan de las clases Reader y Writer.

Ahora veremos otro ejemplo similar al anterior, en el que leemos un fichero de texto, lo convertimos a mayúsculas, y escribimos en otro fichero.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class EjemploStreams {
     public static void main(String[] args) throws IOException {

          FileReader in = null;
          FileWriter out = null;

          try {
              in = new FileReader("prueba.txt");
              out = new FileWriter("copia.txt");
              int c;

              while ((c = in.read()) != -1) {
                   char leido = (char)c;
                   out.write(String.valueOf(leido).toUpperCase());
              }
          } finally {
              if (in != null) {
                   in.close();
              }
              if (out != null) {
                   out.close();
              }
          }
     }
}
```

En este caso, el método *read* también devuelve un *int*, pero ahora la variable contiene un valor de carácter en sus últimos 16 bits (recordemos que java internamente utiliza la codificación Unicode para codificar el texto, y que esta codificación utiliza dos bytes). Por el contrario, el método *read* de InputStream devuelve un *int* que contiene un valor de byte en sus últimos 8 bits.

El método *read* de la clase Reader podría devolver un *char* en vez de un *int,* pero se utiliza *int* para detectar cuándo se alcanza el final del *stream* (en ese caso devuelve -1).

Como podemos imaginar este ejemplo no es muy eficiente, ya que vamos leyendo y escribiendo carácter a carácter. En general trataremos de leer y escribir en bloques para que el rendimiento sea mejor, para lo cual disponemos de varias clases. Ahora vamos a modificar el ejemplo anterior utilizando las clases BufferedReader y PrintWriter. La clase BufferedReader almacena en un buffer interno los caracteres que va leyendo, de manera que las operaciones de lectura sobre el fichero físico no son carácter a carácter, sino en bloques, mejorando por tanto el rendimiento. La clase PrintWriter nos permite escribir la representación textual de distintos tipos de objetos. Además el método *println* escribe un “fin de línea” al final, que es preferible a escribir directamente “\n” o “\r\n”, ya que el fin de línea puede variar de un sistema de codificación a otro. Usando *println* java se encarga de utilizar la codificación correcta en todos los casos.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class EjemploStreams {
	public static void main(String[] args) throws IOException {

		BufferedReader in = null;
		PrintWriter out = null;

		try {
			in = new BufferedReader (new FileReader("prueba.txt"));
			out = new PrintWriter(new FileWriter("copia.txt"));

			String linea;
			while ((linea = in.readLine()) != null) {
				out.println(linea.toUpperCase());
			}
		} finally {
			if (in != null) {
				in.close();
			}
			if (out != null) {
				out.close();
			}
		}
	}
}
```

3. Ejercicios prácticos

3.1. Ejercicio Resuelto: Lectura por teclado y tratamiento de excepciones

A continuación se presentará un ejercicio de ejemplo, donde se observa cómo trabajar con excepciones.

Se crearán cuatro clases de excepciones que representen respectivamente:

- Excepción de vocal
- Excepción de número
- Excepción de blanco
- Excepción de salida.

Estas excepciones serán utilizadas por una clase que leerá caracteres procedentes del teclado.

La clase deberá tener contener:

• **Un atributo de la clase LeerTeclado** que realiza las lecturas de caracteres a través de su método *getChar( )*.

• **Un método que procese el carácter leído y - dependiendo de su valor - lance una excepción** de uno de los distintos tipos. Si se lee una vocal, lanzará "Excepción de vocal", si es un número "Excepción de número", si es un blanco "Excepción de blanco" y si es el carácter x "Excepción de salida". Este método debe manejar el atributo de la clase LeerTeclado con el fin de que éste almacene el carácter leído.

• **Un método main responsable de leer caracteres y capturar las posibles excepciones que se produzcan**, tratándolas como se indica a continuación:

- Si la excepción corresponde a una vocal, a un número o a un blanco, se mostrará un mensaje representativo del tipo de excepción y se continuará leyendo caracteres.

- Si la excepción se debe al carácter de salida, se mostrará un mensaje pertinente y se abandonará el programa.

Caso Práctico:

```
import java.io.IOException;


// Definición de excepciones empleadas por la aplicación
class ExcepcionDeVocal extends Exception {
}

class ExcepcionDeNumero extends Exception {
}
```

```
class ExcepcionDeBlanco extends Exception {
}

class ExcepcionDeSalida extends Exception {
}

class PruebaDeExcepciones {
	// atributos
	LeerTeclado kbd = new LeerTeclado(System.in);

	// métodos
	public void procesarEntradaUsuario() throws ExcepcionDeVocal,
			ExcepcionDeBlanco, ExcepcionDeSalida, ExcepcionDeNumero {
		System.out.print("Introduzca un carácter: ");
		System.out.flush();
		char ch;

		try {
			ch = Character.toUpperCase(kbd.getChar());
		} // try
		catch (IOException x) {

			System.out.println("Se ha producido una IOException.");
			return;
		} // catch

		switch (ch) {
		case 'A':
		case 'E':
		case 'I':
		case 'O':
		case 'U':
			throw new ExcepcionDeVocal();
		case '1':
		case '2':
		case '3':
		case '4':
		case '5':
		case '6':
		case '7':
		case '8':
		case '9':
		case '0':
			throw new ExcepcionDeNumero();
		case ' ':
			throw new ExcepcionDeBlanco();
		case 'X':
			throw new ExcepcionDeSalida();
		} // switch
	} // procesar_Entrada_Usuario

	public static void main(String args[]) {

	   PruebaDeExcepciones miPrueba = new PruebaDeExcepciones();
	   boolean fin = false;
	   do {
	      try {
```

```
                miPrueba.procesarEntradaUsuario();
            } catch (ExcepcionDeVocal x) {
                System.out.println("Se ha producido excepción de vocal.");
            } catch (ExcepcionDeNumero x) {
                System.out.println("Se ha producido excepción de número.");
            } catch (ExcepcionDeBlanco x) {
                System.out.println("Se ha producido excepción de blanco.");
            } catch (ExcepcionDeSalida x) {
                System.out.println("Se ha producido excepción de salida.");
                fin = true;
            } finally {
                System.out.println("Estamos en la cláusula finally.\n");
            } // finally
        } while (!fin);
    } // main
} // class
```

A continuación se presenta la clase *LeerTeclado*:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;

public class LeerTeclado {

	// atributos

	private Reader stream;

	// métodos

	/**
	 * Constructor
	 *
	 * @param fuente la fuente de datos
	 */
	public LeerTeclado(InputStream fuente) {
		stream = new InputStreamReader(fuente);
	}

	/**
	 * Obtiene el siguiente carácter del teclado.
	 *
	 * @return el carácter escrito
	 * @throws IOException
	 */
	public char getChar() throws IOException {
		return (char) this.stream.read();
	}
}
```