

# ESTRUCTURAS DE CONTROL

## 1. INTRODUCCIÓN

Los tipos de estructuras de programación que se emplean para el control del flujo de los datos son los 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 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 recomendables**, es necesario conocerlas.

## 2. SENTENCIAS Y BLOQUES

¿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.

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

Observa 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, bloques y diferentes organizaciones

```
package organizacion_sentencias;

/**
 *
 * Organización de sentencias secuencial
 */
public class Organizacion_sentencias_1 {
    public static void main(String[] args) {
        System.out.println ("Organización secuencial de sentencias");
        int dia=12;
        System.out.println ("El día es: " + dia);
        int mes=11;
        System.out.println ("El mes es: " + mes);
        int anio=2011;
        System.out.println ("El anio es: " + anio);
    }
}
```

**En este primer archivo, las sentencias están colocadas en orden secuencial.**

```
package organizacion_sentencias;

/**
 *
 * Organización de sentencias con declaración previa
 * de variables
 */
public class Organizacion_sentencias_2 {
    public static void main(String[] args) {
        // Zona de declaración de variables
        int dia=10;
        int mes=11;
        int anio=2011;
        System.out.println ("Organización con declaración previa de variables");
        System.out.println ("El día es: " + dia);
        System.out.println ("El mes es: " + mes);
        System.out.println ("El anio es: " + anio);
    }
}
```

**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.**

```

package organizacion_sentencias;

/**
 *
 * Organización de sentencias en zonas diferenciadas
 * según las operaciones que se realicen en el código
 */
public class Organizacion_sentencias_3 {
    public static void main(String[] args) {
        // Zona de declaración de variables
        int dia;
        int mes;
        int anio;
        String fecha;
        //Zona de inicialización o entrada de datos
        dia=10;
        mes=11;
        anio=2011;
        fecha="";
        //Zona de procesamiento
        fecha=dia+"/"+mes+"/"+anio;
        //Zona de salida
        System.out.println ("Organización con zonas diferenciadas en el código");
        System.out.println ("La fecha es: " + fecha);
    }
}

```

**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

### 3. ESTRUCTURAS DE SELECCIÓN

¿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.

Las estructuras de selección se dividen en:

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

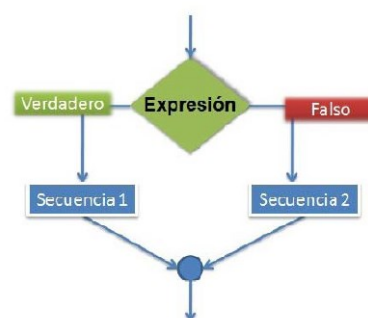
### 3.1 Estructura if y estructura if-else

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.		Estructura if de doble alternativa.	
	<pre>if (expresión-lógica) {     sentencia1;     sentencia2;     ...;     sentenciaN; }</pre>		<pre>if (expresión-lógica) {     sentencia1;     ...;     sentenciaN; } else {     sentencia1;     ...;     sentenciaN; }</pre>
<pre>if (expresión-lógica)     sentencia1;</pre>		<pre>if (expresión-lógica)     sentencia1; else     sentencia2;</pre>	
<p><i>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.</i></p>		<p><i>Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecutará la primera sentencia o el primer bloque de sentencias. Si, por el contrario, la evaluación de la expresión-lógica ofrece un resultado falso, no se ejecutará la primera sentencia o el primer bloque y sí se ejecutará la segunda sentencia o el segundo bloque.</i></p>	

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	
<b>Sintaxis:</b> <pre>switch (expresion) {     case valor1:         sentencial_1;         sentencial_2;         ...         break;     ...     case valorN:         sentenciaN_1;         sentenciaN_2;         ...         break;     default:         sentencias-default; }</pre>	<b>Condiciones:</b> <ul style="list-style-type: none"><li>✓ 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.</li><li>✓ La expresión debe ir entre paréntesis.</li><li>✓ Cada case llevará asociado un valor y se finalizará con dos puntos.</li><li>✓ El bloque de sentencias asociado a la cláusula default puede finalizar con una sentencia de ruptura break o no.</li></ul>

#### Funcionamiento:

- 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

## 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:

- 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?

### 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.

Estructura repetitiva for	
<p>Sintaxis:</p> <pre>for (inicialización; condición; iteración)     sentencia;</pre> <p>(estructura <b>for</b> con una única sentencia)</p>	<p>✓ Donde <b>inicialización</b> es una expresión en la que se inicializa una variable de control, que será la encargada de controlar el final del bucle.</p>
<p>Sintaxis:</p> <pre>for (inicialización; condición; iteración) {     sentencia1;     sentencia2;     ...     sentenciaN; }</pre> <p>(estructura <b>for</b> con un bloque de sentencias)</p>	<p>✓ Donde <b>condición</b> 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 <b>iteración</b> 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>

## 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 haz 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	
<b>Sintaxis:</b>	<b>Sintaxis:</b>
<pre>while (condición) sentencia;</pre>	<pre>while (condición) { sentencia; ... sentenciaN; }</pre>
<b>Funcionamiento:</b> 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 <b>while</b> . 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 dowhile. 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>	<b>Sintaxis:</b>
<pre>do sentencia; while (condición);</pre>	<pre>do { sentencia; ... sentenciaN; } while (condición);</pre>
<b>Funcionamiento:</b> 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 <b>do-while</b> . 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 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.

### 5.1 Sentencia 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.

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.

En la siguiente imagen, puedes apreciar cómo se utilizaría la sentencia break dentro de un bucle for.

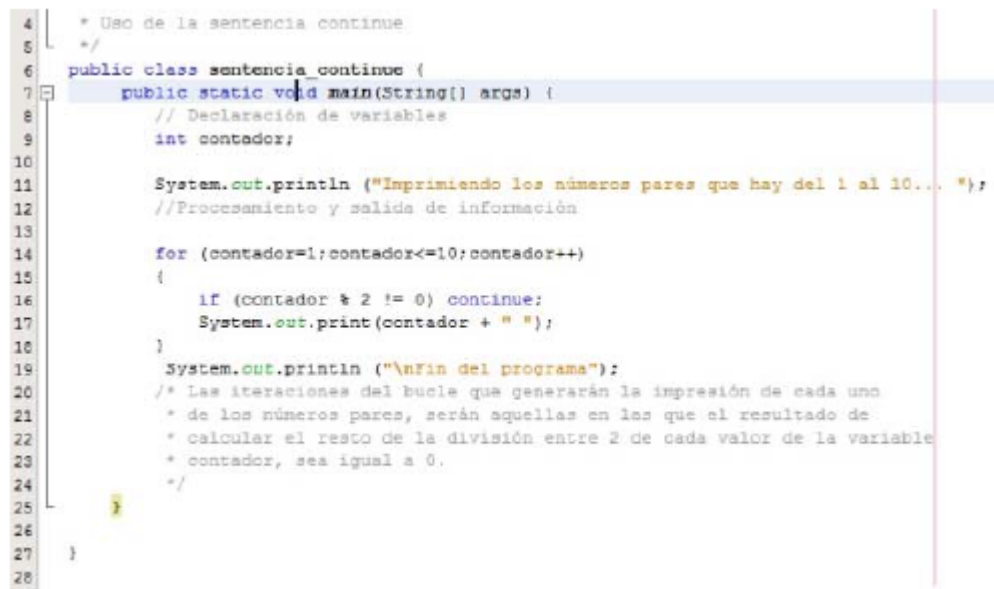
```
6 public class sentencia_break {
7     public static void main(String[] args) {
8         // Declaración de variables
9         int contador;
10
11
12         //Procesamiento y salida de información
13
14         for (contador=1;contador<=10;contador++)
15         {
16             if (contador==7)
17                 break;
18             System.out.println ("Valor: " + contador);
19         }
20         System.out.println ("Fin del programa");
21         /* El bucle sólo se ejecutará en 6 ocasiones, ya que cuando
22          * la variable contador sea igual a 7 encontraremos un break que
23          * romperá el flujo del bucle, transfiriéndonos a la sentencia que
24          * imprime el mensaje de Fin del programa.
25          */
26     }
```

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.

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

En la siguiente imagen, puedes apreciar cómo se utiliza la sentencia **continue** en un bucle **for** para imprimir por pantalla sólo los números pares.



```
4  * Uso de la sentencia continue
5  */
6  public class sentencia_continue {
7      public static void main(String[] args) {
8          // Declaración de variables
9          int contador;
10
11          System.out.println ("Imprimiendo los números pares que hay del 1 al 10... ");
12          //Procesamiento y salida de información
13
14          for (contador=1;contador<=10;contador++)
15          {
16              if (contador % 2 != 0) continue;
17              System.out.print(contador + " ");
18          }
19          System.out.println ("\nfin del programa");
20          /* Las iteraciones del bucle que generarán la impresión de cada uno
21           * de los números pares, serán aquellas en las que el resultado de
22           * calcular el resto de la división entre 2 de cada valor de la variable
23           * contador, sea igual a 0.
24           */
25      }
26  }
27
28  }
```

## 5.2 Etiquetas

Ya lo indicábamos al comienzo del epígrafe 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/as que han programado en HTML les 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 hipervínculo o link que hayamos asociado.

También para aquellos/as que han creado alguna vez archivos por lotes o archivos batch bajo MSDOS es probable que también les 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.

```
1  /**
2   *
3   * Uso de etiquetas en bucle
4   */
5  public class etiquetas {
6      public static void main(String[] args) {
7
8          for (int i=1; i<3; i++) //Creamos cabecera del bucle
9          {
10             bloque_uno: { //Creamos primera etiqueta
11                 bloque_dos:{ //Creamos segunda etiqueta
12                     System.out.println("Iteración: "+i);
13                     if (i==1) break bloque_uno; //Llevamos a cabo el primer salto
14                     if (i==2) break bloque_dos; //Llevamos a cabo el segundo salto
15                 }
16                 System.out.println("después del bloque dos");
17             }
18             System.out.println("después del bloque uno");
19         }
20         System.out.println("Fin del bucle");
21     }
22 }
```

## 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?. 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.