8a. Excepciones.

8b. Documentación del código.

8a. Excepciones.

1. Excepciones.

   1.1. Capturar una excepción.

   1.2. El manejo de excepciones.

   1.3. Delegación de excepciones con throws.

1. Excepciones.

A lo largo de nuestro aprendizaje de Java nos hemos topado en alguna ocasión con **errores**, pero éstos suelen ser los que nos ha indicado el compilador. Un punto y coma por aquí, un nombre de variable incorrecto por allá, pueden hacer que nuestro compilador nos avise de estos descuidos. Cuando los vemos, se corrigen y obtenemos nuestra clase compilada correctamente.

Pero, ¿Sólo existen este tipo de Errores? ¿Podrían existir Errores no sintácticos en nuestros programas? Está claro que sí, un programa perfectamente compilado en el que no existen Errores de sintaxis, puede generar otros tipos de Errores que quizá aparezcan en tiempo de ejecución. A estos Errores se les conoce como **excepciones.**

Aprenderemos a gestionar de manera adecuada estas excepciones y tendremos la oportunidad de utilizar el potente sistema de manejo de Errores que Java incorpora. La potencia de este sistema de manejo de Errores radica en:

1. Que el código que se encarga de manejar los Errores, es perfectamente identificable en los programas. Este código puede estar separado del código que maneja la aplicación.

2. Que Java tiene una gran cantidad de Errores estándar asociados a multitud de fallos comunes, como por ejemplo divisiones por cero, fallos de entrada de datos, etc. Al tener tantas excepciones localizadas, podemos gestionar de manera específica cada uno de los Errores que se produzcan.

En Java se pueden preparar los fragmentos de código que pueden provocar Errores de ejecución para que, si se produce una excepción, el flujo del programa es lanzado (`throw`) hacia ciertas zonas o rutinas que han sido

creadas previamente por el programador y cuya finalidad será el tratamiento efectivo de dichas excepciones. Si no se captura la excepción, el programa se detendrá con toda probabilidad.

En Java, las excepciones están representadas por clases. El paquete **`java.lang.Exception`** y sus subpaquetes contienen todos los tipos de excepciones. Todas las excepciones derivarán de la clase **`Throwable`**, existiendo clases más específicas. Por debajo de la clase **`Throwable`** existen las clases **`Error`** y **`Exception`**. Errores una clase que se encargará de los Errores que se produzcan en la máquina virtual, no en nuestros programas. Y la clase Exception será la que a nosotros nos interese conocer, pues gestiona los Errores provocados en los programas.

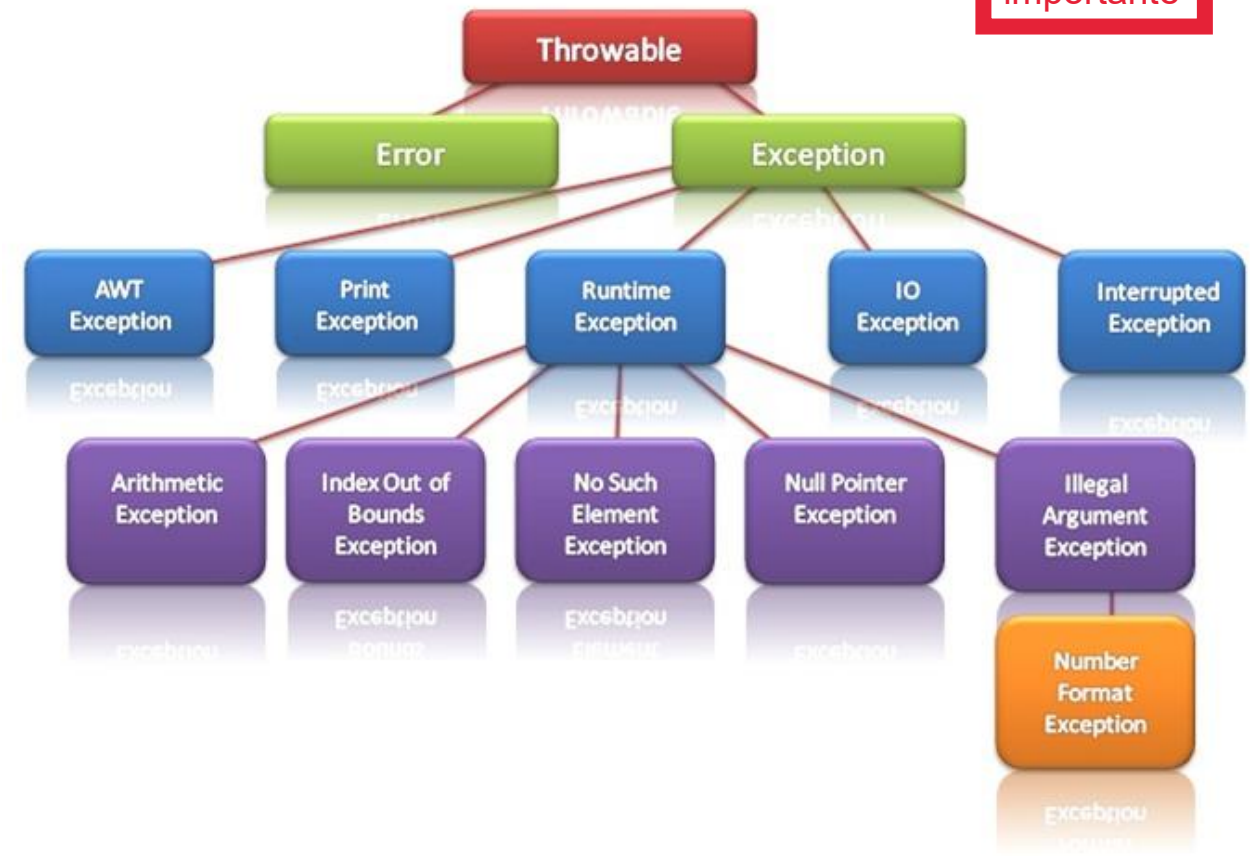
Java lanzará una excepción en respuesta a una situación poco usual. Cuando se produce un Error se genera un objeto asociado a esa excepción. Este objeto es de la clase **`Exception`** o de alguna de sus herederas. Este objeto se pasa al código que se ha definido para manejar la excepción. Dicho código puede manipular las propiedades del objeto **`Exception`**.

El programador también puede lanzar sus propias excepciones. Las excepciones en Java serán objetos de clases derivadas de la clase base **`Exception`**. Existe toda una jerarquía de clases derivada de la clase base **`Exception`**. Estas clases derivadas se ubican en dos grupos principales:

- Las excepciones en tiempo de ejecución, que ocurren cuando el programador no ha tenido cuidado al escribir su código.
- Las excepciones que indican que ha sucedido algo inesperado o fuera de control.

En la siguiente imagen te ofrecemos una aproximación a la jerarquía de las excepciones en Java.

importante



1.1. Capturar una excepción.

Para poder capturar excepciones, emplearemos la estructura de captura de excepciones `try-catch-finally.`

Básicamente, para capturar una excepción lo que haremos será declarar bloques de código donde es posible que ocurra una excepción. Esto lo haremos mediante un bloque try (intentar). Si ocurre una excepción dentro de estos bloques, se lanza una excepción. Estas excepciones lanzadas se pueden capturar por medio de bloques catch. Será dentro de este tipo de bloques donde se hará el manejo de las excepciones.

Su sintaxis es:

```
try {
código que puede generar excepciones;
} catch (Tipo_excepcion_1 objeto_excepcion) {
Manejo de excepción de Tipo_excepcion_1;
} catch (Tipo_excepcion_2 objeto_excepcion) {
Manejo de excepción de Tipo_excepcion_2;
}
...
```

```
finally {
instrucciones que se ejecutan siempre
}
```

opcional

En esta estructura, la parte catch puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte `finally` es opcional y, si aparece, solo podrá hacerlo una sola vez.

Cada catch maneja un tipo de excepción. Cuando se produce una excepción, se busca el catch que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase `Exception` es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo `AritmethicException` y el primer catch captura el tipo genérico `Exception`, será ese catch el que se ejecute y no los demás.

Por eso el último catch debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo catch que capture objetos `Exception`.

Ejercicio resuelto

Realiza un programa en Java en el que se solicite al usuario la introducción de un número por teclado comprendido entre el 0 y el 100. Utilizando manejo de excepciones, debes controlar la entrada de dicho número y volver a solicitarlo en caso de que ésta sea incorrecta.

Solución:

```
/*
 * Ejercicio resuelto sobre manejo de excepciones.
 * El programa solicita que el usuario introduzca por teclado
 * un número entre 0 y 100, debiendo gestionarse la entrada
 * por medio de excepciones.
 */

import java.io.*;

public class ejercicio_resuelto_excepciones {
    public static void main(String[] args){
        int numero=-1;
```

```
        int intentos=0;
        String linea;

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

        do{
            try{
                System.out.print("Introduzca un número entre 0 y
100: ");
                linea = teclado.readLine();
                numero = Integer.parseInt(linea);

            }catch(IOException e){
                System.out.println("Error al leer del teclado.");

            }catch(NumberFormatException e){
                System.out.println("Debe introducir un número
entre 0 y 100.");

            }finally{
                intentos++;
            }
        }while (numero < 0 || numero > 100);

        System.out.println("El número introducido es: " + numero);
        System.out.println("Número de intentos: " + intentos);

    }
}
```

Se han utilizado estructuras try-catch-finally. En este programa se solicita repetidamente un número utilizando una estructura do-while, mientras el

número introducido sea menor que 0 y mayor que 100. Como al solicitar el número pueden producirse los errores siguientes:

- De entrada de información a través de la excepción `IOException` generada por el método `readLine()` de la clase `BufferedReader`.
- De conversión de tipos a través de la excepción `NumberFormatException` generada por el método `parseInt()`.

Entonces se hace necesaria la utilización de bloques catch que gestionen cada una de las excepciones que puedan producirse. Cuando se produce una excepción, se compara si coincide con la excepción del primer catch. Si no coincide, se compara con la del segundo catch y así sucesivamente. Si se encuentra un catch que coincide con la excepción a gestionar, se ejecutará el bloque de sentencias asociado a éste.

Si ningún bloque catch coincide con la excepción lanzada, dicha excepción se lanzará fuera de la estructura try-catch-finally.

El bloque finally, se ejecutará tanto si try terminó correctamente, como si se capturó una excepción en algún bloque catch. Por tanto, si existe bloque finally éste se ejecutará siempre.

1.2. El manejo de excepciones.

Como hemos comentado, siempre debemos controlar las excepciones que se puedan producir o de lo contrario nuestro software quedará expuesto a fallos. Las excepciones pueden tratarse de dos formas:

- **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que originó el error, se anula.

- **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

Java emplea la primera forma, pero puede simularse la segunda mediante la utilización de un bloque `try` en el interior de un `while`, que se repetirá hasta que el error deje de existir. En la siguiente imagen tienes un ejemplo de cómo llevar a cabo esta simulación.

```
7   public static void main(String[] args){
8     boolean fueradelimites=true;
9     int i; //Entero que tomará valores aleatorios de 0 a 9
10    String texto[] = {"uno","dos","tre","cuatro","cinco"}; //String que representa la moneda
11
12    while(fueradelimites){
13        try{
14            i=  (int) Math.round(Math.random()*9); //Generamos un indice aleatorio
15            System.out.println(texto[i]);
16            fueradelimites=false;
17        }catch(ArrayIndexOutOfBoundsException exc){
18            System.out.println("Fallo en el indice");
19        }
20    }
21
22  }
23 }
```

En este ejemplo, a través de la función de generación de números aleatorios se obtiene el valor del índice `i`. Con dicho valor se accede a una posición del array que contiene cinco cadenas de caracteres. Este acceso, a veces puede generar un error del tipo `ArrayIndexOutOfBoundsException`, que debemos gestionar a través de un `catch`. Al estar el bloque catch dentro de un `while`, se seguirá intentando el acceso hasta que no haya error.

1.3. Delegación de excepciones con throws.

¿Puede haber problemas con las excepciones al usar llamadas a métodos en nuestros programas? Efectivamente, si se produjese una excepción es necesario saber quién será el encargado de solucionarla. Puede ser que sea el propio método llamado o el código que hizo la llamada a dicho método.

Quizá pudiéramos pensar que debería ser el propio método el que se encargue de sus excepciones, aunque es posible hacer que la excepción sea resuelta por el código que hizo la llamada. Cuando un método utiliza una sentencia que puede generar una excepción, pero dicha excepción no es capturada y tratada por él, sino que se encarga su gestión a quién llamó al método, decimos que se ha producido **delegación de excepciones.**

Para establecer esta delegación, en la cabecera del método se declara el tipo de excepciones que puede generar y que deberán ser gestionadas por quien invoque a dicho método. Utilizaremos para ello la sentencia `throws` y tras esa palabra se indica qué excepciones puede provocar el código del método. Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se posará en el `catch` apropiado para esa excepción. Su sintaxis es la siguiente:

```
public class delegacion_excepciones {

     ...

     public int leeaño(BufferedReader lector) throws IOException,
NumberFormatException{
```

```
            String linea = teclado.readLine();
            Return Integer.parseInt(linea);
        }
        ...
}
```

Donde `IOException` y `NumberFormatException`, serían dos posibles excepciones que el método `leeaño` podría generar, pero que no gestiona. Por tanto, un método puede incluir en su cabecera un listado de excepciones que puede lanzar, separadas por comas.

Para saber más

Si deseas saber algo más sobre la delegación de excepciones, te proponemos el siguiente enlace:

Excepciones y delegación de éstas.

Además, te volvemos a remitir al vídeo demostrativo sobre manejo de excepciones en Java que se incluyó en el epígrafe anterior, titulado "capturar una excepción".

8b. Documentación del código.

1. Documentación del código.
    1.1. Etiquetas y posición.
    1.2. Uso de las etiquetas.
    1.3. Orden de las etiquetas.

1. Documentación del código.

Llegados a este punto, vamos a considerar una cuestión de gran importancia, la documentación del código fuente. Piensa en las siguientes cuestiones:

- ¿Quién crees que accederá a la documentación del código fuente? Pues serán los autores del propio código u otros desarrolladores.
- ¿Por qué hemos de documentar nuestro código? Porque facilitaremos su mantenimiento y reutilización.
- ¿Qué debemos documentar? Obligatoriamente: clases, paquetes, constructores, métodos y atributos. Opcionalmente: bucles, partes de algoritmos que estimemos oportuno comentar, ...

A lo largo de nuestra vida como programadores es probable que nos veamos en la necesidad de reutilizar, modificar y mantener nuestro propio código o incluso, código de otros desarrolladores. ¿No crees que sería muy útil que dicho código estuviera convenientemente documentado? ¿Cuántas veces no hemos leído código de otros programadores y quizá no hayamos comprendido qué estaban haciendo en tal o cual método? Como podrás comprender, la generación de una documentación adecuada de nuestros programas puede suponer una inestimable ayuda para realizar ciertos procesos en el software.

Si analizamos la documentación de las clases proporcionada en los paquetes que distribuye Sun, nos daremos cuenta de que dicha documentación ha sido generada con una herramienta llamada Javadoc**.** Pues bien, nosotros también podremos generar la documentación de nuestro código a través de dicha herramienta.

Si desde el principio nos acostumbramos a documentar el funcionamiento de nuestras clases desde el propio código fuente, estaremos facilitando la generación de la futura documentación de nuestras aplicaciones. ¿Cómo lo logramos? A través de una serie de comentarios especiales, llamados **comentarios de documentación** que serán tomados por Javadoc para generar una serie de archivos HTML que permitirán posteriormente, navegar por nuestra documentación con cualquier navegador web.

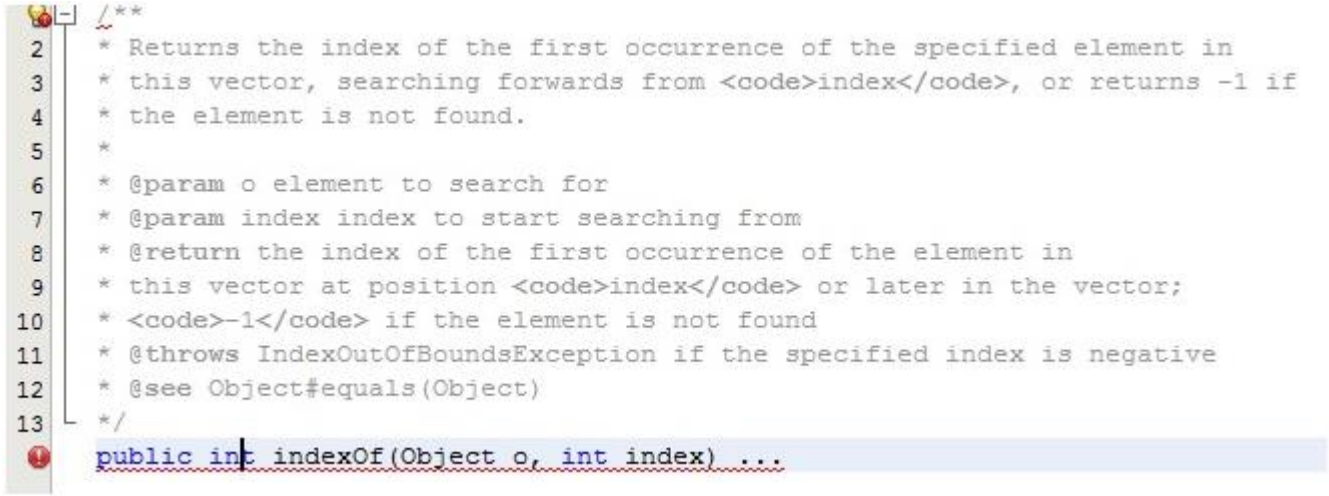
Los comentarios de documentación tienen una marca de comienzo (`/**`) y una marca de fin (`*/`). En su interior podremos encontrar dos partes diferenciadas: una para realizar una descripción y otra en la que encontraremos más etiquetas de documentación. Veamos un ejemplo:

```
/**
* Descripción principal (texto/HTML)
*
* Etiquetas (texto/HTML)
*/
```

Este es el formato general de un comentario de documentación. Comenzamos con la marca de comienzo en una línea. Cada línea de comentario comenzará con un asterisco. El final del comentario de documentación deberá incorporar la marca de fin. Las dos partes diferenciadas de este comentario son:

- **Zona de descripción:** es aquella en la que el programador escribe un comentario sobre la clase, atributo, constructor o método que se vaya a codificar bajo el comentario. Se puede incluir la cantidad de texto que se necesite, pudiendo añadir etiquetas HTML que formateen el texto escrito y así ofrecer una visualización mejorada al generar la documentación mediante Javadoc.
- **Zona de etiquetas:** en esta parte se colocará un conjunto de etiquetas de documentación a las que se asocian textos. Cada etiqueta tendrá un significado especial y aparecerán en lugares determinados de la documentación, una vez haya sido generada.

En la siguiente imagen puedes observar un ejemplo de un comentario de documentación.



Reflexiona

Documentar el código de un programa es añadir suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores sepan qué hacer, sino que además los humanos entiendan qué están haciendo y por qué. Documentar un programa no es sólo un acto de buen hacer del programador por aquello de dejar la obra rematada. Es además una necesidad que sólo se aprecia en su debida magnitud cuando hay errores que reparar o hay que extender el programa con nuevas capacidades o adaptarlo a un nuevo escenario.

1.1. Etiquetas y posición.

Cuando hemos de incorporar determinadas etiquetas a nuestros comentarios de documentación es necesario conocer dónde y qué etiquetas colocar, según el tipo de código que estemos documentando en ese momento. Existirán dos tipos generales de etiquetas:

1. **Etiquetas de bloque:** Son etiquetas que sólo pueden ser incluidas en el bloque de documentación, después de la descripción principal y comienzan con el símbolo @.

2. **Etiquetas en texto:** Son etiquetas que pueden ponerse en cualquier punto de la descripción o en cualquier punto de la documentación asociada a una etiqueta de bloque. Son etiquetas definidas entre llaves, de la siguiente forma {@etiqueta}

En la siguiente tabla podrás encontrar una referencia sobre las diferentes etiquetas y su ámbito de uso.

	@autor	{@code}	{@docRoot}	@deprecated	@exception	{@inheritDoc}	{@link}	{@literal}
Descripción	✓		✓	✓			✓	✓
Paquete	✓		✓	✓			✓	✓
Clases e Interfaces	✓		✓	✓			✓	✓
Atributos			✓	✓			✓	✓
Constructores y métodos			✓	✓	✓	✓	✓	

1.2. Uso de las etiquetas.

¿Cuáles son las etiquetas típicas y su significado? Pasaremos seguidamente a enumerar y describir la función de las etiquetas más habituales a la hora de generar comentarios de documentación.

- `@autor texto con el nombre:` Esta etiqueta sólo se admite en clases e interfaces. El texto después de la etiqueta no necesitará un formato especial. Podremos incluir tantas etiquetas de este tipo como necesitemos.

- `@version texto de la versión:` El texto de la versión no necesitará un formato especial. Es conveniente incluir el número de la versión y la fecha de ésta. Podremos incluir varias etiquetas de este tipo una detrás de otra.

- `@deprecated texto:` Indica que no debería utilizarse, indicando en el texto las causas de ello. Se puede utilizar en todos los apartados de la documentación. Si se ha realizado una sustitución debería indicarse qué utilizar en su lugar. Por ejemplo:

@deprecated El método no funciona correctamente. Se recomienda el uso de {@ling metodoCorrecto}

- **`@exception nombre-excepción texto:`** Esta etiqueta es equivalente a @throws.

- **`@param nombre-atributo texto:`** Esta etiqueta es aplicable a parámetros de constructores y métodos. Describe los parámetros del constructor o método. Nombre-atributo es idéntico al nombre del parámetro. Cada etiqueta @param irá seguida del nombre del parámetro y después de una descripción de éste. Por ejemplo:

@param fromIndex: El índice del primer elemento que debe ser eliminado.

- @return texto: Esta etiqueta se puede omitir en los métodos que devuelven void. Deberá aparecer en todos los métodos, dejando explícito qué tipo o clase de valor devuelve y sus posibles rangos de valores. Veamos un ejemplo:

```
/**
* Chequea si un vector no contiene elementos.
*
* @return <code>verdadero</code>si solo si este vector no contiene
componentes, esto es, su tamaño es cero;
* <code>falso</code> en cualquier otro caso.
*/
public boolean VectorVacio() {
return elementCount == 0;
}
```

- **`@see referencia:`** Se aplica a clases, interfaces, constructores, métodos, atributos y paquetes. Añade enlaces de referencia a otras partes de la documentación. Podremos añadir a la etiqueta: cadenas de caracteres, enlaces HTML a páginas y a otras zonas del código. Por ejemplo:

`* @see "Diseño de patrones`: La reusabilidad de los elementos de la programación orientada a objetos"
* @see Web Accessibility Initiative
* @see String#equals(Object) equals

- **`@throws nombre-excepción texto:`** En nombre-excepción tendremos que indicar el nombre completo de ésta. Podremos añadir una etiqueta por cada excepción que se lance explícitamente con una cláusula **`throws`**, pero siguiendo el orden alfabético. Esta etiqueta es aplicable a constructores y métodos, describiendo las posibles excepciones del constructor/método.

1.3. Orden de las etiquetas.

Las etiquetas deben disponerse en un orden determinado. Ese orden es el siguiente:

Orden de etiquetas de comentarios de documentación.	
Etiqueta.	Descripción.
`@autor`	En clases e interfaces. Se pueden poner varios. Es mejor ponerlas en orden cronológico.
`@version`	En clases e interfaces.
`@param`	En métodos y constructores. Se colocarán tantos como parámetros tenga el constructor o método. Mejor en el mismo orden en el que se encuentren declarados.
`@return`	En métodos.
`@exception`	En constructores y métodos. Mejor en el mismo orden en el que se han declarado, o en orden alfabético.
`@throws`	Es equivalente a `@exception`.
`@see`	Podemos poner varios. Comenzaremos por los más generales y después los más específicos.
`@deprecated`	

Para saber más

Si quieres conocer cómo obtener a través de Javadoc la documentación de tus aplicaciones, sigue los siguientes enlaces:

Documentación con Javadoc.

Documentación de clases y métodos con Javadoc.