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`.

```
public class SentenciaBreak {
    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; // Rompe el flujo del bucle cuando contador es igual a 7 (es buena practica meterlo entre llaves)
            }
            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 que
          romperá el flujo del bucle, transfiriéndonos a la sentencia que
          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.

```
/*
  Uso de la sentencia continue
 */
public class SentenciaContinue {
    public static void main(String[] args) {
        // Declaración de variables
        int contador;

        System.out.println("Imprimiendo los números pares que hay del 1 al 10...");

        // Procesamiento y salida de información
        for (contador = 1; contador <= 10; contador++) {
            if (contador % 2 != 0) {
                continue; // Salta a la siguiente iteración si el número no es par
            }
            System.out.print(contador + " ");
        }

        System.out.println("\nFin del programa");
        /*
          Las iteraciones del bucle que generarán la impresión de cada uno
```

```
            de los números pares, serán aquellas en las que el resultado de
            calcular el resto de la división entre 2 de cada valor de la variable
            contador sea igual a 0.
           */
    }
}
```

Explicación del código:

1. **Declaración de variables:**
   - Se utiliza una variable `contador` para iterar a través del bucle.
2. **Salida de información inicial:**
   - Se imprime un mensaje que indica que se listarán los números pares.
3. **Estructura del bucle `for`:**
   - El bucle recorre los números del 1 al 10.
   - Dentro del bucle, el `if` verifica si el número es impar (`contador % 2 != 0`).
   - Si el número es impar, la sentencia `continue` se ejecuta, lo que hace que el programa pase directamente a la siguiente iteración sin ejecutar el código posterior (la impresión del número).
4. **Salida final:**
   - Se imprime "Fin del programa" después de que el bucle haya terminado.

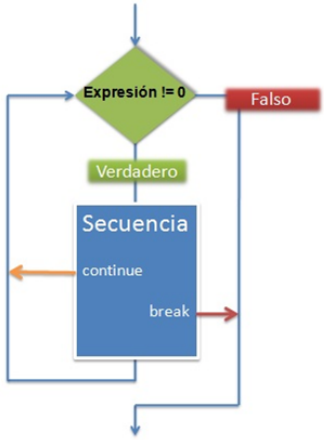
Ejecución:

La salida en la consola será:

```
Imprimiendo los números pares que hay del 1 al 10...
2 4 6 8 10
Fin del programa
```

Beneficio del `continue`:
La instrucción `continue` permite saltarse parte del cuerpo del bucle cuando no cumple una condición, haciendo que el código sea más limpio y eficiente.

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 en condiciones MUY específicas.

¿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.

```
/**
 * Uso de etiquetas en bucle
 */
public class Etiquetas {
    public static void main(String[] args) {
        for (int i = 1; i < 3; i++) { // Creamos cabecera del bucle
            bloque_uno: { // Creamos primera etiqueta
                bloque_dos: { // Creamos segunda etiqueta
                    System.out.println("Iteración: " + i);
                    if (i == 1) break bloque_uno; // Llevamos a cabo el primer salto
                    if (i == 2) break bloque_dos; // Llevamos a cabo el segundo salto
                    System.out.println("después del bloque dos");
                }
                System.out.println("después del bloque uno");
            }
        }
        System.out.println("Fin del bucle");
    }
}
```

Explicación del código:

1. **Etiquetas en Java:**
   - Las etiquetas (`bloque_uno:` y `bloque_dos:`) permiten asociar un bloque de código con un identificador.
   - Esto es útil para controlar el flujo de ejecución, especialmente en bucles anidados.
2. **Uso de `break` con etiquetas:**
   - `break bloque_uno;` interrumpe el bloque identificado como `bloque_uno`, saliendo de todo el contenido asociado a esa etiqueta.
   - `break bloque_dos;` interrumpe el bloque `bloque_dos` específicamente.
3. **Flujo del programa:**
   - En la primera iteración (`i == 1`), se ejecuta `break bloque_uno;` , lo que salta directamente fuera de todo el bloque `bloque_uno` .
   - En la segunda iteración (`i == 2`), se ejecuta `break bloque_dos;` , lo que salta fuera solo del bloque `bloque_dos` .
4. **Salida esperada:** El programa imprimirá:

```
Iteración: 1
Iteración: 2
después del bloque uno
Fin del bucle
```

- 5 **Beneficio del uso de etiquetas:**
  - Las etiquetas permiten un control más granular del flujo, útil en casos complejos con múltiples bucles o bloques anidados.

Las etiquetas se consideran asociadas a *Algoritmos de Programación Lineal* y está
MUY desaconsejado
su uso.

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 devolver o no un valor en función de lo especificado en el método y por lo tanto, terminar de dos maneras posibles :

- Para terminar la ejecución del método, que no devuelve nada, 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, sin devolver valor.
- Para devolver o retornar un valor, cuando el método así esté definido, 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 /*Se verá más adelante*/
        //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));
    }
}
```

Autoevaluación

¿Qué afirmación es correcta?

Con return, se puede finalizar la ejecución del método en el que se encuentre.

Con return, siempre se retornará un valor del mismo tipo o de un tipo compatible al definido en la cabecera del método.

Con return, puede retornarse un valor de un determinado tipo y suele hacerse al final del método. Además, el resto de respuestas también son correctas.