

Uso de estructuras de control.

Caso práctico

Los miembros de BK Programación están inmersos en el mundo de Java, ya conocen una buena parte del lenguaje y ahora van a empezar a controlar el comportamiento de sus programas.

María pregunta a **Juan**: -Dime Juan, ¿En Java también hay condicionales y bucles?

-Efectivamente **María**, como la gran mayoría de los lenguajes de programación, Java incorpora estructuras que nos permiten tomar decisiones, repetir código, etc. Cada estructura tiene sus ventajas e inconvenientes y hay que saber dónde utilizar cada una de ellas. - Aclara **Juan**.



Materiales formativos de FP Online propiedad del Ministerio de Educación, Cultura y Deporte.

[Aviso Legal](#)

1.- Introducción.

En unidades anteriores has podido aprender cuestiones básicas sobre el lenguaje Java: definición de variables, tipos de datos, asignación de valores, uso de literales, diferentes operadores que se pueden aplicar, conversiones de tipos, inserción de comentarios, etc. Posteriormente, nos sumergimos de lleno en el mundo de los objetos. Primero hemos conocido su filosofía, para más tarde ir recorriendo los conceptos y técnicas más importantes relacionadas con ellos: propiedades, métodos, clases, declaración y uso de objetos, librerías, etc.

Vale, parece ser que tenemos los elementos suficientes para comenzar a generar programas escritos en Java, ¿Seguro?

Reflexiona

Piensa en la siguiente pregunta: ¿Cómo un programa puede determinar la aparición en pantalla de un mensaje de ÉXITO o ERROR, según los datos de entrada aportados por un usuario?

Como habrás deducido, con lo que sabemos hasta ahora no es suficiente. Existen múltiples situaciones que nuestros programas deben representar y que requieren tomar ciertas decisiones, ofrecer diferentes alternativas o llevar a cabo determinadas operaciones repetitivamente para conseguir sus objetivos.

Si has programado alguna vez o tienes ciertos conocimientos básicos sobre lenguajes de programación, sabes que la gran mayoría de lenguajes poseen estructuras que permiten a los programadores controlar el flujo de la información de sus programas. Esto realmente es una ventaja para la persona que está aprendiendo un nuevo lenguaje, o tienen previsto aprender más de uno, ya que estas estructuras suelen ser comunes a todos (con algunos cambios de [sintaxis](#)). Es decir, si conocías sentencias de control de flujo en otros lenguajes, lo que vamos a ver a lo largo de esta unidad te va a sonar bastante.



Para alguien que no ha programado nunca, un ejemplo sencillo le va a permitir entender qué es eso de las sentencias de control de flujo. Piensa en un fontanero (programador), principalmente trabaja con agua (datos) y se encarga de hacer que ésta fluya por donde él quiere (programa) a través de un conjunto de tuberías, codos, latiguillos, llaves de paso, etc. (sentencias de control de flujo). Pues esas estructuras de control de flujo son las que estudiaremos, conoceremos su estructura, funcionamiento, cómo utilizarlas y dónde. A través de ellas, al construir nuestros programas podremos hacer que los datos (agua) fluyan por los caminos adecuados para representar la realidad del problema y obtener un resultado adecuado.

Los tipos de **estructuras** de programación que se emplean **para el control del flujo** de los datos son las siguientes:

- ✓ **Secuencia:** compuestas por 0, 1 o N sentencias que se ejecutan en el orden en que han sido escritas. Es la estructura más sencilla y sobre la que se construirán el resto de estructuras.
- ✓ **Selección:** es un tipo de sentencia especial de decisión y de un conjunto de secuencias de instrucciones asociadas a ella. Según la evaluación de la sentencia de decisión se generará un resultado (que suele ser verdadero o falso) y en función de éste, se ejecutarán una secuencia de instrucciones u otra. Las estructuras de selección podrán ser simples, compuestas y múltiples.
- ✓ **Iteración:** es un tipo de sentencia especial de decisión y una secuencia de instrucciones que pueden ser repetidas según el resultado de la evaluación de la sentencia de decisión. Es decir, la secuencia de instrucciones se ejecutará repetidamente si la sentencia de decisión arroja un valor correcto, en otro caso la estructura de repetición se detendrá.

Además de las sentencias típicas de control de flujo, en esta unidad haremos una revisión de las **sentencias de salto**, que aunque no son demasiado recomendables, es necesario conocerlas. Como nuestros programas podrán generar errores y situaciones especiales, echaremos un vistazo al **manejo de excepciones** en Java. Posteriormente, analizaremos la mejor manera de llevar a cabo las **pruebas** de nuestros programas y la **depuración** de los mismos. Y finalmente, aprenderemos a valorar y utilizar las herramientas de **documentación de programas**.

Vamos entonces a ponemos el mono de trabajo y a coger nuestra caja de herramientas, ¡a ver si no nos mojamos mucho!

2.- Sentencias y bloques.

Caso práctico

Ada valora muy positivamente en un programador el orden y la pulcritud. Organizar correctamente el código fuente es de vital importancia cuando se trabaja en entornos colaborativos, en los que son varios los desarrolladores los que forman los equipos de programación. Por ello, incide en la necesidad de recordar a **Juan y María** las nociones básicas a la hora de escribir programas.



Este epígrafe lo utilizaremos para reafirmar cuestiones que son obvias y que en el transcurso de anteriores unidades se han dado por sabidas. Aunque, a veces, es conveniente recordar. Lo haremos como un conjunto de FAQ:

- ✓ **¿Cómo se escribe un programa sencillo?** Si queremos que un programa sencillo realice instrucciones o sentencias para obtener un determinado resultado, es necesario colocar éstas una detrás de la otra, exactamente en el orden en que deben ejecutarse.
- ✓ **¿Podrían colocarse todas las sentencias una detrás de otra, separadas por puntos y comas en una misma línea?** Claro que sí, pero no es muy recomendable. Cada sentencia debe estar escrita en una línea, de esta manera tu código será mucho más legible y la localización de errores en tus programas será más sencilla y rápida. De hecho, cuando se utilizan herramientas de programación, los errores suelen asociarse a un número o números de línea.
- ✓ **¿Puede una misma sentencia ocupar varias líneas en el programa?** Sí. Existen sentencias que, por su tamaño, pueden generar varias líneas. Pero siempre finalizarán con un punto y coma.
- ✓ **¿En Java todas las sentencias se terminan con punto y coma?** Efectivamente. Si detrás de una sentencia ha de venir otra, pondremos un punto y coma. Escribiendo la siguiente sentencia en una nueva línea. Pero en algunas ocasiones, sobre todo cuando utilizamos estructuras de control de flujo, detrás de la cabecera de una estructura de este tipo no debe colocarse punto y coma. No te preocupes, lo entenderás cuando analicemos cada una de ellas.
- ✓ **¿Qué es la sentencia nula en Java?** La sentencia nula es una línea que no contiene ninguna instrucción y en la que sólo existe un punto y coma. Como su nombre indica, esta sentencia no hace nada.
- ✓ **¿Qué es un bloque de sentencias?** Es un conjunto de sentencias que se encierra entre llaves y que se ejecutaría como si fuera una única orden. Sirve para agrupar sentencias y para clarificar el código. Los bloques de sentencias son utilizados en Java en la práctica totalidad de estructuras de control de flujo, clases, métodos, etc. La siguiente tabla muestra dos formas de construir un bloque de sentencias.

Bloques de sentencias.

Bloque de sentencias 1	Bloque de sentencias 2
<pre>{sentencia1; sentencia2;...; sentencia N;}</pre>	<pre>{ sentencia1; sentencia2; ...; sentenciaN; }</pre>

- ✓ **¿En un bloque de sentencias, éstas deben estar colocadas con un orden exacto?** En ciertos casos sí, aunque si al final de su ejecución se obtiene el mismo resultado, podrían ocupar diferentes posiciones

Debes conocer

Accede a los tres archivos que te ofrecemos a continuación y compara su código fuente. Verás que los tres obtienen el mismo resultado, pero la organización de las sentencias que los componen es diferente entre ellos.

Sentencias y bloques.

Sentencias, bloques y diferentes organizaciones		
Sentencias en orden secuencial. (0.01 MB)	Sentencias v declaraciones de variables. (0.01 MB)	Sentencias, declaraciones y organización del código. (0.01 MB)
En este primer archivo, las sentencias están colocadas en orden secuencial.	En este segundo archivo, se declaran al principio las variables necesarias. En Java no es imprescindible hacerlo así, pero sí que antes de utilizar cualquier variable ésta debe estar previamente declarada. Aunque la declaración de dicha variable puede hacerse en cualquier lugar de nuestro programa.	En este tercer archivo, podrás apreciar que se ha organizado el código en las siguientes partes: declaración de variables, petición de datos de entrada, procesamiento de dichos datos y obtención de la salida. Este tipo de organización está más estandarizada y hace que nuestros programas ganen en legibilidad.

Construyas de una forma o de otra tus programas, debes tener en cuenta siempre en Java las siguientes premisas:

- ✓ Declara cada variable antes de utilizarla.
- ✓ Inicializa con un valor cada variable la primera vez que la utilices.
- ✓ No es recomendable usar variables no inicializadas en nuestros programas, pueden provocar errores o resultados imprevistos.



Autoevaluación

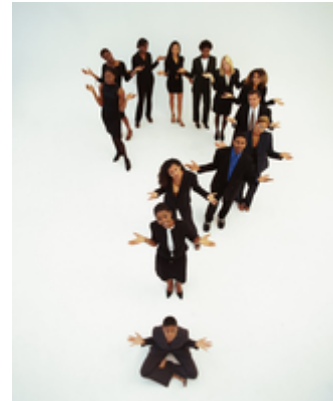
Indica qué afirmación es correcta:

- ☐ Para crear un bloque de sentencias, es necesario delimitar éstas entre llaves. Este bloque funcionará como si hubiéramos colocado una única orden.
- ☐ La sentencia nula en Java, se puede representar con un punto y coma sólo en una única línea.
- ☐ Para finalizar en Java cualquier sentencia, es necesario hacerlo con un punto y coma.
- ☐ Todas las afirmaciones son correctas.

3.- Estructuras de selección.

Caso práctico

Juan está desarrollando un método en el que ha de comparar los valores de las entradas de un usuario y una contraseña introducidas desde el teclado, con los valores almacenados en una base de datos. Para poder hacer dicha comparación necesitará utilizar una estructura condicional que le permita llevar a cabo esta operación, incluso necesitará que dicha estructura condicional sea capaz de decidir qué hacer en función de si ambos valores son correctos o no.



Al principio de la unidad nos hacíamos esta pregunta: ¿Cómo un programa puede determinar la aparición en pantalla de un mensaje de ÉXITO o ERROR, según los datos de entrada aportados por un usuario? Esta y otras preguntas se nos plantean en múltiples ocasiones cuando desarrollamos programas.

¿Cómo conseguimos que nuestros programas puedan tomar decisiones? Para comenzar, lo haremos a través de las estructuras de selección. Estas estructuras constan de una sentencia especial de decisión y de un conjunto de secuencias de instrucciones.

El funcionamiento es sencillo, la sentencia de decisión será evaluada y ésta devolverá un valor (verdadero o falso), en función del valor devuelto se ejecutará una secuencia de instrucciones u otra. Por ejemplo, si el valor de una variable es mayor o igual que 5 se imprime por pantalla la palabra APROBADO y, si es menor, se imprime SUSPENSO. Para este ejemplo, la comprobación del valor de la variable será la sentencia especial de decisión. La impresión de la palabra APROBADO será una secuencia de instrucciones y la impresión de la palabra SUSPENSO será otra. Cada secuencia estará asociada a cada uno de los resultados que puede arrojar la evaluación de la sentencia especial de decisión.

Recomendación

En el lenguaje de programación C, verdadero o falso se representan mediante un literal entero. 0 representará Falso y 1 o cualquier otro valor, representará Verdadero. Como sabes, en Java las variables de tipo `booleano` sólo podrán tomar los valores `true` (verdadero) o `false` (falso).

La evaluación de las sentencias de decisión o expresiones que controlan las estructuras de selección, devolverán siempre un valor verdadero o falso.

Las estructuras de selección se dividen en:

1. Estructuras de selección simples o estructura `if`.
2. Estructuras de selección compuestas o estructura `if-else`.
3. Estructuras de selección basadas en el operador condicional.
4. Estructuras de selección múltiples o estructura `switch`.

A continuación, detallaremos las características y funcionamiento de cada una de ellas. Es importante que a través de los ejemplos que vamos a ver, puedas determinar en qué circunstancias utilizar cada una de estas

estructuras. Aunque un mismo problema puede ser resuelto con diferentes estructuras e incluso, con diferentes combinaciones de éstas.



3.1.- Estructura if / if-else.

La estructura **if** es una estructura de selección o estructura condicional, en la que se evalúa una expresión lógica o sentencia de decisión y en función del resultado, se ejecuta una sentencia o un bloque de éstas.

La estructura **if** puede presentarse de las siguientes formas:

Estructura if e

Estructura if simple.		
Sintaxis: if (expresión-lógica) sentencial;	Sintaxis: if (expresión-lógica) { sentencia1; sentencia2; ...; sentenciaN; }	Si
Funcionamiento: Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecuta la sentencia1 o bien el bloque de sentencias asociado. Si el resultado de dicha evaluación es falso, no se ejecutará ninguna instrucción asociada a la estructura condicional.		Fu Si pri fals seq

Haciendo una interpretación cercana al pseudocódigo tendríamos que si se cumple la condición (expresión lógica), se ejecutará un conjunto de instrucciones y si no se cumple, se ejecutará otro conjunto de instrucciones.

Hay que tener en cuenta que la cláusula **else** de la sentencia **if** no es obligatoria. En algunos casos no necesitaremos utilizarla, pero sí se recomienda cuando es necesario llevar a cabo alguna acción en el caso de que la expresión lógica no se cumpla.

En aquellos casos en los que no existe cláusula **else**, si la expresión lógica es falsa, simplemente se continuarán ejecutando las siguientes sentencias que aparezcan bajo la estructura condicional **if**.

Los condicionales **if** e **if-else** pueden anidarse, de tal forma que dentro de un bloque de sentencias puede incluirse otro **if** o **if-else**. El nivel de anidamiento queda a criterio del programador, pero si éste es demasiado profundo podría provocar problemas de eficiencia y legibilidad en el código. En otras ocasiones, un nivel de anidamiento excesivo puede denotar la necesidad de utilización de otras estructuras de selección más adecuadas.

Cuando se utiliza anidamiento de este tipo de estructuras, es necesario poner especial atención en saber a qué **if** está asociada una cláusula **else**. Normalmente, un **else** estará asociado con el **if** inmediatamente superior o más cercano que exista dentro del mismo bloque y que no se encuentre ya asociado a otro **else**.



Debes conocer

Para completar la información que debes saber sobre las estructuras **if** e **if-else**, accede al siguiente enlace. En él podrás analizar el código de un programa que realiza el cálculo de la nota de un examen de tipo test. Además de calcular el valor de la nota, se ofrece como salida la calificación no numérica de dicho examen. Para obtenerla, se combinarán las diferentes estructuras condicionales aprendidas hasta ahora.

Presta especial atención a los comentarios incorporados en el código fuente, así como a la forma de combinar las estructuras condicionales y a las expresiones lógicas utilizadas en ellas.

[Uso la estructura condicional if e if-else.](#) (0.01 MB)



Autoevaluación

¿Cuándo se mostrará por pantalla el mensaje incluido en el siguiente fragmento de código?

```
If (numero % 2 == 0);  
System.out.print("El número es par /n");
```

- ☐ Nunca.
- ☐ Siempre.
- ☐ Cuando el resto de la división entre 2 del contenido de la variable **numero**, sea cero.

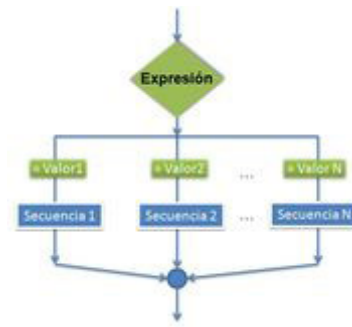
3.2.- Estructura switch.

¿Qué podemos hacer cuando nuestro programa debe elegir entre más de dos alternativas?, una posible [solución](#) podría ser emplear estructuras **if** anidadas, aunque no siempre esta [solución](#) es la más eficiente. Cuando estamos ante estas situaciones podemos utilizar la estructura de selección múltiple **switch**. En la siguiente tabla se muestra tanto la sintaxis, como el funcionamiento de esta estructura.

Estructura switch.

Estructura switch	
Sintaxis: <pre>switch (expresion) { case valor1: sentencial_1; sentencial_2; ... break; case valorN: sentenciaN_1; sentenciaN_2; ... break; default: sentencias-default; }</pre>	Condiciones: <ul style="list-style-type: none">✓ Donde expresión debe ser del tipo char, byte, short o int, y las constantes de cada case deben ser de este tipo o de un tipo compatible.✓ La expresión debe ir entre paréntesis.✓ Cada case llevará asociado un valor y se finalizará con dos puntos.✓ El bloque de sentencias asociado a la cláusula default puede finalizar con una sentencia de ruptura break o no.
Funcionamiento: <ul style="list-style-type: none">✓ Las diferentes alternativas de esta estructura estarán precedidas de la cláusula case que se ejecutará cuando el valor asociado al case coincida con el valor obtenido al evaluar la expresión del switch.✓ En las cláusulas case, no pueden indicarse expresiones condicionales, rangos de valores o listas de valores. (otros lenguajes de programación sí lo permiten). Habrá que asociar una cláusula case a cada uno de los valores que deban ser tenidos en cuenta.✓ La cláusula default será utilizada para indicar un caso por defecto, las sentencias asociadas a la cláusula default se ejecutarán si ninguno de los valores indicados en las cláusulas case coincide con el resultado de la evaluación de la expresión de la estructura switch.✓ La cláusula default puede no existir, y por tanto, si ningún case ha sido activado finalizaría el switch.✓ Cada cláusula case puede llevar asociadas una o varias sentencias, sin necesidad de delimitar dichos bloques por medio de llaves.✓ En el momento en el que el resultado de la evaluación de la expresión coincide con alguno de los valores asociados a las cláusulas case, se ejecutarán todas las instrucciones asociadas hasta la aparición de una sentencia break de ruptura. (la sentencia break se analizará en epígrafes posteriores)	

En resumen, se ha de comparar el valor de una expresión con un conjunto de constantes, si el valor de la expresión coincide con algún valor de dichas constantes, se ejecutarán los bloques de instrucciones asociados a cada una de ellas. Si no existiese coincidencia, se ejecutarían una serie de instrucciones por defecto.



Debes conocer

Accede al siguiente fragmento de código en el que se resuelve el cálculo de la nota de un examen de tipo test, utilizando la estructura **switch**.

[Uso la estructura condicional múltiple switch.](#) (0.01 MB)

4.- Estructuras de repetición.

Caso práctico

Juan ya tiene claro cómo realizar la comprobación de los valores de usuario y contraseña introducidos por teclado, pero le surge una duda: ¿Cómo podría controlar el número de veces que el usuario ha introducido mal la contraseña?

Ada le indica que podría utilizar una estructura de repetición que solicitase al usuario la introducción de la contraseña hasta un máximo de tres veces. Aunque comenta que puede haber múltiples soluciones y todas válidas, lo importante es conocer las herramientas que podemos emplear y saber cuándo aplicarlas.



Nuestros programas ya son capaces de controlar su ejecución teniendo en cuenta determinadas condiciones, pero aún hemos de aprender un conjunto de estructuras que nos permita repetir una secuencia de instrucciones determinada. La función de estas estructuras es repetir la ejecución de una serie de instrucciones teniendo en cuenta una condición.

A este tipo de estructuras se las denomina **estructuras de repetición**, estructuras repetitivas, **bucles** o estructuras **iterativas**. En Java existen cuatro clases de bucles:

- ✓ Bucle **for** (repite para)
- ✓ Bucle **for/in** (repite para cada)
- ✓ Bucle **while** (repite mientras)
- ✓ Bucle **Do While** (repite hasta)



Los bucles **for** y **for/in** se consideran bucles **controlados por contador**. Por el contrario, los bucles **while** y **do...while** se consideran bucles **controlados por sucesos**.

La utilización de unos bucles u otros para solucionar un problema dependerá en gran medida de las siguientes preguntas:

- ✓ ¿Sabemos **a priori** cuántas veces necesitamos repetir un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de repetir un conjunto de instrucciones si una condición satisface un conjunto de valores?
- ✓ ¿Sabemos hasta cuándo debemos estar repitiendo un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de estar repitiendo un conjunto de instrucciones mientras se cumpla una condición?

Recomendación

Estudia cada tipo de estructura repetitiva, conoce su funcionamiento y podrás llegar a la conclusión de que algunos de estos bucles son equivalentes entre sí. Un mismo problema, podrá ser resuelto empleando diferentes tipos de bucles y obtener los mismos resultados.

Estas y otras preguntas tendrán su respuesta en cuanto analicemos cada una de estructuras repetitivas en detalle.

4.1.- Estructura for.

Hemos indicado anteriormente que el bucle **for** es un bucle controlado por contador. Este tipo de bucle tiene las siguientes características:

- ✓ Se ejecuta un número determinado de veces.
- ✓ Utiliza una variable contadora que controla las iteraciones del bucle.

En general, existen tres operaciones que se llevan a cabo en este tipo de bucles:

- ✓ Se inicializa la variable contadora.
- ✓ Se evalúa el valor de la variable contador, por medio de una comparación de su valor con el número de iteraciones especificado.
- ✓ Se modifica o actualiza el valor del contador a través de incrementos o decrementos de éste, en cada una de las iteraciones.



Recomendación

La inicialización de la variable contadora debe realizarse correctamente para garantizar que el bucle se lleve a cabo, al menos, la primera repetición de su código interno.

La condición de terminación del bucle debe variar en el interior del mismo, de no ser así, podemos caer en la creación de un bucle infinito. Cuestión que se debe evitar por todos los medios.

Es necesario estudiar el número de veces que se repite el bucle, pues debe ajustarse al número de veces estipulado.

En la siguiente tabla, podemos ver la especificación de la estructura **for**:

Estructura repetitiva for.

Estructura repetitiva for	
<p>Sintaxis:</p> <pre>for (inicialización; condición; iteración) sentencia;</pre> <p>(estructura for con una única sentencia)</p>	<p>Donde inicialización es una expresión en la que se inicializa una variable de control, que será la encargada de controlar el final del bucle. Donde condición es una expresión que evaluará la variable de control. Mientras la condición sea falsa, el cuerpo del bucle estará repitiéndose. Cuando la condición se cumpla, terminará la ejecución del bucle.</p> <p>Donde iteración indica la manera en la que la variable de control va cambiando en cada iteración del bucle. Podrá ser mediante incremento o decremento, y no solo de uno en uno.</p>
<p>Sintaxis:</p> <pre>for (inicialización; condición; iteración) { sentencia1; sentencia2; ... sentenciaN; }</pre> <p>(estructura for con un bloque de sentencias)</p>	

Debes conocer

Como venimos haciendo para el resto de estructuras, accede al siguiente archivo Java y podrás analizar un ejemplo de utilización del bucle for para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle.

[Uso la estructura repetitiva for.](#) (0.01 MB)



Autoevaluación

Cuando construimos la cabecera de un bucle for, podemos prescindir de alguno de los tres elementos que la forman e incluso, podemos utilizar más de una variable contadora separando éstas por comas. Pero, ¿Qué conseguiremos si construimos un bucle de la siguiente forma?

```
for (;;){ //instrucciones }
```

- ☐ Un bucle infinito.
- ☐ Nada, dará un error.
- ☐ Un bucle que se ejecutaría una única vez.

4.2.- Estructura for/in.

Junto a la estructura **for**, **for/in** también se considera un bucle controlado por contador. Este bucle es una mejora incorporada en la versión 5.0. de Java.

Este tipo de bucles permite realizar recorridos sobre **arrays** y colecciones de objetos. Los **arrays** son colecciones de variables que tienen el mismo tipo y se referencian por un nombre común. Así mismo, las colecciones de objetos son objetos que se dice son iterables, o que se puede iterar sobre ellos.

Este bucle es nombrado también como bucle **for** mejorado, o bucle **foreach**. En otros lenguajes de programación existen bucles muy parecidos a este.

La sintaxis es la siguiente:

```
for (declaración: expresión) {  
    sentencia1;  
    ...  
    sentenciaN;  
}
```

- ✓ Donde expresión es un array o una colección de objetos.
- ✓ Donde declaración es la declaración de una variable cuyo tipo sea compatible con expresión. Normalmente, será el tipo y el nombre de la variable a declarar.

El funcionamiento consiste en que para cada elemento de la expresión, guarda el elemento en la variable declarada y realiza las instrucciones contenidas en el bucle. Después, en cada una de las iteraciones del bucle tendremos en la variable declarada el elemento actual de la expresión. Por tanto, para el caso de los **arrays** y de las colecciones de objetos, se recorrerá desde el primer elemento que los forma hasta el último.

Observa el contenido del código representado en la siguiente imagen, puedes apreciar cómo se construye un bucle de este tipo y su utilización sobre un **array**.

Los bucles **for/in** permitirán al programador despreocuparse del número de veces que se ha de iterar, pero no sabremos en qué iteración nos encontramos salvo que se añada artificialmente alguna variable contadora que nos pueda ofrecer esta información.



4.3.- Estructura while.

El bucle **while** es la primera de las estructuras de repetición controladas por sucesos que vamos a estudiar. La utilización de este bucle responde al planteamiento de la siguiente pregunta: ¿Qué podemos hacer si lo único que sabemos es que se han de repetir un conjunto de instrucciones mientras se cumpla una determinada condición?

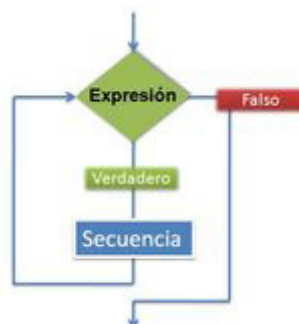
La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle podría ser necesario ejecutarlas o no. Es decir, en el bucle **while** siempre se evaluará la condición que lo controla, y si dicha condición es cierta, el cuerpo del bucle se ejecutará una vez, y se seguirá ejecutando mientras la condición sea cierta. Pero si en la evaluación inicial de la condición ésta no es verdadera, el cuerpo del bucle no se ejecutará.

Es imprescindible que en el interior del bucle **while** se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

Estructura repetitiva while.

Estructura repetitiva while	
Sintaxis: <pre>while (condición) sentencia;</pre>	Sintaxis: <pre>while (condición) { sentencia1; ... sentenciaN; }</pre>
Funcionamiento: Mientras la condición sea cierta, el bucle se repetirá, ejecutando la/s instrucción/es de su interior. En el momento en el que la condición no se cumpla, el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle while. La condición se evaluará siempre al principio, y podrá darse el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca si no se satisface la condición de partida.	

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.



Debes conocer

Accede al siguiente archivo java y podrás analizar un ejemplo de utilización del bucle **while** para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle. Como podrás comprobar, el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle for.

[Uso la estructura repetitiva while.](#) (0.01 MB)



Autoevaluación

Utilizando el siguiente fragmento de código estamos construyendo un bucle infinito. ¿Verdadero o Falso?

```
while (true) System.out.println("Imprimiendo desde dentro del bucle \n");
```



Verdadero. ☐ Falso. ☐

4.4.- Estructura do-while.

La segunda de las estructuras repetitivas controladas por sucesos es **do-while**. En este caso, la pregunta que nos planteamos es la siguiente: ¿Qué podemos hacer si lo único que sabemos es que se han de ejecutar, al menos una vez, un conjunto de instrucciones y seguir repitiéndose hasta que se cumpla una determinada condición?

La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle necesitan ser ejecutadas, al menos, una vez y repetir su ejecución hasta que la condición sea verdadera. Por tanto, en esta estructura repetitiva siempre se ejecuta el cuerpo del bucle una primera vez.

Es imprescindible que en el interior del bucle se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

Estructura repetitiva do-while.

Estructura repetitiva do-while	
Sintaxis: <pre>do sentencia; while (condición);</pre>	Sintaxis: <pre>do { sentencia1; ... sentenciaN; } while (condición);</pre>
Funcionamiento: El cuerpo del bucle se ejecuta la primera vez, a continuación se evaluará la condición y, si ésta es falsa, el cuerpo el bucle volverá a repetirse. El bucle finalizará cuando la evaluación de la condición sea verdadera. En ese momento el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle do-while. La condición se evaluará siempre después de una primera ejecución del cuerpo del bucle, por lo que no se dará el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca.	

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.



Debes conocer

Accede al siguiente [archivo java](#) y podrás analizar un ejemplo de utilización del bucle **do-while** para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle. Como podrás comprobar,

el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle **for** y el bucle **while**.

[Uso la estructura repetitiva do-while.](#) (0.01 MB)

5.- Estructuras de salto.

Caso práctico

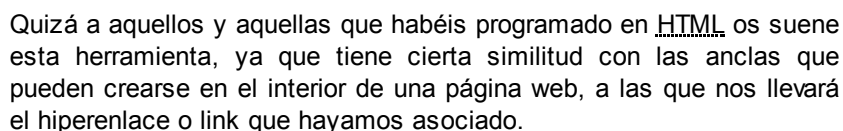
Juan recuerda que algunos lenguajes de programación permitían realizar saltos a lo largo de la ejecución de los programas y conoce dos sentencias que aún se siguen utilizando para ello.

Ada, mientras toma un libro sobre Java de la estantería del despacho, le aconseja: -las instrucciones de salto a veces han sido mal valoradas por la comunidad de programadores, pero en Java algunas de ellas son totalmente necesarias. Mira, en este libro se habla del uso de las sentencias **break**, **continue** y **return**.



¿Saltar o no saltar? he ahí la cuestión. En la gran mayoría de libros de programación y publicaciones de Internet, siempre se nos recomienda que prescindamos de sentencias de salto incondicional, es más, se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento de los mismos. Pero Java incorpora ciertas sentencias o estructuras de salto que es necesario conocer y que pueden ser útiles en algunas partes de nuestros programas.

Estas estructuras de salto corresponden a las sentencias **break**, **continue**, las **etiquetas de salto** y la sentencia **return**. Pasamos ahora a analizar su sintaxis y funcionamiento.



5.3.- Sentencia Return.

Ya sabemos como modificar la ejecución de bucles y estructuras condicionales múltiples, pero ¿Podríamos modificar la ejecución de un método? ¿Es posible hacer que éstos detengan su ejecución antes de que finalice el código asociado a ellos? Sí es posible, a través de la sentencia **return** podremos conseguirlo.



La sentencia **return** puede utilizarse de dos formas:

- ✓ Para terminar la ejecución del método donde esté escrita, con lo que transferirá el control al punto desde el que se hizo la llamada al método, continuando el programa por la sentencia inmediatamente posterior.
- ✓ Para devolver o retornar un valor, siempre que junto a **return** se incluya una expresión de un tipo determinado. Por tanto, en el lugar donde se invocó al método se obtendrá el valor resultante de la evaluación de la expresión que acompañaba al método.

En general, una sentencia **return** suele aparecer al final de un método, de este modo el método tendrá una entrada y una salida. También es posible utilizar una sentencia **return** en cualquier punto de un método, con lo que éste finalizará en el lugar donde se encuentre dicho **return**. No será recomendable incluir más de un **return** en un método y por regla general, deberá ir al final del método, como hemos comentado.

El valor de retorno es opcional, si lo hubiera debería de ser del mismo tipo o de un tipo compatible al tipo del valor de retorno definido en la cabecera del método, pudiendo ser desde un entero a un objeto creado por nosotros. Si no lo tuviera, el tipo de retorno sería **void**, y **return** serviría para salir del método sin necesidad de llegar a ejecutar todas las instrucciones que se encuentran después del **return**.

Para saber más

En el siguiente archivo java encontrarás el código de un programa que obtiene la suma de dos números, empleando para ello un método sencillo que retorna el valor de la suma de los números que se le han pasado como parámetros. Presta atención a los comentarios y fíjate en las conversiones a entero de la entrada de los operandos por consola.

[Uso de return en métodos.](#) (0.01 MB)



Autoevaluación

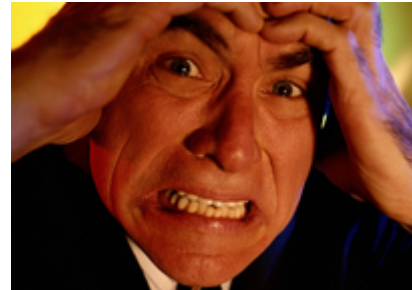
¿Qué afirmación es correcta?

- ☐ Con **return**, se puede finalizar la ejecución del método en el que se encuentre.
- ☐ Con **return**, siempre se retornará un valor del mismo tipo o de un tipo compatible al definido en la cabecera del método.
- ☐ Con **return**, puede retornarse un valor de un determinado tipo y suele hacerse al final del método. Además, el resto de respuestas también son correctas.

6.- Excepciones.

Caso práctico

Para que las aplicaciones desarrolladas por BK Programación mantengan una valoración positiva a lo largo del tiempo por parte de sus clientes, es necesario que éstas no se vean comprometidas durante su ejecución. Hay innumerables ocasiones en las que programas que aparentemente son formidables (por su interfaz, facilidad de uso, etc.) comienzan a generar errores en tiempo de ejecución que hacen que el usuario desconfíe de ellos día a día.



Para evitar estas situaciones, Ada va a fomentar en María y Juan la cultura de la detección, control y **solución** de errores a través de las poderosas herramientas que Java les ofrece.

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.



6.1.- 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
}
```

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 `ArithmeticException` 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.

Debes conocer

Si deseas conocer en mayor detalle el manejo de excepciones, te aconsejamos el siguiente enlace en el que podrás encontrar un vídeo ilustrativo sobre su creación y manejo.

Excepciones en Java.

Gestion de excepciones con try c...



[Resumen textual alternativo](#)



Autoevaluación

Si en un programa no capturamos una excepción, será la máquina virtual de Java la que lo hará por nosotros, pero inmediatamente detendrá la ejecución del programa y mostrará una traza y un mensaje de error. Siendo una traza, la forma de localizar dónde se han producido errores. ¿Verdadero o Falso?

Verdadero. ☐ Falso. ☐

6.2.- 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 irreparable. 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.



```
17 public static void main(String[] args) {
18     //Creamos un array de cadenas
19     String[] palabras = {"Java", "Python", "JavaScript", "C++", "C"};
20     //Creamos un objeto Random para generar números aleatorios
21     Random random = new Random();
22
23     //Creamos un bucle while para simular la reanudación
24     while (true) {
25         //Generamos un índice aleatorio entre 0 y 4
26         int indice = random.nextInt(5);
27         //Intentamos acceder al elemento del array
28         String palabra = palabras[indice];
29         //Imprimimos la palabra
30         System.out.println(palabra);
31     }
32 }
```

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.

6.3.- 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 leeañó(BufferedReader lector) throws IOException, NumberFormatException {  
        String linea = teclado.readLine();  
        Return Integer.parseInt(linea);  
    }  
    ...  
}
```

Donde **IOException** y **NumberFormatException**, serían dos posibles excepciones que el método **leeañó** 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".

7.- Prueba de programas.

Caso práctico

Continuando con el especial interés de BK Programación porque sus aplicaciones sean de verdadera calidad, **Ada** está dispuesta a emprender un plan de pruebas de software que sea capaz de reducir al mínimo posible los errores que puedan contener las aplicaciones que están desarrollando.

Juan y María ya conocían de la existencia de ciertas pruebas que se suelen hacer al software, pero necesitarán aprender bien las técnicas y utilizar las herramientas que los entornos de desarrollo ofrecen para tal proceso.



A veces, los programas son complejos y es difícil hacer que éstos funcionen correctamente. Las pruebas del software son un conjunto de técnicas utilizadas para verificar que un programa lleva a cabo su tarea correctamente. Viéndolo desde otro punto de vista, podríamos decir que la realización de pruebas en nuestros programas intentan revelar la existencia de errores. Cuando detectamos que existe un error, necesitamos localizarlo llevando a cabo técnicas de depuración de código, para luego acometer las modificaciones necesarias que eliminen dicho error.

No obstante, las técnicas de prueba del software no garantizan que todos los errores vayan a salir a la luz, y aún así, habrá programas de gran tamaño que sigan conteniendo errores ocultos en ellos. De todas maneras, la realización de pruebas de software está considerada como una etapa de gran importancia en el desarrollo de software. Casi la mitad del tiempo dedicado al desarrollo de un programa, se emplea en la realización de pruebas al mismo. En algunas organizaciones, la realización de pruebas se considera tan importante que existen equipos de desarrollo de software en los que hay tantos probadores de software como programadores.

Los procesos de prueba y depuración requieren una considerable cantidad de tiempo y esfuerzo, haciéndose difícil tomar una decisión acertada entre continuar la prueba del programa o entregarlo en su estado actual al cliente.

Un concepto muy importante con el que vamos a trabajar es el de verificación de programas. La verificación de programas es un proceso por el que se intenta corroborar que un programa hace lo que se espera de él. Se trata de confirmar que el programa cumple con sus especificaciones.

En esta parte de la unidad nos centraremos en las técnicas de prueba sistemática de programas, verificación y análisis de deficiencias del software.

Las técnicas que veremos son:

- ✓ Pruebas de caja negra o test funcionales.
- ✓ Pruebas de caja blanca o test estructurales.
- ✓ Revisiones o recorridos.
- ✓ Análisis paso a paso del código con un depurador (debugger).

Un pequeño programa que consiste sólo en una única clase, normalmente puede ser probado de una sola vez. En cambio, un programa de mayor tamaño que incluye más cantidad de clases, es posible que por su complejidad, deba ser probado por partes. En Java, el tamaño natural de cada una de esas partes será la clase y será conveniente probar nuestros programas clase por clase. Esto es llamado **Pruebas de Unidad**, y cuando se realizan pruebas de funcionamiento reuniendo todas las partes del programa completo, las pruebas reciben el nombre de **Pruebas de Integración o Pruebas de Sistema**.



Autoevaluación

Rellena los espacios con los conceptos adecuados.

En el proceso de se trata de confirmar que el programa cumple con sus

.

7.1.- La especificación.

El punto de partida para cualquier prueba es la especificación. No podemos considerar una pérdida de tiempo el análisis claro de dicha información. En ocasiones, podría ser necesario volver a entrevistar a un cliente o al futuro usuario del programa.

Para entender la importancia de este elemento, supongamos la siguiente especificación:

- ✓ Es necesario escribir un programa que solicite la entrada al usuario de números a través de cajas de texto. El programa debe calcular y mostrar la suma de los números.



A priori, esta especificación puede parecer simple y clara. Nada más lejos de la realidad, ya que existen lagunas como:

1. ¿Los números son enteros o coma flotante?
2. ¿Cuál es el rango y precisión de dichos números?
3. ¿Pueden incluirse números negativos?

Si estas preguntas no son aclaradas antes de que el programador comience su trabajo, pueden producirse dificultades en el desarrollo del programa. Por tanto, será parte del proceso de programación el estudio de la especificación para descubrir cualesquiera omisiones o confusión y para conseguir una especificación totalmente clara.

A continuación se muestra una versión mucho más clara de la especificación anterior:

- ✓ Es necesario escribir un programa que solicite la entrada al usuario de una serie de números enteros a través de una caja de texto. Los enteros están en el rango 0 a 10.000. El programa calcula y muestra la suma de los números.

Como puedes apreciar, esta especificación es más precisa al establecer el rango permitido de entrada de valores, por ejemplo.

Una vez valorada la importancia de una correcta especificación, el programador puede comenzar el proceso de pruebas del software a través de las herramientas que se detallarán en los siguientes epígrafes de la unidad.

7.2.- Pruebas exhaustivas.

¿He de probar todas las posibilidades? Es una pregunta que puede surgir cuando comenzamos a programar. Podríamos pensar en que la realización de una prueba requeriría probar nuestro software con todos los posibles valores de entrada. Este tipo de prueba se denomina **prueba exhaustiva** y significa que seleccionamos todos los posibles valores de entrada, así como todas sus posibles combinaciones y hacemos que nuestro programa opere con ellos. Pensemos un momento en el amplio rango de posibles valores que pueden tomar los números `int` de Java. ¿Te imaginas lo que puede suponer realizar una prueba de este tipo? ¡Podríamos estar hablando de años!

En resumen, la prueba exhaustiva (aunque sea para un programa pequeño) no es factible. Es importante reconocer que la prueba completa de programas no es posible, por lo que tendremos que adoptar otras técnicas más adecuadas.



Autoevaluación

Durante una prueba exhaustiva hemos de seleccionar una muestra de los posibles valores de entrada del programa y hacer que éste tome diferentes caminos de ejecución para ver los resultados que se obtienen.

Verdadero. ☐ Falso. ☐

7.3.- Pruebas de Caja Negra o Pruebas Funcionales.

Descartadas las pruebas exhaustivas, una técnica más adecuada para la realización de nuestras pruebas es el empleo de las **Pruebas de Caja Negra**. Este tipo de pruebas se basa en utilizar unos datos de entrada que pueden ser representativos de todos los posibles datos de entrada. Con ellos, se pone en funcionamiento el programa y se analiza qué ocurre.



Se llaman de Caja Negra porque no se utiliza ningún conocimiento del funcionamiento interno del programa como parte de la prueba, **sólo se consideran las entradas y las salidas**. El programa se piensa como si fuera una caja negra que recibe datos de entrada y ofrece unas salidas determinadas. También reciben el nombre de Pruebas Funcionales porque utiliza solamente el conocimiento de la función del programa (no cómo trabaja internamente).

Inicialmente, se anotan los datos de prueba y el resultado previsto, antes de ejecutar la prueba. Es a lo que se denomina especificación o planificación de la prueba. Posteriormente, se pone el programa en funcionamiento, se introducen los datos y se examinan las salidas para ver si existen diferencias entre el resultado previsto y el resultado obtenido. Los datos de prueba deberán también comprobar si las excepciones son manejadas por el programa, de acuerdo a su especificación. Es decir, tendremos que poner en aprietos a nuestro programa a través de unos datos de entrada determinados, para comprobar si reacciona correctamente a las especificaciones que deba cumplir.

7.4.- Pruebas de Caja Blanca o Pruebas Estructurales.

Este otro tipo de pruebas se basa en el **conocimiento del funcionamiento interno del programa**, la estructura del mismo para seleccionar los datos de prueba. A lo largo de las pruebas de caja blanca cada declaración que forma parte del programa es ejecutada en algún momento. Por lo que cada secuencia de instrucciones o camino por el que pueda fluir el programa es ejecutada en alguna ocasión durante dicha prueba. Considerándose caminos nulos, sentencias condicionales simples o compuestas, bucles, etc. La prueba deberá incluir cualquier camino del programa que pueda generar una excepción.



Este tipo de pruebas está muy ligado al código fuente del programa, de tal forma, que la persona que realiza las pruebas debe escoger distintos valores de entrada para chequear cada uno de los posibles caminos de ejecución que existen en el programa y verificar que los resultados de cada uno de ellos son adecuados.



Autoevaluación

Rellano los espacios con los conceptos adecuados.

Si nos encontramos realizando pruebas y estamos examinando el funcionamiento de un bucle **while**, estaremos realizando Pruebas de Caja .

7.5.- Otras pruebas.

Además de las típicas pruebas de Caja Blanca y Negra, existen otras técnicas de prueba que detallamos a continuación:

a. Revisiones, inspecciones o recorridos:

Esta técnica no hace uso del computador para intentar erradicar errores en el código. En un recorrido se estudia el listado del programa (junto con la especificación) para intentar hacer visibles los posibles errores. Es recomendable que quien lleve a cabo el recorrido no sea la misma persona que desarrolló el código a revisar.

Si has programado alguna vez, quizá te suene esta situación:

Llevas más de una hora buscando un error en el código de tu programa, ya que éste te está generando errores de compilación o ejecución. Miras, vuelves a mirar, revisas, repasas, cambias alguna variable o sentencia, ...-¡Nada, no encuentro el fallo! -De pronto, un colega relee tu código mientras se toma un café y oyes detrás de ti una voz que dice: -¡Anda, te falta un paréntesis ahí!



Citas para pensar

Sabiduría popular: "Cuatro ojos ven más que dos".

Para llevar a cabo este tipo de prueba necesitaremos la especificación del programa y el código fuente en papel. Las acciones que suelen realizarse son: inspeccionar la inicialización de variables, realización de llamadas a métodos correctas, definición adecuada de cabeceras de métodos, parámetros correctos, etc. Posteriormente, se ha de llevar a cabo una revisión de la lógica del programa simulando ejecutar los métodos como si nosotros mismos fuéramos el computador. Las pruebas de revisión o recorrido no comprueban el estilo del código, sino posibles deficiencias que pueden provocar errores. Está demostrado que los recorridos son una vía adecuada para encontrar errores.



a. Análisis paso a paso:

Existen algunos entornos de desarrollo para Java que incorporan depuración paso a paso del código. Esto permite al programador ejecutar una a una cada instrucción de su programa. De este modo, es posible ver qué camino de ejecución se está tomando, los valores de las distintas variables, etc. Este análisis paso a paso está estructurado y automatizado, permitiéndonos comprobar si el flujo del programa es el correcto, si los valores de las variables varían en función de la ejecución del programa, etc.

Esta técnica suele confirmar o desmentir la existencia de un determinado fallo, aunque está más cerca de la depuración de programas que de la propia prueba de los mismos.

7.6.- Realización de Pruebas Unitarias con JUnit.

Al comienzo de esta parte de la unidad dedicada a las pruebas y depuración de software, hablamos de un tipo de pruebas que se aplican a cada una de las clases que hayamos definido. Eran las Pruebas Unitarias.

Es cierto que cualquier código que escribamos debería de ser probado antes de dar por finalizada la implementación de una clase, ya que si no no estaríamos totalmente seguros de su correcto funcionamiento. Para verificar que el código no contiene errores de programación y que además realiza adecuadamente lo que nosotros esperábamos de él, se realizan una serie de test que lo corroboren.

¿Quién no ha ido escribiendo mensajes de salida de texto a lo largo de su código? Esta es una técnica muy utilizada para controlar el valor de ciertas variables y para la detección de posibles errores, lo que confirma que todos realizamos pruebas a nuestros programas de alguna u otra manera.

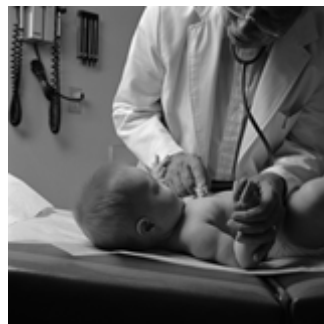
Pero, ¿Existen formas más avanzadas, eficientes y seguras de probar el código? Sí. Podemos utilizar para ello **JUnit**. A través de **JUnit** tendremos la posibilidad de realizar test a nuestras clases de forma sencilla, rápida y elegante, así como validar los resultados que nos ofrecen.

¿Y qué es **JUnit**? Es un conjunto de bibliotecas que se utilizan en programación para hacer Pruebas Unitarias a aplicaciones Java. El conjunto de clases (framework) incluidas en **JUnit** permite controlar la ejecución de las clases Java, de tal forma que podremos evaluar si el funcionamiento de cada uno de los métodos de la clase realiza su trabajo como debería. **JUnit** se emplea para indicar si la clase en cuestión ha pasado los test de prueba, en otro caso se devolverá una notificación de fallo en el método correspondiente.

Actualmente, el entorno de desarrollo NetBeans cuenta con plug-ins que permiten la generación automática de plantillas de pruebas de clases Java.

¿Por qué utilizar **JUnit**? A continuación te damos algunas razones de peso por la que este framework está muy extendido entre la comunidad de programadores Java:

- ✓ A través de los test de JUnit se incrementa la calidad y velocidad de generación de código.
- ✓ La escritura de test es sencilla.
- ✓ Los test JUnit chequean sus propios resultados y proporcionan información de retorno al instante.
- ✓ Los tests JUnit incrementan la estabilidad del software.
- ✓ Los test JUnit se escriben en Java.
- ✓ JUnit es gratuito.



Para saber más

El IDE NetBeans 7.0. ya incorpora **JUnit** desde la instalación, por lo que podemos beneficiarnos de esta herramienta de testeo sólo con utilizarla desde el IDE. Si quieres conocer cómo realizar los primeros pasos con ella y ver ejemplos de utilización, te recomendamos los siguientes enlaces:

[Primeros pasos con JUnit.](#)

[Creación de tests en NetBeans con JUnit.](#)

[Introducción y ejemplos con JUnit.](#)

Tutorial Java en Español - Capitu...



[Resumen textual alternativo](#)



Autoevaluación

En versiones anteriores a NetBeans 7.0., para poder trabajar con el FrameWork **JUnit** era necesario incorporarlo manualmente al IDE.

Verdadero. ☐ Falso. ☐

8.- Depuración de programas.

Caso práctico

Ada y **Juan** ya conocen las capacidades del depurador que incorpora el entorno NetBeans y van a enseñar a **María** las ventajas de utilizarlo.

-Puedes depurar tus programas haciendo dos cosas: creando Breakpoints o haciendo ejecuciones paso a paso -Le comenta **Juan**.

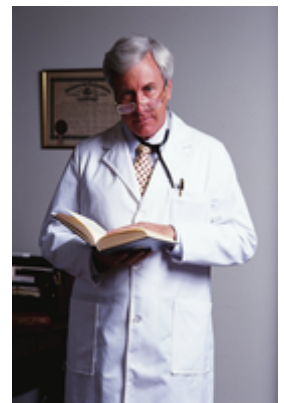


María, que estaba codificando un nuevo método, se detiene un momento y pregunta: -Entonces, ¿cuando el programa llega al Breakpoint podré saber qué valor tiene una variable determinada?

-Efectivamente **María**, y podrás saber el valor de aquellas que tú decidas. De este modo a través de los puntos de ruptura y de las ejecuciones paso a paso podrás descubrir dónde puede haber errores en tus programas. Conocer bien las herramientas que el depurador nos ofrece es algo que puede ahorrarnos mucho trabajo -Aporta **Ada**.

La Depuración de programas es el proceso por el cual se identifican y corrigen errores de programación. Generalmente, en el argot de programación se utiliza la palabra debugging, que significa localización y eliminación de bichos (bugs) o errores de programa. A través de este proceso se descubren los errores y se identifica qué zonas del programa los producen. Hay tres etapas por las que un programa pasa cuando éste es desarrollado y que pueden generar errores:

- ✓ **Compilación:** Una vez que hemos terminado de afinar un programa, solemos pasar generalmente cierto tiempo eliminando errores de compilación. El compilador de Java mostrará una serie de chequeos en el código, sacando a la luz errores que pueden no apreciarse a simple vista. Una vez que el programa es liberado de los errores de compilación, obtendremos de él algunos resultados visibles pero quizá no haga aún lo que queremos.
- ✓ **Enlazado:** Todos los programas hacen uso de librerías de métodos y otros utilizan métodos generados por los propios programadores. Un método es enlazado (linked) sólo cuando éste es llamado, durante el proceso de ejecución. Pero cuando el programa es compilado, se realizan comprobaciones para saber si los métodos llamados existen y sus parámetros son correctos en número y tipo. Así que los errores de enlazado son detectados durante la etapa de compilación.
- ✓ **Ejecución:** Cuando el programa entra en ejecución, es muy frecuente que éste no funcione como se esperaba. De hecho, es normal que el programa falle. Algunos errores serán detectados automáticamente y el programador será informado, como acceder a una parte de un array que no existe (error de índices) por ejemplo. Otros son más sutiles y dan lugar simplemente a comportamientos no esperados, debido a la existencia de errores ocultos (bugs) en el programa. De ahí los términos bug y debugging. El problema de la depuración es que los síntomas de un posible error son generalmente poco claros, hay que recurrir a una labor de investigación para encontrar la causa.



La depuración de programas es algo así como ser doctor: existe un síntoma, hemos de encontrar la causa y entonces determinar el problema. El trabajo de eliminación de errores puede ser interesante. La depuración y la prueba suelen requerir una cantidad de tiempo considerable en comparación con el tiempo dedicado a la primera codificación del programa. Pero no te preocupes, es normal emplear más tiempo en este proceso.

¿Y cómo llevamos a cabo la depuración de nuestros programas?, pues a través del debugger o depurador del sistema de desarrollo Java que estemos utilizando. Este depurador será una herramienta que nos ayudará a eliminar posibles errores de nuestro programa. Podremos utilizar depuradores simples, como el **jdb propio de Java** basado en línea de comandos. O bien, utilizar el **depurador existente en nuestro IDE** (en nuestro caso NetBeans). Este último tipo de depuradores muestra los siguientes elementos en pantalla: El programa en funcionamiento, el código fuente del programa y los nombres y valores actuales de las variables que se seleccionen.

¿Qué elementos podemos utilizar en el depurador? Existen al menos dos elementos fundamentales que podemos utilizar en nuestro debugger o depurador, son los siguientes:

- ✔ **Breakpoints o puntos de ruptura:** Estos puntos pueden ser determinados por el propio programador a lo largo del código fuente de su aplicación. Un Breakpoint es un lugar en el programa en el que la ejecución se detiene. Estos puntos se insertan en una determinada línea del código, entonces el programa se pone en funcionamiento y cuando el flujo de ejecución llega hasta él, la ejecución queda congelada y un puntero indica el lugar en el que la ejecución se ha detenido. El depurador muestra los valores de las variables tal y como están en ese momento de la ejecución. Cualquier discrepancia entre el valor actual y el valor que deberían tener supone una importante información para el proceso de depuración.
- ✔ **Ejecución paso a paso:** El depurador también nos permite ejecutar un programa paso a paso, es decir, línea por línea. A través de esta herramienta podremos seguir el progreso de ejecución de nuestra aplicación y supervisar su funcionamiento. Cuando la ejecución no es la esperada quizá estemos cerca de localizar un error o bug. En ocasiones, si utilizamos métodos procedentes de la biblioteca estándar no necesitaremos hacer un recorrido paso a paso por el interior de éstos métodos, ya que es seguro que no contendrán errores internos y podremos ahorrar tiempo no entrando en su interior paso a paso. El debugger ofrece la posibilidad de entrar o no en dicho métodos.

Debes conocer

Para completar tus conocimientos sobre la depuración de programas, te proponemos los siguientes enlaces en los que podrás encontrar cómo se llevan a cabo las tareas básicas de depuración a través del IDE NetBeans.

[Depuración básica en NetBeans.](#)

[Uso básico del depurador en NetBeans.](#)

Para saber más

Si deseas conocer algo más sobre depuración de programas, pero a un nivel algo más avanzado, puedes ver el siguiente vídeo.

Debugging avanzado en NetBeans.

Debugger Java de Netbeans



[Resumen textual alternativo](#)

9.- Documentación del código.

Caso práctico

Ada está mostrando a **Juan** la documentación sobre una serie de métodos estándar que van a necesitar para completar el desarrollo de una parte de la aplicación. Esta documentación tiene un formato estructurado y puede accederse a ella a través del navegador web.

-¡Qué útil y fácil está siendo el acceso a esta documentación! La verdad es que los que la han generado se lo han currado bastante. ¿Generar esta documentación se llevará mucho tiempo, no Ada? -Pregunta **Juan** mientras recoge de la impresora la documentación impresa.

Ada prepara rápidamente una clase en blanco y comienza a incorporarle una serie de comentarios: -Verás Juan, documentar el código es vital y si incorporas a tu código fuente unos comentarios en el formato que te voy a mostrar, la documentación puede ser generada automáticamente a través de la herramienta **Javadoc**. Observa... -Responde **Ada**.



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.

```
1  /**
2   * Returns the index of the first occurrence of the specified element in
3   * this vector, starting from the startElementIndex, to return -1 if
4   * the element is not found.
5   *
6   * @param element to search for
7   * @param start index to start searching from
8   * @return the index of the first occurrence of the element or
9   *         -1 if no such element is found in the vector.
10    * @throws NoSuchElementException if the element is not found
11    * @since VectorSupportVersion1.0
12    */
13    int indexOfElement(E element, int startElementIndex);
14 }
```

Citas para pensar

R. Caron: "do not document bad code - rewrite it".

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.

9.1.- 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.

	@author	@todo	@author	@deprecated	@description	@param	@return	@throws
Descripción	✓	✓	✓	✓	✓	✓	✓	✓
Proyecto	✓	✓	✓	✓	✓	✓	✓	✓
Clases e interfaces	✓	✓	✓	✓	✓	✓	✓	✓
Atributos	✓	✓	✓	✓	✓	✓	✓	✓
Constructores y métodos	✓	✓	✓	✓	✓	✓	✓	✓

9.2.- 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 {@link me
```

- ✓ `@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, es
 * <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 or
* @see <a href="http://www.w3.org/WAI/">Web Accessibility Initiative</a>
* @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 etiquetas es aplicable a constructores y métodos, describiendo las posibles excepciones del constructor/método.



Autoevaluación

¿Qué etiqueta podría omitirse en un método que devuelve void?

- ☐ @param.
- ☐ @throws.
- ☐ @return.



9.3.- 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.
<code>@autor</code>	En clases e interfaces. Se pueden poner varios. Es mejor ponerlas en orden cronológico.
<code>@version</code>	En clases e interfaces.
<code>@param</code>	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.
<code>@return</code>	En métodos.
<code>@exception</code>	En constructores y métodos. Mejor en el mismo orden en el que se han declarado, o en orden alfabético.
<code>@throws</code>	Es equivalente a <code>@exception</code> .
<code>@see</code>	Podemos poner varios. Comenzaremos por los más generales y después los más específicos.
<code>@deprecated</code>	



Para saber más






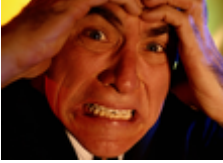


Si quieres conocer cómo obtener a través de **Javadoc** la documentación de tus aplicaciones, sigue los siguientes enlaces:

[Documentación con Javadoc.](#)

[Documentación de clases y métodos con Javadoc.](#)

Anexo.- Licencias de recursos.

Licencias de recursos utilizado

Recurso (1)	Datos del recurso (1)
	<p>Autoría: Stockbyte.</p> <p>Licencia: Uso educativo no comercial para plataformas públicas de Formación Pr distancia.</p> <p>Procedencia: CD-DVD Num. V43</p>
	<p>Autoría: Stockbyte.</p> <p>Licencia: Uso educativo no comercial para plataformas públicas de Formación Pr distancia.</p> <p>Procedencia: CD-DVD Num. CD165 stk22466btm</p>
	<p>Autoría: Marcin Wichary</p> <p>Licencia: Creative Commons Attribution 2.0 Generic</p> <p>Procedencia: http://commons.wikimedia.org/wiki/File:IBM_1620_Memory_address_register_display_s</p>
	<p>Autoría: HuBoro</p> <p>Licencia: Dominio público</p> <p>Procedencia: http://commons.wikimedia.org/wiki/File:Ambox_deletion_recycle.svg</p>
	<p>Autoría: Abrev</p> <p>Licencia: Creative Commons Attribution-Share Alike 3.0 Unported</p> <p>Procedencia: http://commons.wikimedia.org/wiki/File:Textilkennzeichnungab.jpg</p>
	<p>Autoría: Stockbyte.</p> <p>Licencia: Uso educativo no comercial para plataformas públicas de Formación Pr distancia.</p> <p>Procedencia: CD-DVD Num. V43</p>
	<p>Autoría: Beatrice Murch</p> <p>Licencia: Creative Commons Attribution-Share Alike 2.0 Generic</p> <p>Procedencia: http://commons.wikimedia.org/wiki/File:Meal_worm_in_venus_fly_trap.jpg</p>
	<p>Autoría: Stockbyte.</p> <p>Licencia: Uso educativo no comercial para plataformas públicas de Formación Pr distancia.</p> <p>Procedencia: CD-DVD Num. V43</p>
	<p>Autoría: Stockbyte.</p>



Licencia: Uso educativo no comercial para plataformas públicas de Formación Pr
distancia.
Procedencia: CD-DVD Num. V07



Autoría: Yan Shuangchun
Licencia: GPL
Procedencia: http://commons.wikimedia.org/wiki/File:Torchlight_core.png



Autoría: Dr. Timo Mappes
Licencia: CC-BY-SA-3.0; CC-BY-SA-3.0-DE; BILD-GFDL-NEU;
Procedencia: http://commons.wikimedia.org/wiki/File:Leitz_117298_frei.jpg



Autoría: Stockbyte.
Licencia: Uso educativo no comercial para plataformas públicas de Formación Pr
distancia.
Procedencia: CD-DVD Num. V43



Autoría: Stockbyte.
Licencia: Uso educativo no comercial para plataformas públicas de Formación Pr
distancia.
Procedencia: CD-DVD Num. V07



Autoría: Lensim
Licencia: Creative Commons Attribution-Share Alike 3.0 Unported
Procedencia: http://commons.wikimedia.org/wiki/File:Notebook_with_Post-its.jpg