

1º TÉCNICO SUPERIOR  
DESARROLLO APLICACIONES WEB

# PROGRAMACIÓN

## USO DE ESTRUCTURAS DE CONTROL

*Tema 3*

BEATRIZ LÓPEZ PALACIO

**ÍNDICE**

<b>1. INTRODUCCIÓN .....</b>	<b>2</b>
<b>2. SENTENCIAS Y BLOQUES .....</b>	<b>4</b>
<b>3. ESTRUCTURAS DE SELECCIÓN .....</b>	<b>6</b>
3.1.   ESTRUCTURA IF / IF-ELSE .....	7
3.2.   ESTRUCTURA SWITCH.....	8
<b>4. ESTRUCTURAS DE REPETICIÓN.....</b>	<b>11</b>
4.1.   ESTRUCTURA FOR.....	12
4.2.   ESTRUCTURA FOR / IN.....	13
4.3.   ESTRUCTURA WHILE .....	14
4.4.   ESTRUCTURA DO-WHILE .....	15
<b>5. ESTRUCTURAS DE SALTO .....</b>	<b>17</b>
5.1.   SENTENCIAS BREAK Y CONTINUE.....	17
5.2.   ETIQUETAS .....	19
5.3.   SENTENCIA RETURN.....	21
<b>6. EXCEPCIONES.....</b>	<b>23</b>
6.1.   CAPTURAR UNA EXCEPCIÓN .....	25
6.2.   MANEJO DE EXCEPCIONES.....	26
6.3.   DELEGACIÓN DE EXCEPCIONES CON THROWS .....	28

## 1. INTRODUCCIÓN

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 otras. 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.

## 2. SENTENCIAS Y BLOQUES

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.
- **¿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 en nuestro programa.

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.

### 3. 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.

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 estas.

### 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 simple.

##### Sintaxis:

```
if (expresión-lógica)
    sentencial;
```

##### Sintaxis:

```
if (expresión-lógica)
{
    sentencial;
    sentencia2;
    ...;
    sentenciaN;
}
```

#### 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.

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**.

### 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

**Sintaxis:**

```

switch (expresión) {
    case valor1:
        sentencial_1;
        sentencial_2;
        ...
        break;
        ...
        ...
        ...
    case valorN:
        sentencial_N_1;
        sentencial_N_2;
        ...
        break;
    default:
        sentencias-default;
}

```

**Condiciones:**

- ✓ 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:**

- Las diferentes alternativas de esta estructura estarán precedidas de la cláusula **case** que se ejecutará cuando el valor asociado al **case** coincide 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.

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.

## 4. ESTRUCTURAS DE REPETICIÓN

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:

1. Bucle **for** (repite para)
2. Bucle **for/in** (repite para cada)
3. Bucle **While** (repite mientras)
4. 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?

#### 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.

Recomendaciones:

- 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**

**Sintaxis:**

```
for (inicialización; condición; iteración)
    sentencia;
```

(estructura **for** con una única sentencia)

**Sintaxis:**

```
for (inicialización; condición; iteración)
{
    sentencia1;
    sentencia2;
    ...
    sentenciaN;
}
```

(estructura **for** con un bloque de sentencias)

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.

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.

## 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.

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	
Sintaxis:	Sintaxis:
<b>Sintaxis:</b> <pre>while (condición)     sentencia;</pre>	<b>Sintaxis:</b> <pre>while (condición) {     sentencial;     ...     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.

## 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	
<b>Sintaxis:</b>  do sentencia; while (condición);	<b>Sintaxis:</b>  do { sentencial; ... sentenciaN; } while (condición);

### Funcionamiento:

El cuerpo del bucle se ejecuta la primera vez, a continuación se evaluará la condición y, si ésta es falsa, el cuerpo del 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.

## 5. ESTRUCTURAS DE SALTO

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 sernos 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.

### 5.1. SENTENCIAS BREAK Y CONTINUE

Se trata de dos instrucciones que permiten modificar el comportamiento de otras estructuras o sentencias de control, simplemente por el hecho de estar incluidas en algún punto de su secuencia de instrucciones.

La **sentencia break** incidirá sobre las estructuras de control **switch**, **while**, **for** y **do-while** del siguiente modo:

- Si aparece una sentencia break dentro de la secuencia de instrucciones de cualquiera de las estructuras mencionadas anteriormente, dicha estructura terminará inmediatamente.
- Si aparece una sentencia break dentro de un bucle anidado sólo finalizará la sentencia de iteración más interna, el resto se ejecuta de forma normal.

### Ejemplo:

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Dentro del bucle");  
    break;  
    System.out.println("Nunca lo escribirá");  
}  
System.out.println("Tras el bucle");
```

Es decir, que **break** sirve para romper el flujo de control de un bucle, aunque no se haya cumplido la condición del bucle. Si colocamos un **break** dentro del código de un bucle, cuando se alcance el **break**, automáticamente se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él.

La **sentencia continue** incidirá sobre las sentencias o estructuras de control **while**, **for** y **do-while** del siguiente modo:

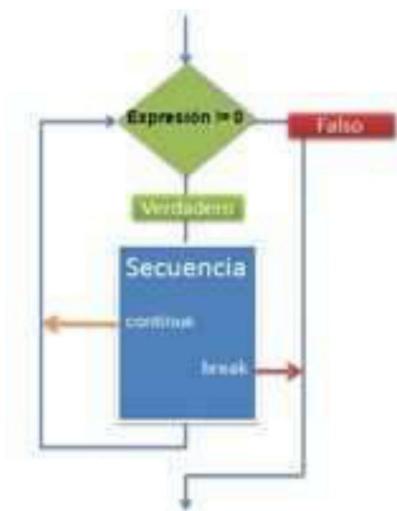
- Si aparece una sentencia **continue** dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia dará por terminada la iteración actual y se ejecuta una nueva iteración, evaluando de nuevo la expresión condicional del bucle.
- Si aparece en el interior de un bucle anidado solo afectará a la sentencia de iteración más interna, el resto se ejecutaría de forma normal.

### Ejemplo:

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Dentro del bucle");  
    continue;  
    System.out.println("Nunca lo escribirá");  
}
```

Es decir, la sentencia **continue** forzará a que se ejecute la siguiente iteración del bucle, sin tener en cuenta las instrucciones que pudiera haber después del **continue**, y hasta el final del código del bucle.

Para clarificar algo más el funcionamiento de ambas sentencias de salto, te ofrecemos a continuación un diagrama representativo.



## 5.2. ETIQUETAS

Ya lo indicábamos al comienzo del apartado dedicado a las estructuras de salto, los saltos incondicionales y en especial, saltos a una etiqueta son totalmente desaconsejables. No obstante, Java permite asociar etiquetas cuando se va a realizar un salto. De este modo puede conseguirse algo más de legibilidad en el código.

Las estructuras de salto **break** y **continue**, pueden tener asociadas etiquetas. Es a lo

que se llama un **break etiquetado** o un **continue etiquetado**. Pero sólo se recomienda su uso cuando se hace necesario salir de bucles anidados hacia diferentes niveles.

¿Y cómo se crea un salto a una etiqueta? En primer lugar, crearemos la etiqueta mediante un **identificador seguido de dos puntos (:)** . A continuación, se escriben las sentencias Java asociadas a dicha etiqueta encerradas entre llaves. Por así decirlo, la creación de una etiqueta es como fijar un punto de salto en el programa para poder saltar a él desde otro lugar de dicho programa.

¿Cómo se lleva a cabo el salto? Es sencillo, en el lugar donde vayamos a colocar la sentencia break o continue, añadiremos detrás el identificador de la etiqueta. Con ello, conseguiremos que el salto se realice a un lugar determinado.

La sintaxis será:

```
break <etiqueta>
```

Quizá a aquellos y aquellas que habéis programado en HTML os suene esta herramienta, ya que tiene cierta similitud con las anclas que pueden crearse en el interior de una página web, a las que nos llevará el hiperenlace o link que hayamos asociado.

También para aquellos y aquellas que habéis creado alguna vez archivos por lotes o archivos batch bajo MSDOS es probable que también os resulte familiar el uso de etiquetas, pues la sentencia **GOTO** que se utilizaba en este tipo de archivos, hacía saltar el flujo del programa al lugar donde se ubicaba la etiqueta que se indicara en dicha sentencia.

A continuación, te ofrecemos un **ejemplo** de declaración y uso de etiquetas en un bucle. Como podrás apreciar, las sentencias asociadas a cada etiqueta están encerradas entre llaves para delimitar así su ámbito de acción.

```
bucle1:  
for (int i = 0; i < 10; i++) {  
    System.out.println("Bucle i. i = "+i);  
    for (int j = 0; j < 10; j++) {  
        System.out.println("Bucle j. j = "+j);  
        for (int k = 0; k < 10; k++) {  
            System.out.println("Bucle k. k = "+k);  
            break bucle1;  
        }  
    }  
}
```

fueras:

```
for (int i=0; i<5; i++) {  
    for (int j=0; j<5; j++) {  
        System.out.println("Dentro");  
        continue fuera;  
    } // fin del bucle anidado  
    System.out.println("fuera"); // nunca lo escribe  
}  
System.out.println("Fuera del bucle");
```

### 5.3. SENTENCIA RETURN

Ya sabemos cómo 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? Si 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**.

Ejemplo sin valor de retorno:

```
void enteroPositivo(int a){  
    if (a<=0)  
        return;  
    else {  
        //Código que realiza una tarea intensiva para el procesador  
        //...  
    }  
}
```

## 6. 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 quizás 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:

- 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.
- 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.

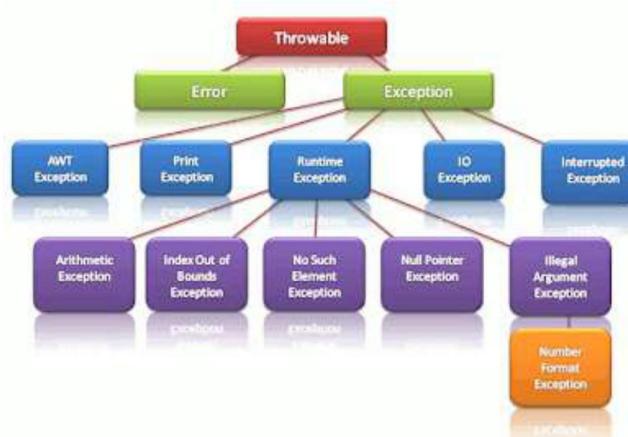
En Java, las excepciones están representadas por clases. El paquete **java.lang.Exception** y sus subpaquetes contienen todos los tipos de excepciones. Todas las excepciones derivarán de la clase **Throwable**, existiendo clases más específicas. Por debajo de la clase **Throwable** existen las clases **Error** y **Exception**. Los errores son 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 se observa 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 `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`.

## 6.2. 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.

Ejemplo:

```
public class Programa {  
  
    public static void main (String [ ] args) {  
  
        try{  
  
            System.out.println("Intentamos ejecutar el bloque de instrucciones:");  
  
            System.out.println("Instrucción 1.");  
  
            System.out.println("Instrucción 2.");  
  
            System.out.println("Instrucción 3, etc.");  
  
        }  
  
        catch (Exception e) { System.out.println("Instrucciones a ejecutar cuando se produce un error");}  
  
        finally{ System.out.println("Instrucciones a ejecutar finalmente tanto si se producen errores como si no.");}  
  
    }  
  
}
```

La salida obtenida tras ejecutar el programa anterior es:

```
Intentamos ejecutar el bloque de instrucciones:  
Instrucción 1.  
Instrucción 2.  
Instrucción 3, etc.  
Instrucciones a ejecutar finalmente tanto si se producen errores como si no.
```

### Ejemplo:

```
FileReader lector = null;  
try {  
    lector = new FileReader("archivo.txt");  
    int i=0;  
    while(i != -1){  
        i = lector.read();  
        System.out.println((char) i );  
    }  
} catch (IOException e) {  
    System.out.println("Error");  
} finally {  
    if(lector != null){  
        lector.close();  
    }  
}
```

El código contenido en finally se ejecutará tras terminar el bloque try, haya habido o no excepción, lo que permite liberar los recursos reservados para abrir el archivo.

### **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 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.