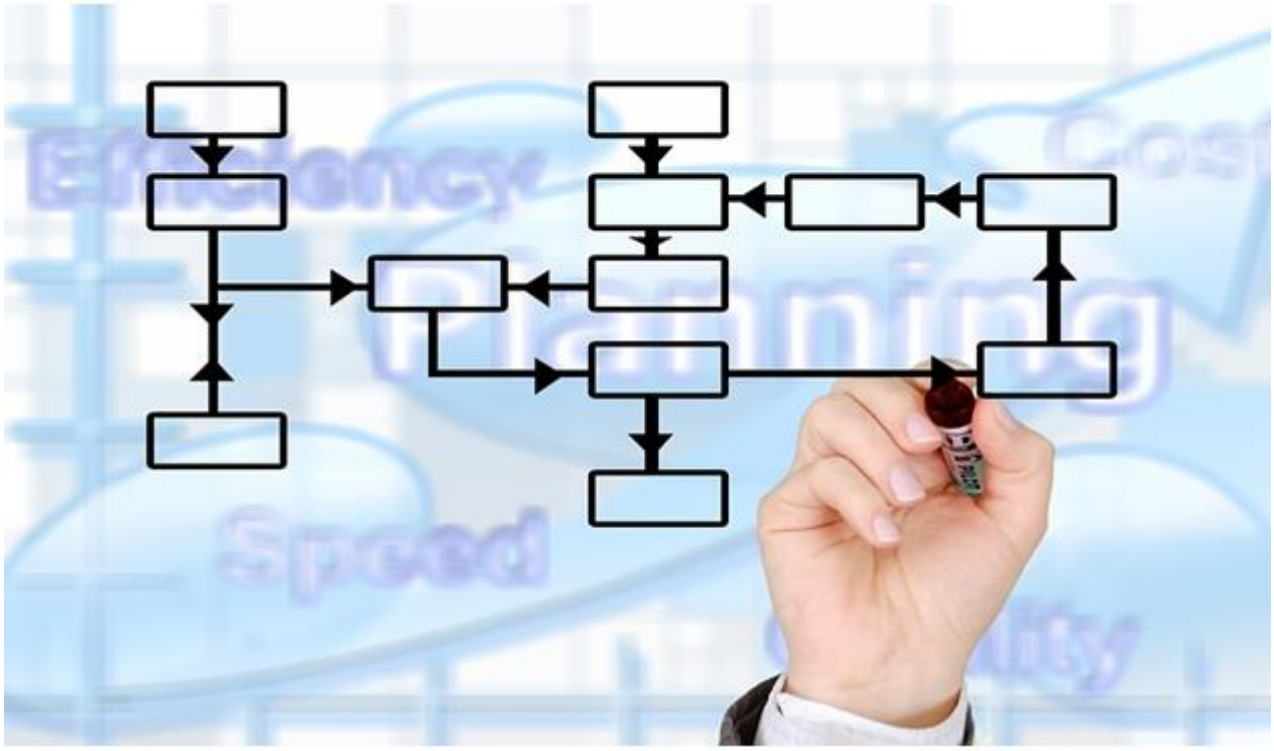
TEMA 3: Uso de estructuras de control

3.A. INTRODUCCIÓN.

3.B. ESTRUCTURAS DE SELECCIÓN.

3.C. ESTRUCUTURAS DE REPETICIÓN

3.D. ESTRUCUTURAS DE SALTO



3.A. 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.

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

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

1. 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.

Bloques de sentencias.	
Bloque de sentencias 1	**Bloque de sentencias 2**

{sentencia1; sentencia2;...; sentencia N;}	{
	sentencia1;
	sentencia2;
	…;
	sentenciaN;
	}

- **¿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.

Debes conocer

Analiza el código de los siguientes 3 programas comparando su código fuente. Verás que los tres obtienen el mismo resultado, pero la organización de las sentencias que los componen es diferente entre ellos:

A) En este primer programa, las sentencias están colocadas en orden secuencial:

```
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);

    }
}
```

B) En este segundo programa, 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 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 dia es: " + dia);
        System.out.println ("El mes es: " + mes);
        System.out.println ("El anio es: " + anio);

    }
}
```

C) En este tercer programa, 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.

```
 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);

    }

}
```

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.B. Estructuras de selección.

1. 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.

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.

1.1. 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:

<table>
<tr><th colspan="4">Estructura if e if-else.</th></tr>
<tr><th colspan="2">Estructura <code>if</code> simple.</th><th colspan="2">Estructura <code>if</code> de doble alternativa.</th></tr>
<tr>
<td>Sintaxis:

<pre>if (expresión-lógica)
sentencia1;</pre></td>
<td>Sintaxis:

<pre>if (expresión-lógica)
{
sentencia1;
sentencia2;
…;
sentenciaN;
}</pre></td>
<td>Sintaxis:

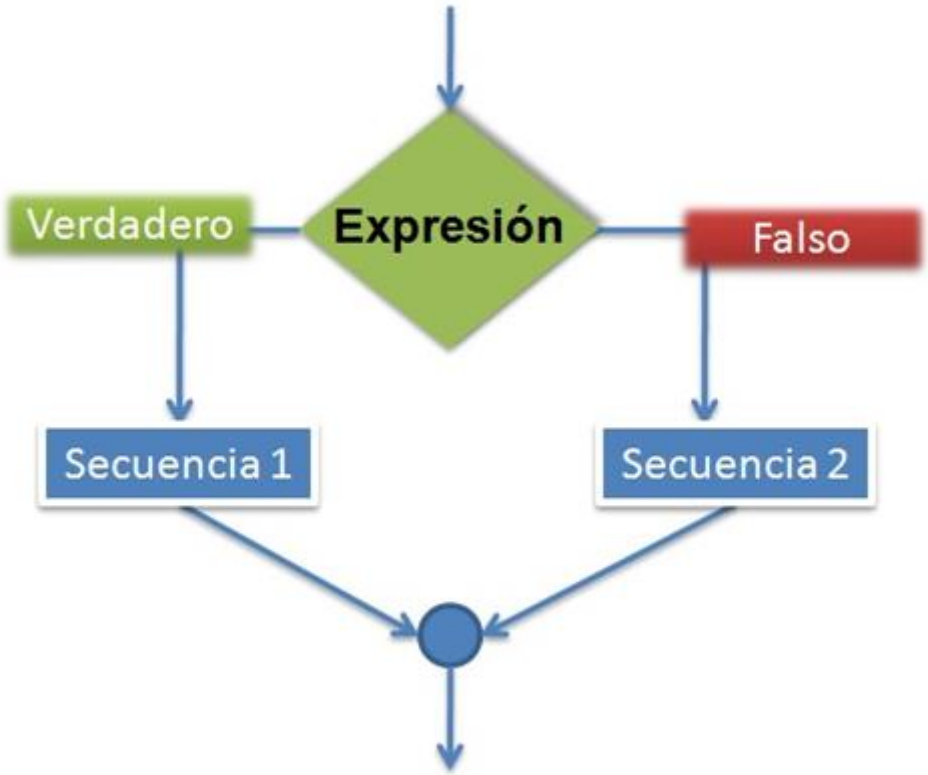
<pre>if (expresión-lógica)
sentencia1;
else
sentencia2;</pre></td>
<td>Sintaxis:

<pre>if (expresión-lógica)
{
sentencia1;
…;
sentenciaN;
}
else
{
sentencia1;
…;
sentenciaN;
}</pre></td>
</tr>
<tr>
<td colspan="2">Funcionamiento:

Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecuta la <code>sentencia1</code> 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.</td>
<td colspan="2">Funcionamiento:

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.</td>
</tr>
</table>

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, unelseestará 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, estudia el siguiente código. 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.

```
package sentencias_condicionales;

/**
 *
 * Ejemplos de utilización de diferentes estructuras
 * condicionales simples, completas y anidamiento de éstas.
 */
public class Sentencias_condicionales {


    /*Vamos a realizar el cálculo de la nota de un examen
     * de tipo test. Para ello, tendremos en cuenta el número
     * total de pregunta, los aciertos y los errores. Dos errores
     * anulan una respuesta correcta.
     *
     * Finalmente, se muestra por pantalla la nota obtenida, así
     * como su calificación no numérica.
     *
     * La obtención de la calificación no numérica se ha realizado
     * combinando varias estructuras condicionales, mostrando expresiones
     * lógicas compuestas, así como anidamiento.
     *
     */
    public static void main(String[] args) {
         // Declaración e inicialización de variables
        int num_aciertos = 12;
        int num_errores = 3;
        int num_preguntas = 20;
        float nota = 0;
```

```
        String calificacion="";

        //Procesamiento de datos
        nota = ((num_aciertos - (num_errores/2))*10)/num_preguntas;

        if (nota < 5)
        {
           calificacion="INSUFICIENTE";
        }
        else
        {
           /* Cada expresión lógica de estos if está compuesta por dos
            * expresiones lógicas combinadas a través del operador Y o AND
            * que se representa con el símbolo &&. De tal manera, que para
            * que la expresión lógica se cumpla (sea verdadera) la variable
            * nota debe satisfacer ambas condiciones simultáneamente
            */
            if (nota >= 5 && nota <6)
               calificacion="SUFICIENTE";
            if (nota >= 6 && nota <7)
               calificacion="BIEN";
            if (nota >= 7 && nota <9)
               calificacion="NOTABLE";
            if (nota >= 9 && nota <=10)
               calificacion="SOBRESALIENTE";
        }

        //Salida de información
        System.out.println ("La nota obtenida es: " + nota);
        System.out.println ("y la calificación obtenida es: " + calificacion);
    }
}
```

Escalera if-else-if. Se van ejecutando una detrás de otra en caso de que ninguna sea verdadera. Si ninguna lo es, se ejecutaría el último else

```
if (condición)
    declaración;
else if (condición)
    declaración;
.
.
    declaración
else
      declaración;
```

1.2. 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.

<table>
<tr><th colspan="2">Estructura <code>switch</code></th></tr>
<tr><td>Sintaxis:

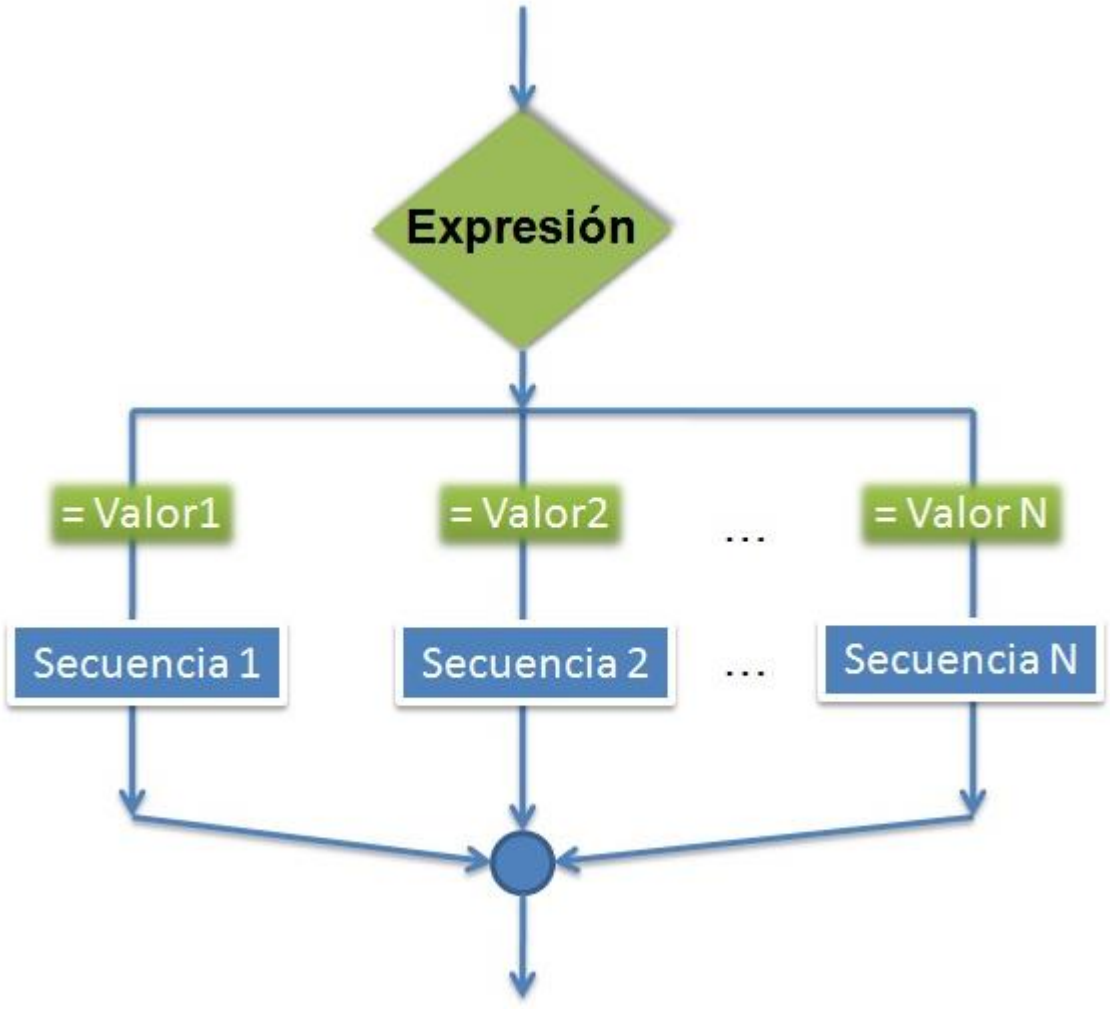
<code>switch (expresion)
{
case valor1:
sentencia1_1;
sentencia1_2;
….
break;
….
….
case valorN:
sentenciaN_1;
sentenciaN_2;
….
break;
default:
sentencias-
default;
}</code></td><td>Condiciones:

• Donde expresión debe ser del tipo <code>char</code>, <code>byte</code>, <code>short</code> o <code>int</code>, y las constantes de cada case deben ser de este tipo o de un tipo compatible.
• La expresión debe ir entre paréntesis.
• Cada <code>case</code> llevará asociado un valor y se finalizará con dos puntos.
• El bloque de sentencias asociado a la cláusula <code>default</code> puede finalizar con una sentencia de ruptura <code>break</code> o no.</td></tr>
<tr><td colspan="2">Funcionamiento:

• Las diferentes alternativas de esta estructura estarán precedidas de la cláusula <code>case</code> que se ejecutará cuando el valor asociado al <code>case</code> coincida con el valor obtenido al evaluar la expresión del <code>switch</code>.
• En las cláusulas <code>case</code>, no pueden indicarse expresiones condicionales, rangos de valores o listas de valores. (otros lenguajes de programación sí lo permiten). Habrá que asociar una cláusula <code>case</code> a cada uno de los valores que deban ser tenidos en cuenta.</td></tr>
</table>

- **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

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

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package sentencias_condicionales;
/**
 *
 * @author Pc
 */
public class condicional_switch {

     /*Vamos a realizar el cálculo de la nota de un examen
     * de tipo test. Para ello, tendremos en cuenta el número
     * total de preguntas, los aciertos y los errores. Dos errores
     * anulan una respuesta correcta.
     *
     * La nota que vamos a obtener será un número entero.
     *
     * Finalmente, se muestra por pantalla la nota obtenida, así
     * como su calificación no numérica.
     *
     * La obtención de la calificación no numérica se ha realizado
     * utilizando la estructura condicional múltiple o switch.
     *
     */
    public static void main(String[] args) {
         // Declaración e inicialización de variables
        int num_aciertos = 17;
        int num_errores = 3;
        int num_preguntas = 20;
```

```
        int nota = 0;
        String calificacion="";

        //Procesamiento de datos
        nota = ((num_aciertos - (num_errores/2))*10)/num_preguntas;

        switch (nota) {
            case 5: calificacion="SUFICIENTE";
                    break;
            case 6: calificacion="BIEN";
                    break;
            case 7: calificacion="NOTABLE";
                    break;
            case 8: calificacion="NOTABLE";
                    break;
            case 9: calificacion="SOBRESALIENTE";
                    break;
            case 10: calificacion="SOBRESALIENTE";
                    break;
            default: calificacion="INSUFICIENTE";
        }
        //Salida de información
        System.out.println ("La nota obtenida es: " + nota);
        System.out.println ("y la calificación obtenida es: " + calificacion);
    }
}
```

3.C. ESTRUCUTURAS 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?

Recomendación

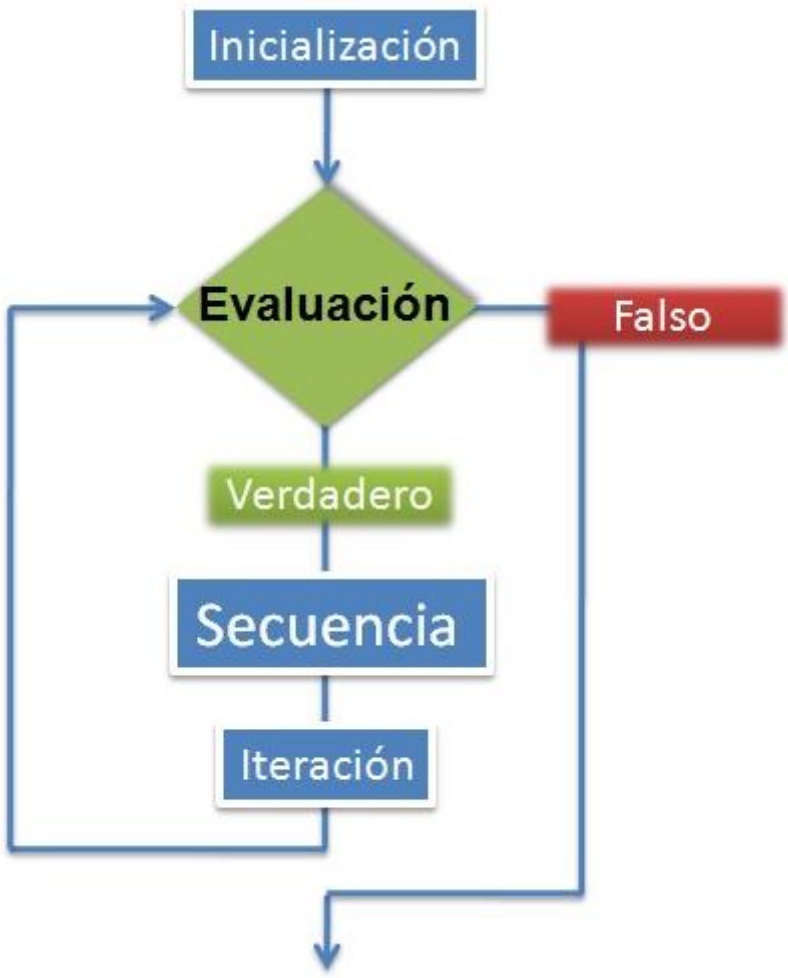
Estudia cada tipo de estructura repetitiva, conoce su funcionamiento y podrás llegar a la conclusión de que algunos de estos bucles son equivalentes entre sí. Un mismo problema, podrá ser resuelto empleando diferentes tipos de bucles y obtener los mismos resultados.

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

1.1. 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 realizase 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:

<table>
<tr><th colspan="2">Estructura repetitiva for</th></tr>
<tr><td>Sintaxis:

for (inicialización; condición; iteración)

sentencia;

(estructura for con una única sentencia)</td><td rowspan="2">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.</td></tr>
<tr><td>Sintaxis:

for (inicialización; condición; iteración)

{

sentencia1;

sentencia2;

...

sentenciaN;</td></tr>
</table>

Estructura repetitiva for	
} (estructura for con un bloque de sentencias)	

Debes conocer

Como venimos haciendo para el resto de estructuras, visualiza el siguiente programa 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.

```
public class repetitiva_for {
     /* En este ejemplo se utiliza la estructura repetitiva for
     * para representar en pantalla la tabla de multiplicar del siete
     */
    public static void main(String[] args) {
        // Declaración e inicialización de variables
        int numero = 7;
        int contador;
        int resultado=0;

        //Salida de información
        System.out.println ("Tabla de multiplicar del " + numero);
        System.out.println ("............................. ");

        //Utilizamos ahora el bucle for
        for (contador=1; contador<=10;contador++)
        /* La cabecera del bucle incorpora la inicialización de la variable
         * de control, la condición de multiplicación hasta el 10 y el
         * incremento de dicha variable de uno en uno en cada iteración del
         * bucle.
         * En este caso contador++ incrementará en una unidad el valor de
         * dicha variable.
         */

        {
            resultado = contador * numero;
            System.out.println(numero + " x " + contador + " = " + resultado);
            /* A través del operador + aplicado a cadenas de caracteres,
             * concatenamos los valores de las variables con las cadenas de
             * caracteres que necesitamos para representar correctamente la
             * salida de cada multiplicación.
             */
        }
    }
}
```

1.2. 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.

```
1
2   public class repetitiva_for_in {
3       public static void main(String[] args) {
4           // Declaración e inicialización de variables
5           String[] semana = {"Lunes","Martes","Miércoles","Jueves","Viernes","Sá
6
7           //Salida de información
8
9           //Utilizamos ahora el bucle for/in
10          for (String dia: semana){
11          /* La cabecera del bucle incorpora la declaración de la variable dia
12           * a modo de contenedor temporal de cada uno de los elementos que form
13           * el array semana.
14           * En cada una de las iteraciones del bucle, se irá cargando en la var
15           * dia el valor de cada uno de los elementos que forman el array seman
16           * desde el primero al último.
17           */
18              System.out.println(dia);
19
20          }
21      }
22
23  }
```

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.

1.3. 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.

<table>
<tr><th colspan="2">Estructura repetitiva while</th></tr>
<tr><td>Sintaxis:

while (condición)

sentencia;</td><td>Sintaxis:

while (condición) {

sentencia1;

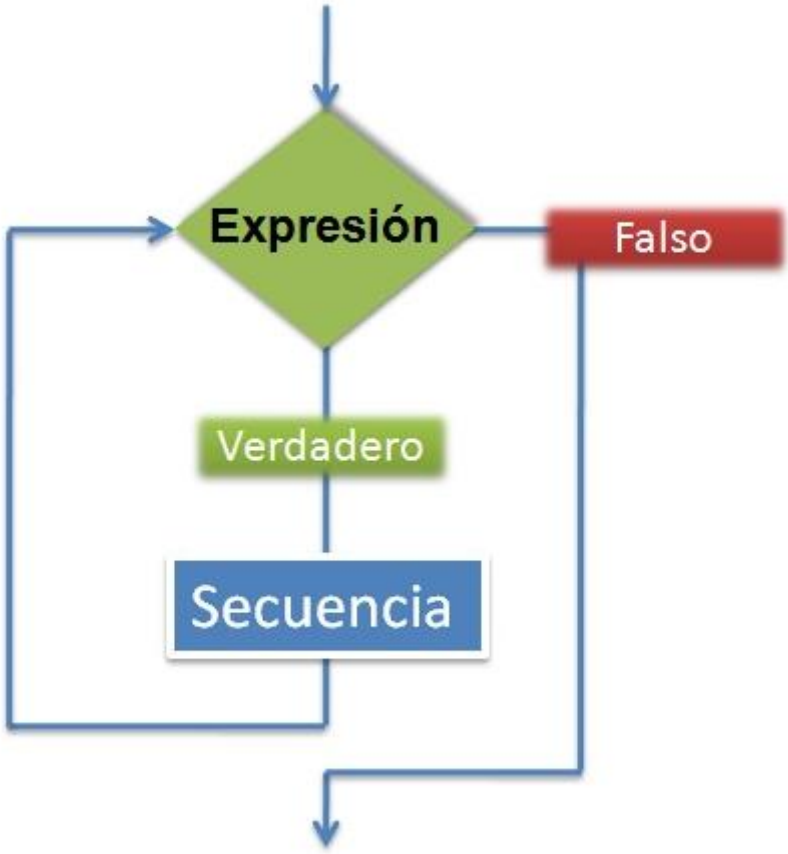
…

sentenciaN;

}</td></tr>
<tr><td colspan="2">Funcionamiento:</td></tr>
</table>

Estructura repetitiva while
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 código 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.

```
public class repetitiva_while {

    public static void main(String[] args) {
```

```
        // Declaración e inicialización de variables
        int numero = 7;
        int contador;
        int resultado=0;

        //Salida de información
        System.out.println ("Tabla de multiplicar del " + numero);
        System.out.println (".............................. ");

        //Utilizamos ahora el bucle while
        contador = 1; //inicializamos la variable contadora
        while (contador <= 10) //Establecemos la condición del
bucle
        {
            resultado = contador * numero;
            System.out.println(numero + " x " + contador + " = " +
resultado);
            //Modificamos el valor de la variable contadora, para
hacer que el
            //bucle pueda seguir iterando hasta llegar a finalizar
            contador++;
        }
    }
}
```

1.4. 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`	

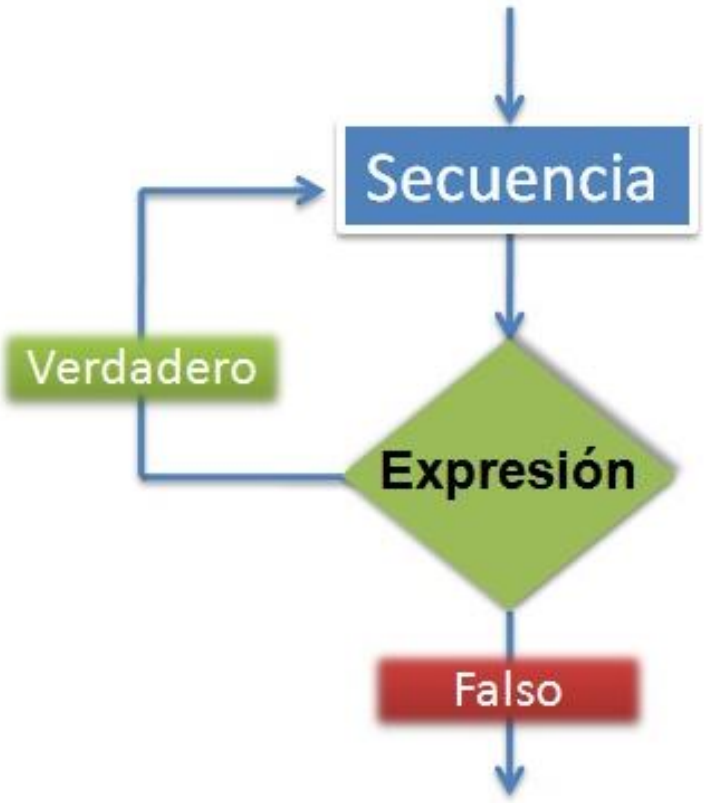
Sintaxis:
 	
`do`
`sentencia;`
`while (condición);`
	Sintaxis:
	
	`do`
	`{`
	`sentencia1;`

Estructura repetitiva `do-while`	

	…
	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 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

Estudia el siguiente programa 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`.

```
public class repetitiva_dowhile {
```

```
    public static void main(String[] args) {
      // Declaración e inicialización de variables
      int numero = 7;
      int contador;
      int resultado=0;

      //Salida de información
      System.out.println ("Tabla de multiplicar del " + numero);
      System.out.println (".............................. ");

      //Utilizamos ahora el bucle do-while
      contador = 1; //inicializamos la variable contadora
      do
      {
          resultado = contador * numero;
          System.out.println(numero + " x " + contador + " = " + resultado);
          //Modificamos el valor de la variable contadora, para hacer que el
          //bucle pueda seguir iterando hasta llegar a finalizar
          contador++;
      }while (contador <= 10); //Establecemos la condición del bucle

  }

}
```

3.D. ESTRUCUTURAS DE SALTO

1. Estructuras de salto.

¿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 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.

1.1. 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.

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`.

```java
public class sentencia_break {
    public static void main(String[] args) {
        // Declaración de variables
        int contador;


        //Procesamiento y salida de información

        for (contador=1;contador<=10;contador++)
        {
            if (contador==7)
                break;
            System.out.println ("Valor: " + contador);
        }
         System.out.println ("Fin del programa");
        /* El bucle sólo se ejecutará en 6 ocasiones, ya que cuando
         * la variable contador sea igual a 7 encontraremos un break q
         * romperá el flujo del bucle, transfiriéndonos a la sentencia
         * imprime el mensaje de Fin del programa.
         */
    }
```

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

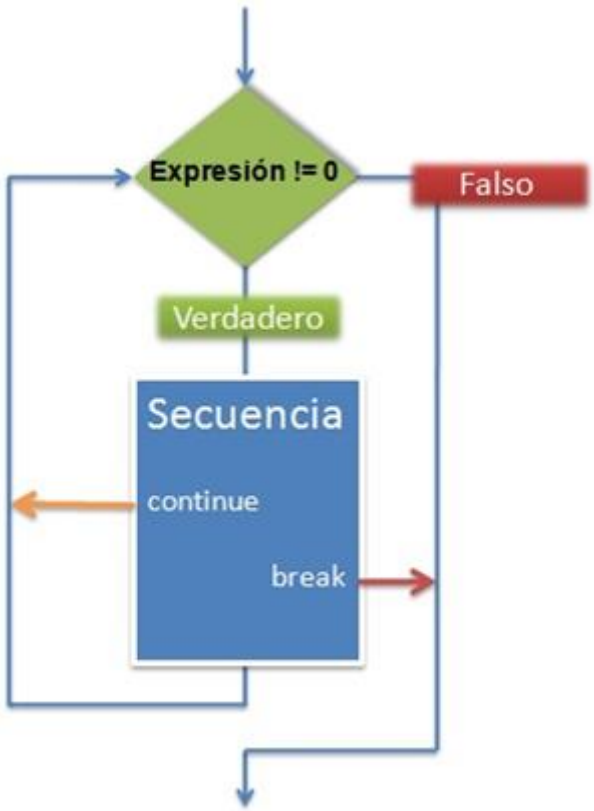
- Si aparece una sentencia `continue` dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia dará por terminada la iteración actual y se ejecuta una nueva iteración, evaluando de nuevo la expresión condicional del bucle.
- Si aparece en el interior 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
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 t
21            * de los números pares, serán aquellas en las que el resultado c
22            * calcular el resto de la división entre 2 de cada valor de la v
23            * contador, sea igual a 0.
24            */
25       }
26
27   }
28
```

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



1.2. 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 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.

```
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
14                     if (i==2) break bloque_dos;   //Llevamos a cabo el
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 }
```

1.3. 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.

Para saber más

En el siguiente programa 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.

```
import java.io.*;

/**

 *

 * Uso de return en un método

 */

public class sentencia_return {


    private static BufferedReader stdin = new BufferedReader( new
InputStreamReader(System.in));


    public static int suma(int numero1, int numero2)

    {

        int resultado;

        resultado = numero1 + numero2;

        return resultado; //Mediante return devolvemos el resultado de la suma

    }


    public static void main(String[] args) throws IOException {
```

```
        //Declaración de variables

        String input; //Esta variable recibirá la entrada de teclado

        int primer_numero, segundo_numero; //Estas variables almacenarán los
operandos



        // Solicitamos que el usuario introduzca dos números por consola

        System.out.print ("Introduce el primer operando:");

        input = stdin.readLine(); //Leemos la entrada como cadena de caracteres

        primer_numero = Integer.parseInt(input); //Transformamos a entero lo
introducido

        System.out.print ("Introduce el segundo operando: ");

        input = stdin.readLine(); //Leemos la entrada como cadena de caracteres

        segundo_numero = Integer.parseInt(input);  //Transformamos a entero lo
introducido



        //Imprimimos los números introducidos

        System.out.println ("Los operandos son: " + primer_numero + " y " +
segundo_numero);

        System.out.println ("obteniendo su suma... ");



        //Invocamos al método que realiza la suma, pasándole los parámetros
adecuados

        System.out.println ("La suma de ambos operandos es: " +
suma(primer_numero,segundo_numero));

    }

}
```