DAM/DAW

PROGRAMACIÓN

07

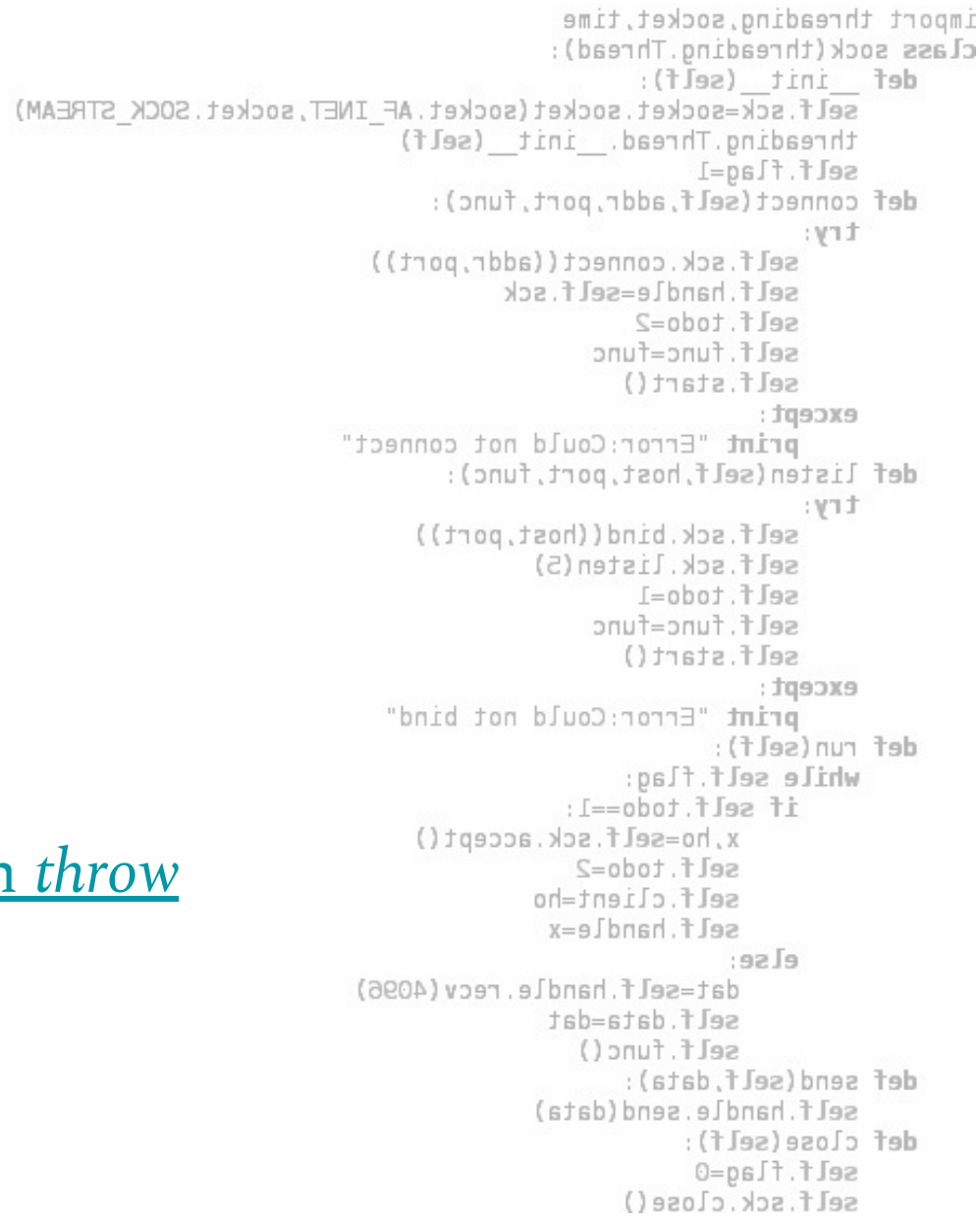
Excepciones en Java

Rev: 3.2

fran@iessanclemente.net

Indice

- Introducción
- *java.lang.Exception*
- Gestión de excepciones
- El bloque *try/catch*
- *catch* múltiple
- Bloques *try* anidados
- La cláusula *finally*
- Lanzamiento de excepciones con *throw*
- La cláusula *throws*
- Excepciones personalizadas



Introducción (I)

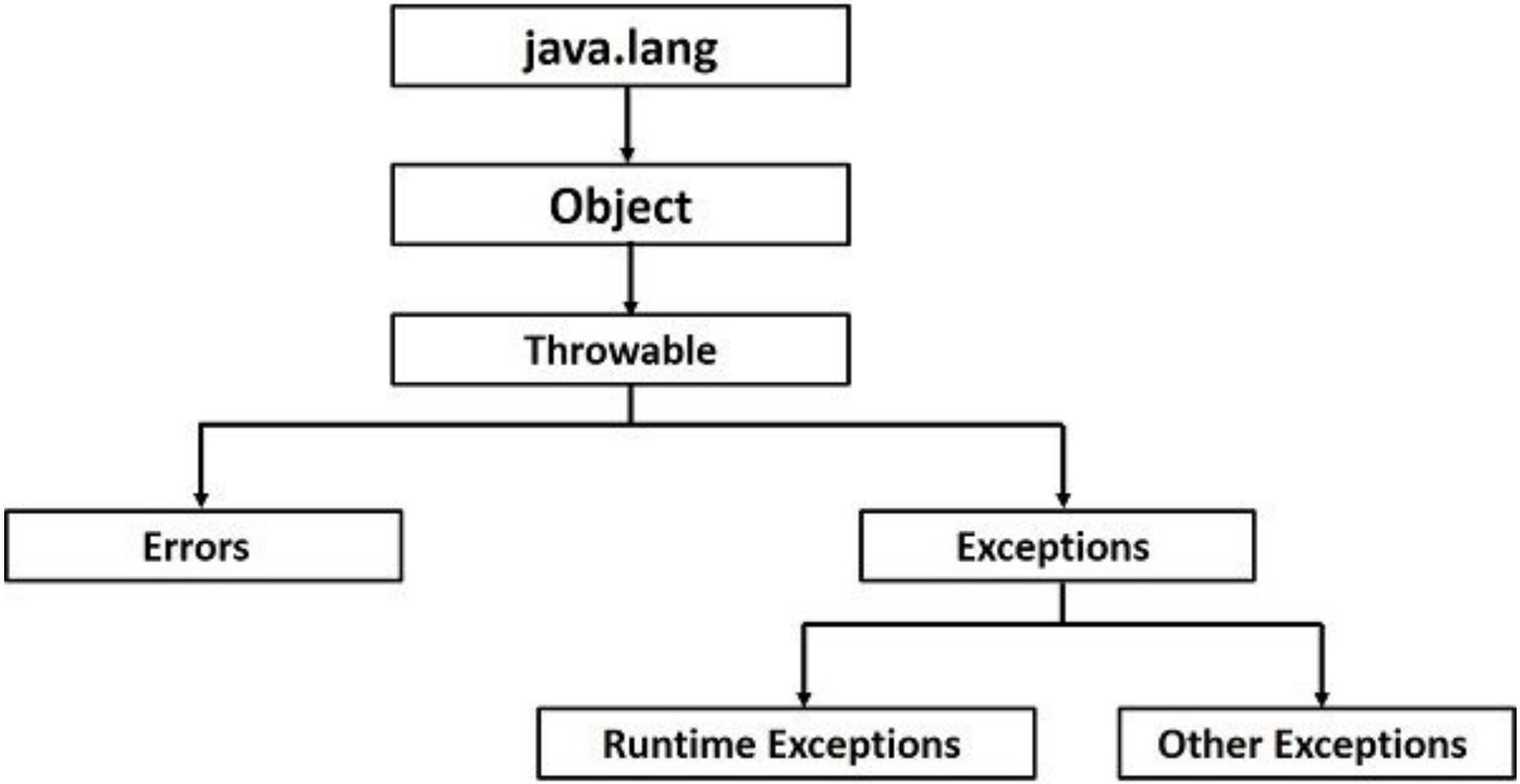
- Una excepción es un error que se produce en tiempo de ejecución
- Mediante el subsistema de manejo de excepciones de la JVM podremos, de una manera estructurada y controlada, gestionar estas situaciones que, de otro modo, producirán la finalización abrupta de nuestra aplicación
- Los lenguajes como Java, que proporciona un sistema de gestión de excepciones, nos permitirán definir un bloque de código (*exception handler*) que se ejecutará automáticamente en el momento en que se produzca uno de estos errores.
- De este modo, no es necesario el estar continuamente chequeando en nuestro código si se produce una determinada situación anómala o si la llamada a un método genera una condición de error. En el momento en que se produzca dicho error, se ejecutará el código que hayamos de definido para el manejo de la excepción correspondiente

Introducción (II)

- Java define una serie de excepciones para situaciones de error comunes: *división por cero*, *índice del array fuera de rango*, *fichero no encontrado*,...
- En Java, todas las excepciones son clases y derivan de la clase *Throwable*. Cuando se produce una excepción, se genera un objeto de alguna de ellas
- *Throwable* tiene dos subclases: *Error* y *Exception* (ver gráfico)
  - Las excepciones del tipo *Error* se corresponden con errores serios de la JVM y que nuestra aplicación no capturará (*OutOfMemoryError*,...)
  - Las excepciones del tipo *Exception* se corresponden con errores debidos a la propia actividad del programa (*ArithmeticException*, *NullPointerException*,...) y serán, habitualmente, capturadas
- Además de las excepciones pertenecientes a la jerarquía de clases definida en el API de Java, nosotros podremos crear nuestras propias excepciones extendiendo la subclase *Exception*

Introducción (y III)

- Jerarquía de *Throwable*



java.lang.Exception (I)

- *java.lang.Exception* es la *superclase* de una jerarquía de decenas de *subclases* para el tratamiento de todo tipo de excepciones. Algunas ejemplos...

Excepción	Descripción
`ArithmeticException`	Error en una operación aritmética
`ArrayIndexOutOfBoundsException`	Acceso a un *array* mediante un valor de índice fuera de rango (es una subclase de *IndexOutOfBoundsException*)
`StringIndexOutOfBoundsException`	Acceso a un *string* mediante un valor de índice fuera de rango es una subclase de *IndexOutOfBoundsException*)
`NullPointerException`	Se genera al usar lo que debería ser una referencia a un objeto pero tiene valor *null*
`NumberFormatException`	Error al tratar de convertir una cadena no válida en un tipo numérico
`ClassCastException`	Intento erróneo de *casting* de un objeto a un tipo del que no es instancia
`SecurityException`	Lanzada por el *Security Manager* para indicar una violación de seguridad

java.lang.Exception (y II)

- *java.lang.Exception* hereda una serie de métodos de *Throwable* y que, a su vez, estarán disponibles en sus *subclases*. Entre los más interesantes:

Método	Descripción
`String getMessage()`	Devuelve una descripción de la excepción
`String toString()`	Devuelve una cadena conteniendo el tipo de la excepción y una descripción de la misma
`void printStackTrace()`	Imprime la pila de llamadas que generó la excepción. Este método está sobrecargado para soportar la impresión sobre diferentes flujos de salida
`StackTraceElement[] getStackTrace()`	Devuelve un array donde se recogen todas las diferentes entradas de la pila de llamadas al método que generó la excepción

Gestión de excepciones

- Java facilita la gestión de excepciones mediante el empleo de las siguientes palabras reservadas: *try*, *catch*, *throw*, *throws* y *finally*
- El proceso general de captura y manejo de una excepción sería:
  - Aquellas sentencias de nuestro programa susceptibles de generar una excepción estarán contenidas dentro de un bloque *try*
  - Si la excepción monitorizada se produce, se generará una instancia de la clase correspondiente a dicha excepción y será "lanzada" (*thrown*)
  - Nuestro programa podrá definir un bloque *catch* para "capturar" la excepción y actuar en consecuencia
  - *throw* nos permitirá lanzar manualmente una nueva excepción
  - *throws* informa de una excepción no capturada lanzada por un método
  - *finally* se emplea para definir código a ejecutar al salir del bloque *try*, con independencia de que se haya generado la excepción monitorizada

El bloque *try/catch* (I)

- El elemento base de la gestión de excepciones es el bloque *try/catch*
- Mediante esta construcción de lenguaje podremos:
  - Definir la parte del código que debe ser monitorizada
  - Definir las acciones a realizar en el caso de que se produzce el error
- Su sintaxis (sin la cláusula *finally*) es:

```
try {
  // bloque de código que puede generar error y se va a a monitorizar
}
catch(TipoExcepcion1 exObj) {
  // Código para manejar la excepción de tipo TipoExcepcion1
}
catch(TipoExcepcion2 exObj) {
  // Código para manejar la excepción de tipo TipoExcepcion2
}
...
```

El bloque *try/catch* (II)

- Cuando se produce una excepción por la ejecución del código del bloque *try*, ésta será "capturada" por el bloque *catch* que se corresponda con el tipo de la excepción generada y su código ejecutado
- Asociados a un bloque *try* podremos definir todos lo bloques *catch* que deseemos. Hasta JDK7, cada bloque *catch* podía capturar una única excepción. Desde JDK7, podemos añadir múltiples excepciones en el mismo bloque *catch* separándolas con el símbolo *barra vertical* (|)

```
catch(TipoExcepcion1 | TipoExcepcion2  exObj) {
  // Código para manejar excepciones de tipo TipoExcepcion1 ó TipoExcepción2
}
```

- Si no se genera ninguna excepción, no se ejecutará ningún bloque *catch*
- Si se genera pero no existe un *catch* para capturarla (en el propio método o en el que lo haya invocado), se captura por la JVM y finaliza el programa

El bloque try/catch (y III)

- Ejemplo (*java.util.InputMismatchException*):

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class DemoEx1 {
  public static void main(String[] args) {
    Scanner cin = new Scanner(System.in);
    int num;
    try {
      num = cin.nextInt();
    }
    catch(InputMismatchException e) {
      System.out.println("Entrada no válida!");
    }
    // Continúa la ejecución normal del programa...
  }
}
```

Importamos la clase (tipo) de la excepción que deseamos capturar. No es necesario importar las excepciones contenidas en el paquete java.lang

Encerramos en un bloque *try* el código que puede generar la excepción que, de producirse, deseamos capturar

Bloque *catch* que se ejecutará en caso de que se genere una excepción *InputMismatchException*

Tras la ejecución del bloque *try/catch* continúa la ejecución normal del programa

catch múltiple (I)

- A la hora de encadenar varios bloques *catch* asociados al mismo bloque *try*, debemos tener en cuenta lo siguiente:
  - Cada bloque *catch* debe capturar un tipo de excepción diferente
  - Cuando se produce una excepción, se evalúan las diferentes sentencias *catch* del bloque *try* siguiendo el mismo orden en que han sido introducidas. Se ejecutará el primer bloque *catch* que capture la excepción generada. Todos los demás serán ignorados. Por tanto, sólo se ejecutará un único bloque *catch*.
  - Un *catch* de una excepción particular, capturará también cualquier excepción de una de sus *subclases*. Así, un *catch* de *Throwable* capturaría cualquier error o excepción
  - De lo anterior se desprende que, en caso de añadir varios bloques *catch* a un bloque *try*, deben introducirse de más específico a menos

catch múltiple (II)

- Ejemplo:

```
public class DemoEx2 {
  public static void main(String[] args) {
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.length; i++)
      try {
        System.out.println(numer[i] + "/" + denom[i] + "=" + (numer[i]/denom[i]));
      }
      catch(AritmeticException e) { System.out.println(e); }
      catch(IndexOutOfBoundsException e) { System.out.println(e); }
  }
}
```

Captura la *subclase ArrayIndexOutOfBoundsException*

Dado que ambos bloques *catch* tratan de igual modo sus excepciones, podríamos haberlas agrupado en un único bloque (*multicatch*):

```
catch(AritmeticException|IndexOutOfBoundsException e){ System.out.println(e); }
```

catch múltiple (y III)

- Más ejemplos:

```
...
for(int i=0; i<numer.length; i++)
  try {
    System.out.println(numer[i] + "/" + denom[i] + "=" + (numer[i]/denom[i]));
  }
  catch(Exception e) { System.out.println(e); }
```

Captura todas las excepciones

```
...
for(int i=0; i<numer.length; i++)
  try {
    System.out.println(numer[i] + "/" + denom[i] + "=" + (numer[i]/denom[i]));
  }
  catch(AritmeticException e) { System.out.println(e); }
  catch(Exception e) { System.out.println("Error inesperado!!! -> " + e); }
```

Añadido al final para captura cualquier excepción inesperada

Bloques *try* anidados (I)

- Podemos anidar bloques *try* dentro de otros bloques *try*
- Cuando se anidan estos bloques, una excepción generada dentro de un bloque *try* interno no capturada por ningún *catch* asociado a ese *try*, se propagará hacia arriba en la jerarquía de anidamiento y podrá ser capturada por un *catch* del bloque *try* inmediatamente superior
- Con frecuencia, la razón para anidar bloques *try* es la de permitir un tratamiento diferenciado de los errores según su categoría. Algunos serán graves y no podrán ser solucionados. Otros, sin embargo, serán leves y podrán ser manejados de forma inmediata. De este modo, podríamos definir bloques *try* externos para capturar aquellos errores más severos, dejando para los bloques *try* internos la gestión de los errores menos graves
- Veámoslo sobre el ejemplo anterior...

Bloques *try* anidados (y II)

- Ejemplo:

```
public class DemoEx3 {
  public static void main(String[] args) {
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    try { // try externo
      for(int i=0; i<numer.length; i++) {
        try {  // try interno
          System.out.println(numer[i]+"/"+denom[i]+"="+(numer[i]/denom[i]));
        }
        catch(AritmeticException e) { System.out.println(e); }
      }
    }
    catch(IndexOutOfBoundsException e) { System.out.println(e); }
  }
}
```

bloques *try* anidados

La cláusula *finally*

- La cáusula *finally* nos permite definir código que se ejecutará siempre tras la ejecución de un bloque *try/catch*, independientemente de que se haya generado o no una excepción y de que se capture o no
- Un ejemplo típico es el de un método en el que se ha producido algún tipo de error durante su ejecución y debe retornar pero que, antes de hacerlo, debe cerrar archivos o conexiones abiertas previamente
- La sintaxis del bloque *try/catch/finally* es:

```
try { // bloque de código que puede generar error
}
catch(TipoExcepcion1 exObj) { // excepción de tipo TipoExcepcion1
}
catch(TipoExcepcion2 exObj) { // excepción de tipo TipoExcepcion2
}
...
finally { // Código que se ejecutará al finalizar el bloque try/catch
}
```

Lanzamiento de excepciones con *throw* (I)

- Hasta ahora hemos capturado y procesado excepciones generadas internamente por la JVM
- En determinadas situaciones nos encontraremos ante la necesidad de que nuestra propia aplicación lance una excepción de forma manual. Por ej.:
  - relanzar una excepción capturada por un bloque *catch*
  - generar y lanzar una excepción del JDK o propia
- Para ello, disponemos de la sentencia *throw* cuya sintaxis es la siguiente:

```
throw objeto_Exception;
```

- Fíjate que la sentencia *throw* necesita una instancia (objeto) del tipo de excepción que nos interese provocar. Es decir, dicha sentencia no genera la excepción, simplemente la lanza (como con cualquier otra clase, el operador *new* nos permitirá generar objetos de las diferentes excepciones)

Lanzamiento de excepciones con *throw* (y II)

```
public class DemoEx4 {
  public static int dameNum() {
    java.util.Scanner cin = new java.util.Scanner(System.in);
    int num;
    try {
        num = Integer.parseInt(cin.nextLine().trim());
    }
    catch(NumberFormatException e) {
        System.out.println("Entrada no válida. Relanzando excepción...");
        throw e;
    }
    return num;
  }
  public static void main(String[] args) {
    System.out.println("Dame un número: ");
    try {
        System.out.println(dameNum());
    }
    catch(Exception e) {
        System.out.println("Excepción recibida --> " + e);
  }
}
```

En caso de producirse una entrada no numérica...

...se captura la excepción en el bloque *catch* y, a continuación, se relanza

Si la llamada al método *dameNum*() produjo una excepción, se captura

La cláusula *throws* (I)

- Cuando el código de un método puede generar una excepción pero no la gestiona el mismo, debe indicarlo para que cualquier otro código que invoque a dicho método se preocupe de capturar dicha excepción
- Para ello, Java proporciona la cláusula *throws* que se añadirá al final de la declaración del método, seguida por la lista de excepciones (separadas por comas) susceptibles de ser generadas por el método (y no capturadas)

```
[modif] tipo nombreMetodo(lista_params) throws lista_excepciones;
```

- Esta regla no se aplica si el tipo de excepción es *RuntimeException*, *Error* o cualquiera de sus subclases, pues no se espera que se produzcan como resultado del funcionamiento normal del programa
- La no declaración por el método de una excepción no capturada, provocará el consiguiente error en tiempo de compilación

La cláusula *throws* (y II)

❖ Ejemplo:

```
public class DemoEx5 {
  public static int dameNum() throws NumberFormatException {
    java.util.Scanner cin = new java.util.Scanner(System.in);

    return Integer.parseInt(cin.nextLine().trim());
  }

  public static void main(String[] args) {
    System.out.println("Dame un número: ");
    try {
        System.out.println(dameNum());
    }
    catch(Exception e) {
        System.out.println("Excepción recibida --> " + e);
  }
}
```

El método *dameNum()* puede generar una excepción, pero no la captura. Tenemos que declararla

Si la llamada al método *dameNum()* produjo una excepción, se captura

Excepciones personalizadas (I)

- El sistema de gestión de excepciones de Java no está limitado únicamente a las excepciones definidas por él, si no que nos da la posibilidad de crear nuestras propias excepciones “a medida”
- A través de la creación y uso de estas excepciones propias, podremos manejar errores que sean específicos de nuestra aplicación. Estas nuevas excepciones serán gestionadas empleando los mismos mecanismos proporcionados por Java para capturar, procesar y relanzar cualquiera de las excepciones predefinidas en el lenguaje.
- Si bien el empleo de las excepciones estándar de Java será más que suficiente para solventar cualquier incidencia en nuestra aplicación, las excepciones personalizadas nos proporcionan un mayor grado de flexibilidad. Al poder añadir atributos y métodos, estos podrían almacenar información adicional, códigos de error específicos,...

Excepciones personalizadas (II)

- Para crear una excepción propia sólo necesitamos crear una clase que extienda la clase *Exception* que, a su vez, hereda de *Throwable*. De este modo, heredaremos todos sus métodos (*getMessage( )*, *printStackTrace( )*, *toString( )*, ...) que podremos sobreescribir de considerarlo necesario.
- A la hora de gestionar nuestra propias excepciones, suele ser de ayuda establecer una descripción personalizada para las nuevas instancias de la misma. Esto, podemos hacerlo de diversas maneras:
  - La clase *Exception* dispone de un constructor que acepta un *String* para establecer la descripción del error. Esta descripción se devuelve en las llamadas a los métodos *toString( )* y *getMessage( )*
  - Alternativamente, podemos sobreescribir el método *toString( )*. Por defecto, este método devuelve la descripción del error precedida por el nombre de la excepción (clase).

Excepciones personalizadas (III)

- Veamos un ejemplo. Para ello, crearemos la siguiente excepción:

```
public class DatoNoValido extends Exception {
  int errCode;
  String valor;
  public DatoNoValido(String valor, int errCode, String msg) {
    super(msg);
    this.valor = valor;
    this.errCode = errCode;
  }
  public int getErrorCode() {
    return this.errCode;
  }
  public String toString() {
    return "[ERR: " + this.errCode + "] " +
           this.getMessage() + " (value: " + this.valor + ")";
  }
}
```

Extendemos la clase *Exception*

Constructor y método personalizado (son opcionales)

Sobreescribimos el método heredado *toString()*

Excepciones personalizadas (y IV)

- Creamos ahora el código para probar la nueva excepción...

```
public class DemoEx6 {
  public static void validaEdad(int edad) throws DatoNoValido {
    if(edad < 18)
      throw new DatoNoValido(Integer.toString(edad), 101, "Menor de edad");
  }
  public static void main(String[] args) {
    try {
        validaEdad(15);
    }
    catch(Exception e) {
        System.out.println("Excepción recibida --> " + e);
    }
  }
}
```

Lanzamos nuestra excepción

```
Excepción recibida --> [ERR: 101] Menor de edad (value: 15)
```