FAMILIA PROFESIONAL:	**Informática y Comunicaciones**
CICLOS FORMATIVOS:	**Desarrollo de Aplicaciones Multiplataforma, Desarrollo de Aplicaciones Web**
MÓDULO:	**Programación**

UNIDAD 7: TRATAMIENTO DE ERRORES Y EXCEPCIONES

CONTENIDOS

AUTORES: **Fernando Rodríguez Alonso**
Sonia Pasamar Franco

ÍNDICE DE CONTENIDOS

1. ERRORES Y EXCEPCIONES

Una **situación anómala en la ejecución de un programa** impide que se siga ejecutando el flujo de instrucciones del programa.

Cuando ocurre una situación anómala, se necesita información para corregirla:

- Si en el ámbito donde se produce la situación anómala no se dispone de la información necesaria para solucionar la situación y continuar la ejecución del programa, se transferirá el control del flujo del programa a otro ámbito (al método que lo ha invocado) en el que se disponga de más información y en el que quizás sea posible manejar esa situación anómala.
- Si el programa principal (método `main`) no tiene la información necesaria para tratar la situación anómala, se transferirá el control a la Máquina Virtual de Java, que realizará lo siguiente:
  - Se escribirá en consola un mensaje de error indicando el tipo de error y la causa del error.
  - Se escribirá en consola la lista (también denominada pila) de invocaciones a métodos que ha producido la situación anómala, especificando concretamente ficheros fuente, métodos y líneas.
  - Se terminará la ejecución del programa.

1.1. TIPOS DE SITUACIONES ANÓMALAS

En Java, cualquier situación anómala en la ejecución de un programa genera directa o indirectamente un objeto de la clase **`Throwable`**. Esta clase representa todo lo que se puede "lanzar" en Java:

- Contiene una instantánea del estado de la pila (denominado también *stack trace* o *call chain*) en el momento en el que se creó el objeto.

  El método **`printStackTrace`** escribe en consola la pila de invocaciones que ha ocasionado la situación anómala.

- Almacena un mensaje informativo indicando el nombre de la situación anómala. Puede incluir una causa que permita representar el origen de la situación anómala.

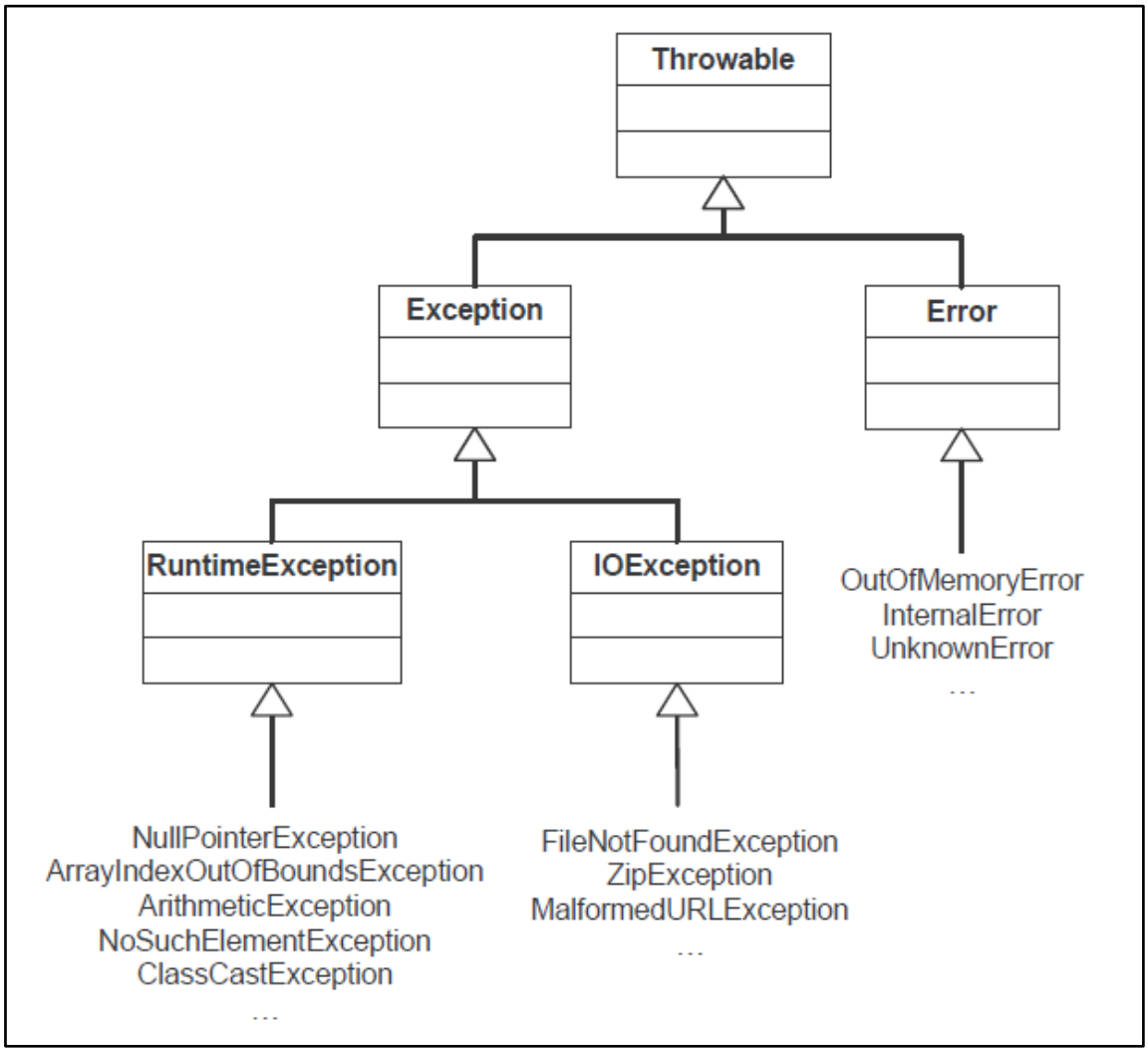
  El método **`getMessage`** devuelve el mensaje informativo asociado a la situación anómala.

  El método **`toString`** devuelve el nombre de la situación anómala junto con el mensaje devuelto por el método `getMessage`.

La clase **`Throwable`** tiene dos grandes subclases definidas para los **tipos de situaciones anómalas en ejecución**:

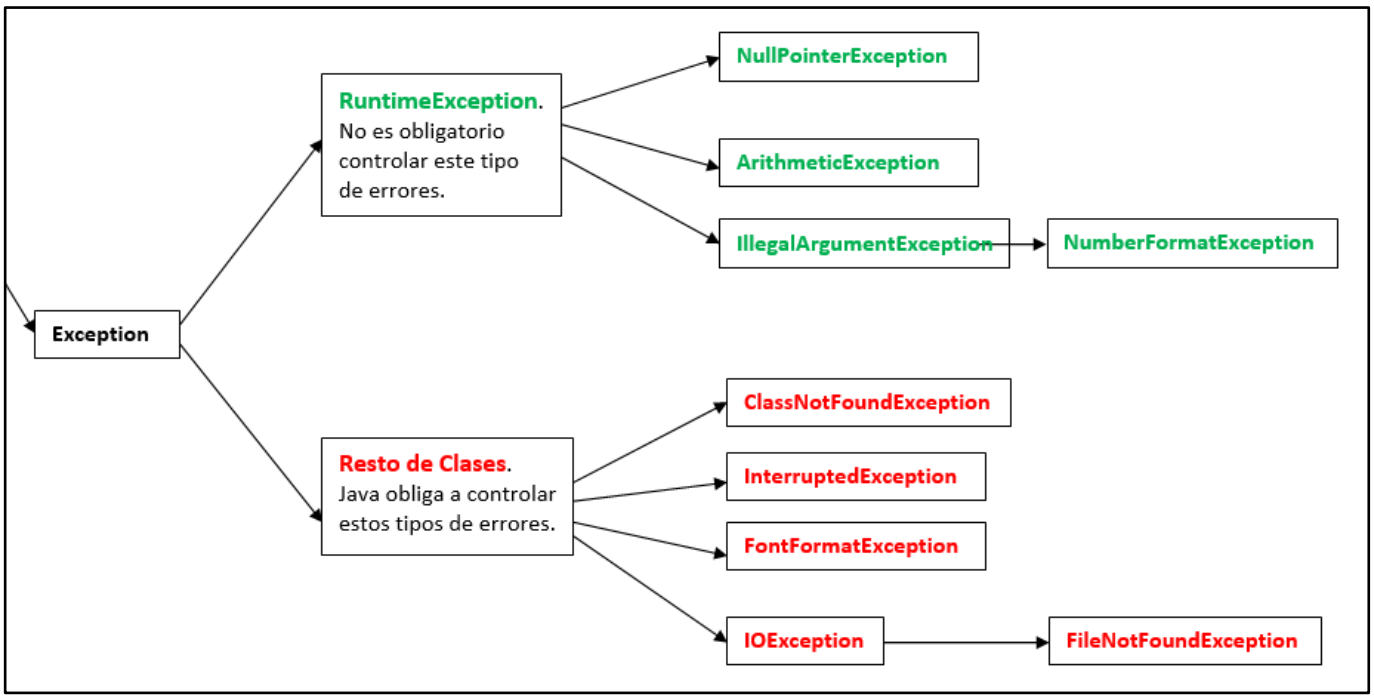
- **`Error`.** Indica que se ha producido un fallo irrecuperable, para el que es imposible recuperar y continuar la ejecución del programa. La Máquina Virtual de Java presentará un mensaje informativo en el dispositivo de salida y finalizará la ejecución del programa. Para este tipo de situaciones, el programador no podrá hacer nada.
- **`Exception`.** Indica una situación anormal, pero que puede corregirse y reconducirse para que el programa no tenga que terminar forzosamente. Java proporciona cerca de 70 subclases directas de la clase `Exception` ya predefinidas, cada una para un tipo de situación anómala concreta. Además, cada subclase suele tener a su vez otras subclases, que definen excepciones más concretas y específicas.

El siguiente diagrama muestra la **jerarquía de clases para el manejo de situaciones anómalas** en Java:



1.2. JERARQUÍA DE EXCEPCIONES

El siguiente diagrama muestra la **jerarquía de clases para el manejo de excepciones** en Java, incluyendo las excepciones más relevantes:



Existen dos **tipos de excepciones** en Java:

- **`RuntimeException`.** Define una excepción que no es obligatorio controlar en un programa. Se utiliza para tratar errores del programador, como por ejemplo, una división por cero o el acceso fuera de los límites de un vector.
- **Resto de Excepciones (como `IOException`).** Define una excepción que Java obliga a controlar en un programa. Se utiliza para tratar errores que no puede evitar el programador, como por ejemplo, los relacionados con la entrada/salida del programa.

2. TRATAMIENTO DE EXCEPCIONES

Cuando se produce una **excepción** en un método durante la ejecución de un programa, existen dos maneras básicas de tratarla: capturar la excepción en el método donde se ha producido o propagar la excepción al método desde el cual se realizó la invocación al método causante.

También se puede lanzar una excepción específica de forma explícita en situaciones particulares y se pueden crear nuevos tipos de excepciones a partir de otras ya definidas para problemas concretos.

2.1. CAPTURA DE EXCEPCIONES

La **captura de excepciones** consiste en tratar una excepción en el mismo ámbito o bloque donde ha ocurrido. En Java se realiza englobando el código fuente que puede producir excepciones con la **estructura `try-catch`**:

- Cuando se produce una excepción dentro del bloque **`try`**, se detiene la ejecución de este bloque de instrucciones. Es decir, no se ejecutan las siguientes instrucciones a la instrucción que ha producido la excepción.
- El bloque **`catch`** recibe como parámetro un objeto de la clase `Throwable`. Se comprueba que el tipo de la excepción coincida con el tipo de excepción indicado en el bloque **`catch`**. Si es así, se ejecutan las instrucciones contenidas en este bloque y se considera la excepción capturada y tratada.
- Si la excepción se ha capturado y tratado con la estructura **`try-catch`**, el programa continua su ejecución a partir de la instrucción siguiente a esta estructura.

El siguiente ejemplo muestra la **captura de una excepción en Java** con una estructura `try-catch`:

CAPTURA DE EXCEPCIONES	CASOS

`// Bloque 1`
`try {`
`  // Bloque 2`
`}`
`catch (Exception e) {`
`  // Bloque 3`
`}`
`// Bloque 4`
	Sin excepciones:
	`1 → 2 → 4`
	Con una excepción en el bloque 2:
	`1 → 2* → 3 → 4`
	Con una excepción en el bloque 1:
	`1*`

2.1.1. Múltiples Cláusulas catch

En una estructura `try-catch`, debe haber un único bloque `try` y un bloque `catch` al menos. Puede haber varios bloques `catch`, cada uno encargado de un tratamiento concreto para un determinado tipo distinto de excepción. No se permite un bloque `try` sin ningún bloque `catch`, ni un bloque `catch` sin ningún bloque `try`.

Además, el tratamiento de cada excepción posible mediante un bloque `catch` debe contener siempre instrucciones: el bloque `catch` no puede quedar sin tratamiento, no debe estar vacío.

El siguiente ejemplo muestra la **captura de una excepción en Java** con una estructura `try-catch` con varios `catch`:

CAPTURA DE EXCEPCIONES	CASOS

`// Bloque 1`
`try {`
`  // Bloque 2`
`}`
`catch (ArithmeticException ae) {`
`  // Bloque 3`
`}`
`catch (NullPointerException npe) {`
`  // Bloque 4`
`}`
`// Bloque 5`
	Sin excepciones:
	1 → 2 → 5
	Con una excepción de tipo aritmético en el bloque 2:
	1 → 2* → 3 → 5
	Con una excepción de tipo acceso a objeto nulo en el bloque 2:
	1 → 2* → 4 → 5
	Con una excepción de otro tipo diferente en el bloque 2:
	1 → 2*

El siguiente ejemplo muestra la **captura de una excepción en Java** con una estructura `try-catch` con varios `catch`:

CAPTURA DE EXCEPCIONES	CASOS

`// Bloque 1`
`try {`
`  // Bloque 2`
`}`
`catch (ArithmeticException ae) {`
`  // Bloque 3`
`}`
`catch (Exception e) {`
`  // Bloque 4`
`}`
`// Bloque 5`
	Sin excepciones:
	1 → 2 → 5
	Con una excepción de tipo aritmético en el bloque 2:
	1 → 2* → 3 → 5
	Con una excepción de otro tipo diferente en el bloque 2:
	1 → 2* → 4 → 5

El siguiente ejemplo muestra la **captura de una excepción en Java** con una estructura `try-catch` con varios `catch`:

CAPTURA DE EXCEPCIONES	CASOS

`// Bloque 1`
`try {`
`  // Bloque 2`
`}`
`catch (Exception e) {`
`  // Bloque 3`
`}`
`catch (ArithmeticException ae) {`
`  // Bloque 4`
`}`
`// Bloque 5`
	Sin excepciones:
	1 → 2 → 5
	Con una excepción de tipo aritmético en el bloque 2:
	1 → 2* → 3 → 5
	Con una excepción de otro tipo diferente en el bloque 2:
	1 → 2* → 3 → 5

Cuando una estructura `try-catch` contiene varios bloques `catch`, estos se evalúan en orden. Si se produce una excepción y se captura en un bloque `catch`, después de tratar la excepción en dicho catch, los siguientes bloques `catch` no se comprueban y termina la ejecución de la estructura.

Dado que las excepciones están definidas en una jerarquía de clases, cuando una estructura try-catch incluya varios bloques `catch`, estos deben codificarse en orden de más específico a más genérico. Es decir, las excepciones de tipo más específico (subclases) se colocan antes que las excepciones de tipo más genérico (superclases).

2.1.2. La Cláusula finally

En ocasiones, interesa ejecutar un fragmento de código independientemente de si se produce o no una excepción. Se utiliza para garantizar la liberación de los recursos que el bloque `try` ha acaparado de forma exclusiva y que podrían quedar definitivamente retenidos por este bloque si no se ejecutan las correspondientes instrucciones de liberación, como en el caso de que se produzca una excepción antes de ejecutar las instrucciones encargadas de liberar estos recursos y se transfiera el control al primer bloque `catch`.

Dos ejemplos típicos en los que se utilizan **estructuras `try-catch-finally`** son:

- En el **acceso a ficheros**, el bloque `finally` se usa para cerrar los ficheros abiertos.
- En el **acceso a bases de datos**, el bloque `finally` se usa para cerrar las conexiones establecidas con las bases de datos.

En una estructura `try-catch`, el bloque `finally` es opcional y se coloca detrás del último bloque `catch`. Cuando se incluye este bloque `finally` en la estructura `try-catch`, se permite que no haya ningún bloque `catch` en la estructura.

El siguiente ejemplo muestra la **captura de una excepción en Java** con una estructura `try-catch` con un `finally`:

CAPTURA DE EXCEPCIONES	CASOS

`// Bloque 1`
`try {`
`  // Bloque 2`
`}`
`catch (ArithmeticException ae) {`
`  // Bloque 3`
`}`
`finally {`
`  // Bloque 4`
`}`
`// Bloque 5`
	Sin excepciones:
	1 → 2 → 4 → 5
	Con una excepción de tipo aritmético en el bloque 2:
	1 → 2* → 3 → 4 → 5
	Con una excepción de otro tipo diferente en el bloque 2:
	1 → 2* → 4

En una estructura `try-catch-finally`, cuando comienza la ejecución del bloque `try`, el bloque `finally` siempre se ejecutará:

- Después del bloque `try`, si no se producen excepciones.
- Después de un bloque `catch`, si se produce una excepción y este bloque `catch` realiza su captura y tratamiento.
- Justo después de que se produzca una excepción y antes de que la excepción se propague a otro método, si ningún bloque `catch` captura la excepción.

2.2. PROPAGACIÓN DE EXCEPCIONES

La **propagación de excepciones** consiste en pasar o enviar una excepción desde el método donde ha ocurrido la excepción al método que lo ha llamado para que este método trate la excepción. En Java se realiza añadiendo a la cabecera del método una cláusula **`throws`** que incluya una lista de los tipos de excepciones que se pueden producir al invocar el método.

En la cláusula `throws` de la cabecera de un método no es necesario declarar las excepciones de tipo **`RuntimeException`**, ya que este tipo de excepciones se propagan de forma automática al método que lo ha invocado.

En la cláusula `throws` de la cabecera de un método es obligatorio declarar el **resto de excepciones** (como **`IOException`**) para que se propaguen al método que lo ha invocado.

Los siguientes ejemplos muestran la **propagación de excepciones en Java** añadiendo una cláusula `throws` a la cabecera del método:

```
public static
void insertarLinea(String rutaFicheroTexto, String linea)
throws IOException
```

```
public static
String[] consultarLineas(String rutaFicheroTexto)
throws FileNotFoundException, IOException
```

En resumen, al codificar un método, se deberá decidir si las excepciones se capturarán en el propio método con una estructura `try-catch-finally` o se propagarán al método que lo ha llamado con una cláusula `throws` añadida a la cabecera del método:

MÉTODO QUE PROPAGA UNA EXCEPCIÓN	`public void procesar()`
	`throws IOException {`
	`  // fragmento de código que puede lanzar`
	`  // una excepción de tipo IOException`
	`}`

MÉTODO EQUIVALENTE QUE CAPTURA UNA EXCEPCIÓN	`public void procesar() {`
	`  // fragmento de código libre de excepciones y`
	`  // que declara los recursos necesarios`
	`  try {`
	`  // fragmento de código que utiliza`
	`  // los recursos declarados y que puede lanzar`
	`  // una excepción de tipo IOException`
	`  }`

```
    catch (IOException ioe) {
        // tratamiento de la excepción
    }
    finally {
        // liberación de los recursos utilizados
    }
}
```

2.3. LANZAMIENTO DE EXCEPCIONES

El **lanzamiento de excepciones** consiste en crear o generar de forma explícita una excepción concreta para una situación específica. En Java se realiza con la instrucción **`throw`**, que lanza un objeto de la clase `Throwable`.

Los siguientes ejemplos muestran el **lanzamiento de excepciones en Java** con la instrucción `throw`:

```
throw new Exception("Mensaje de error genérico.");
throw new RuntimeException("Mensaje de error de tiempo de ejecución");
throw new IOException("Mensaje de error de entrada/salida.");
```

Cuando se lanza una excepción:

- Se sale inmediatamente del bloque del código actual.
- Si el bloque tiene asociada una clausula `catch` adecuada para el tipo de la excepción generada, se ejecuta el cuerpo de la cláusula `catch` y la excepción queda tratada.
- Si no, se sale inmediatamente del bloque (o método) dentro del cual está el bloque en el que se produjo la excepción y se busca una cláusula `catch` apropiada.
- El proceso continúa hasta llegar al método `main` del programa. Si ahí tampoco existe una cláusula `catch` adecuada, la Máquina Virtual Java finaliza su ejecución con un mensaje de error.

2.4. CREACIÓN Y USO DE EXCEPCIONES PERSONALIZADAS

La **creación de excepciones personalizadas** consiste en definir mediante herencia una nueva excepción como subclase de otra excepción ya existente.

Los siguientes ejemplos muestran la **creación de excepciones personalizadas en Java** a partir de excepciones predefinidas:

```
public class ExcepcionPersonalizada
extends Exception {
    public ExcepcionPersonalizada(String mensaje) {
        super(mensaje);
    }
}
```

```
public class ExcepcionTiempoEjecucionPersonalizada
extends RuntimeException {
    public ExcepcionTiempoEjecucionPersonalizada(String mensaje) {
        super(mensaje);
    }
}
```

```
public class ExcepcionEntradaSalidaPersonalizada
extends IOException {
    public ExcepcionEntradaSalidaPersonalizada(String mensaje) {
        super(mensaje);
    }
}
```

Un programa Java puede definir sus propias excepciones personalizadas como subclases de las clases **`Exception`**, **`RuntimeException`**, **`IOException`** u otras más específicas, según el contexto en el que esté enfocado o las situaciones particulares que gestione.

El tratamiento de las excepciones personalizadas es el mismo que el de las excepciones predefinidas en Java: se pueden capturar, se pueden propagar y se pueden lanzar.

El **uso de excepciones personalizadas** en un programa en ejecución consta de varios pasos:

- Cuando un método del programa detecte una condición de error específica, lanzará con la instrucción `throw` una excepción personalizada con un mensaje de error concreto. Esta excepción personalizada se propagará al método que lo ha llamado.
- Entonces, el método que recibe la excepción personalizada tendrá dos opciones para su tratamiento: capturarla con un bloque `try-catch` o propagarla a otro método superior.
- En última instancia, será el programa principal `main` el encargado de tratar esta excepción personalizada capturándola. Si ahí no existe una cláusula `catch` adecuada, la Máquina Virtual Java finalizará la ejecución del programa con un mensaje de error.

3. CASOS PRÁCTICOS

3.1. CAPTURA DE EXCEPCIONES

En los siguientes ejemplos, se muestra la forma de tratar excepciones, desde el manejo de una sola posible excepción hasta el uso de varios bloques `catch` como parte del programa.

3.1.1. Captura de una excepción

El siguiente ejemplo muestra la captura de una excepción que se produce al intentar realizar una división por cero. Se crea una variable para el dividendo y otra para el divisor con valores asignados en su declaración. Al tratar de hacer la división se producirá una excepción del tipo **`AritmeticException`**, por lo que se detiene la ejecución del `try`, el fujo del programa pasará al `catch` y la sentencia que mostraría el resultado nunca se ejecutará.

```
public class Ejemplo1 {

    public static void main(String[] args) {

        try {
            int dividendo = 10;
            int divisor = 0;

            int cociente = dividendo / divisor;

            System.out.println("cociente = " + cociente);
        }
        catch (ArithmeticException ae) {
            System.out.println("Excepción Aritmética: "
                    + ae.getMessage());
            ae.printStackTrace();
        }
    }

}
```

3.1.2. Captura de varias excepciones

En el siguiente ejemplo se pretende hacer una división pidiendo al usuario los datos del dividendo y del divisor. Al recoger los datos introducidos por teclado, se recuperan en forma de cadena de caracteres y deberá hacerse posteriormente una conversión a entero. Esta operación puede generar una excepción del tipo **`NumberFormatException`** si el usuario no ha escrito un dato en formato de número entero.

Además, al igual que en el ejemplo anterior, al tratar de hacer la división, puede producirse una excepción del tipo **`AritmeticException`** si el divisor es cero.

Dado que este código puede provocar excepciones de más de un tipo, se incluirá un bloque `catch` para cada una de ellas. En caso de que se produzca alguna excepción, el flujo del programa pasará al bloque `catch` correspondiente al tipo de la excepción.

El orden en el que se ponen los bloques `catch` en este caso no es relevante puesto que no existe una relación entre ellas, pero al tratar de capturar varias excepciones cuyas clases estén en una misma jerarquía, deberá hacerse siempre de la más específica a la más general.

```
import java.util.Scanner;

public class Ejemplo2 {

    public static void main(String[] args) {

        Scanner teclado = new Scanner(System.in);
        int dividendo, divisor, cociente;
        String cadDividendo, cadDivisor;

        try {
            System.out.print("dividendo? ");
            cadDividendo = teclado.nextLine();
            dividendo = Integer.parseInt(cadDividendo);

            System.out.print("divisor? ");
            cadDivisor = teclado.nextLine();
            divisor = Integer.parseInt(cadDivisor);

            cociente = dividendo / divisor;
            System.out.println("cociente = " + cociente);
        }
        catch (NumberFormatException nfe) {
            System.out.println("Caracteres no numéricos.");
        }
        catch (ArithmeticException ae) {
            System.out.println("Error de división por cero.");
        }
    }

}
```

3.1.3. Uso del bloque catch como parte del programa

En el siguiente ejemplo se pretende hacer una división pidiendo al usuario los datos del dividendo y del divisor. La recogida de datos se realiza mediante un objeto de la clase **`BufferedReader`**, que necesita un objeto de la clase **`InputStreamReader`** conectado a la entrada estándar. Esta operación requiere que se capture una posible excepción del tipo **`IOException`**.

Además, al igual que en el ejemplo anterior, y dado que los datos se recuperan en forma de cadena de caracteres, deberá hacerse una conversión a entero. Esta operación puede generar una excepción del tipo **`NumberFormatException`** si el usuario no ha escrito un dato entero.

También podría producirse una excepción del tipo **`AritmeticException`** si el divisor es cero.

En consecuencia, el código contempla tres bloques `catch` para los tipos de excepciones que puedan producirse.

Asimismo, si se produce una excepción se vuelven a pedir los datos hasta que no haya error con la limitación de cinco errores de formato. Esto se codifica con un bucle `while` con dos variables de control y conteniendo una estructura `try-catch`. Para ello, dentro de cada bloque `catch` se asocia el valor false a una variable booleana que marca el error y dentro del bloque `catch` para el error de formato se gestiona el contador de intentos.

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Ejemplo3 {

    public static void main(String[] args) {

        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader teclado = new BufferedReader(isr);

        String cadDividendo, cadDivisor;
        int dividendo, divisor, cociente;

        boolean error = true;
        int numIntentos = 0;

        while (error && numIntentos <= 5) {
            try {
                System.out.print("Dividendo? ");
                cadDividendo = teclado.readLine();
                dividendo = Integer.parseInt(cadDividendo);
                System.out.print("Divisor? ");
                cadDivisor = teclado.readLine();
                divisor = Integer.parseInt(cadDivisor);
                cociente = dividendo / divisor;
                System.out.println("Cociente = " + cociente);
                error = false;
            }
            // Error de entrada/salida
            catch (IOException ioe) {
                System.out.println("Error de entrada/salida.");
                error = true;
            }
            // Error de formato de número (contiene caracteres no numéricos)
            catch (NumberFormatException nfe) {
                System.out.println("Error de formato de número (hay letras).");
                numIntentos++;
                if (numIntentos > 5) {
                    System.out.println("Alcanzado el número máximo de intentos (5).");
                    System.out.println("Se cierra el programa por insistencia del usuario.");
                }
                error = true;
            }
            // Error aritmético (división por cero)
            catch (ArithmeticException ae) {
                System.out.println("Error aritmético (división por cero).");
                error = true;
            }
        }
    }

}
```

3.2. PROPAGACIÓN DE EXCEPCIONES

El siguiente ejemplo muestra un programa principal que solicita un dato numérico al usuario. En el propio `main` es donde se realiza el control de excepciones, que pueden producirse al hacer la conversión de cadena a entero o en la propia solicitud del dato. El código que provoca las excepciones no se encuentra en el bloque que las trata, sino que se realiza una propagación de las mismas desde el bloque donde se producen hasta el programa principal.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EjemploPropagar {

    public static String solicitarCadena() throws IOException {
        String cadena = "";
        InputStreamReader isr =
                new InputStreamReader(System.in);
        BufferedReader teclado = new BufferedReader(isr);
        cadena = teclado.readLine();
        return cadena;
    }

    public static int solicitarEntero() throws IOException {
        int entero = 0;
        String cadena = solicitarCadena();
        entero = Integer.parseInt(cadena);
        return entero;
    }

    public static void main(String[] args) {
        try {
            System.out.print("Número? ");
            int numero = solicitarEntero();
            System.out.println("Número = " + numero);
        }
        catch (NumberFormatException nfe) {
            System.out.println("Se han introducido letras en lugar de números.");
        }
        catch (IOException ioe) {
            System.out.println("Error de entrada/salida.");
            System.out.println("El programa se cierra debido al error.");
        }
    }

}
```

Por un lado, las excepciones que no es obligatorio capturar, como **`NumberFormatException`**, si no se tratan en el método `solicitarEntero`, que es donde se produce, se propagarán de forma automática al método que lo invocó, que es el programa principal `main`. Como se observa en el ejemplo, esta excepción sí que es tratada en el bloque `main`, por lo que, en caso de producirse, se ejecutará el `catch` correspondiente.

Por otro lado, si se quiere que una excepción de las que es obligatorio tratar se propague desde un método a otro, como es el caso de **`IOException`**, será necesario añadir `throws IOException` en la cabecera del método donde se puede producir la excepción. Como se observa en el ejemplo, en el método `solicitarCadena` se puede producir una de estas excepciones al tratar de recuperar información del teclado, para no tener que capturar la excepción en este bloque se añade a la firma `throws IOException`. De esta manera, si se produce esta excepción, saltará al método se lo invocó

que es `solicitarEntero`. En este método ocurre lo mismo, se fuerza la propagación de la excepción al método llamador, que es el programa principal `main`, donde se trata la excepción.

3.3. LANZAMIENTO DE EXCEPCIONES

El siguiente ejemplo muestra un programa principal donde se solicitan dos datos al usuario y se realiza una división entera. El control de excepciones se realiza en el `main` y pueden producirse al hacer la conversión de cadena a entero o al hacer una división por cero.

```
import java.util.Scanner;

public class EjemploLanzar {

    public static int hacerDivisionEntera(int dividendo, int divisor) {
        if(divisor == 0) {
            throw new ArithmeticException("Error de división por cero.");
        }
        else {
            return dividendo / divisor;
        }
    }

    public static int convertirAEntero(String cadena) {
        int numeroEntero;
        try {
            numeroEntero = Integer.parseInt(cadena);
            return numeroEntero;
        }
        catch (NumberFormatException nfe) {
            throw nfe;
        }
    }

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int dividendo, divisor, cociente;
        String cadDividendo, cadDivisor;

        try {
            System.out.print("dividendo? ");
            cadDividendo = teclado.nextLine();
            dividendo = convertirAEntero(cadDividendo);

            System.out.print("divisor? ");
            cadDivisor = teclado.nextLine();
            divisor = convertirAEntero(cadDivisor);

            cociente = hacerDivisionEntera(dividendo, divisor);

            System.out.println("cociente = " + cociente);
        }
        catch (NumberFormatException nfe) {
            System.out.println(nfe.toString());
        }
        catch (ArithmeticException ae) {
            System.out.println(ae.getMessage());
        }
    }

}
```

En el método `hacerDivisionEntera`, se comprueba si el divisor es cero y en caso afirmativo, se lanza una excepción del tipo **`ArithmeticExcepcion`** indicando un mensaje personalizado en el constructor. En este caso no se deja que se produzca una excepción, sino que se fuerza que se produzca.

En el método `convertirAEntero`, se trata una posible excepción **`NumberFormatException`** y en caso de producirse, se lanza de nuevo para que sea tratada por el bloque que invoca el método.

3.4. CREACIÓN Y USO DE EXCEPCIONES PERSONALIZADAS

El siguiente ejemplo muestra la creación de una excepción personalizada y su uso.

La clase **`ExcepcionDivisionPorCero`** deriva de la clase **`ArithmeticException`** y en su constructor recibe un mensaje que contendrá información relativa a la excepción y que podrá ser utilizada para mostrarla al usuario.

```
public class ExcepcionDivisionPorCero extends ArithmeticException{

    public ExcepcionDivisionPorCero(String mensaje) {
        super(mensaje);
    }

}
```

La siguiente clase, en su `main`, solicita al usuario valores para dos datos numéricos y los utiliza para hacer una llamada al método `hacerDivisionEntera`. Este método comprobará que se puede hacer la división, es decir, que su divisor no es cero. En caso de poder hacer la operación, devolverá el resultado de la misma y en caso contrario, lanzará una excepción personalizada del tipo creado anteriormente.

En el programa principal `main` se tratará esta posible excepción, mostrando un mensaje en caso de producirse.

```java
import java.util.Scanner;

public class EjemploCrear {

    public static int hacerDivisionEntera(int dividendo, int divisor) {
        if(divisor == 0) {
            throw new ExcepcionDivisionPorCero("Error de división por cero.");
        }
        else {
            return dividendo / divisor;
        }
    }

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int dividendo, divisor, cociente;

        try {
            System.out.print("dividendo? ");
            dividendo =  teclado.nextInt();

            System.out.print("divisor? ");
            divisor = teclado.nextInt();

            cociente = hacerDivisionEntera(dividendo, divisor);

            System.out.println("cociente = " + cociente);
        }
        catch (ExcepcionDivisionPorCero ae) {
            System.out.println(ae.getMessage());
        }
        catch (Exception e) {
            System.out.println(e.toString());
        }
    }

}
```