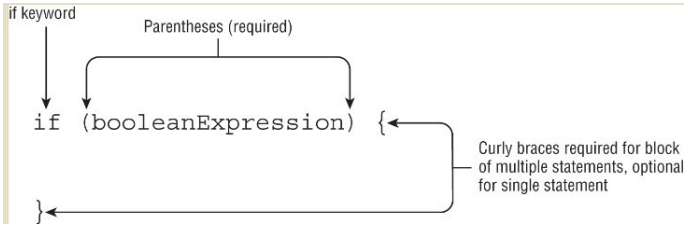
Tomando decisiones

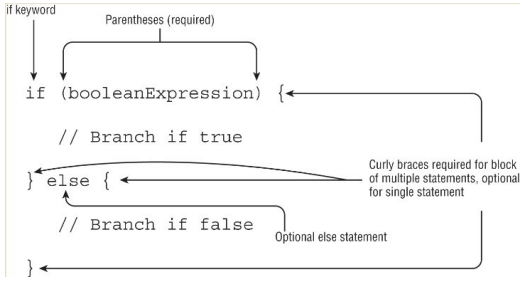
sentencia *IF*

El objetivo de una instrucción de toma de decisiones puede ser una sola instrucción o un bloque de instrucciones.




```java
// sentencia de una sola linea cuando solo hay una ejecución después del IF
if (a > b)
 a++;

// sentencia de múltiples lineas, cuando queremos hacer mas de una sentencia
if (a > 3){
 a++;
}
```

la sentencia ***IF-ELSE*** es utilizada para evitar tener que hacer 2 IF




```java
if(hora < 11){
 System.out.println("adios");
} else System.out.println("hola");
```

Reducción de código de coincidencia de patrones: Java 16 introdujo oficialmente la coincidencia de patrones con las instrucciones ***if*** y el operador ***instanceof***.

La coincidencia de patrones es una técnica para controlar el flujo del programa que solo ejecuta una sección de código que cumple con ciertos criterios. Para comprender por qué se agregó esta herramienta, considera el siguiente código que toma una instancia de ***Number*** y la compara con el valor 5. Si no has visto ***Number*** o ***Integer***, solo necesitas saber que ***Integer*** hereda de ***Number*** por ahora.

```java
void compararEnteros(Number number) {
    if (number instanceof Integer) {
        Integer data = (Integer) number;
        System.out.println(data.compareTo(5));
    }
}
```

El casting no es necesario ya que el método ***compareTo()*** está definido en ***Integer***, pero no en Number.

El código que primero verifica si una variable es de un tipo particular y luego la convierte inmediatamente a ese tipo es extremadamente común en el mundo Java. Es tan común que los autores de Java decidieron implementar una sintaxis más corta para ello:

```java
void compararEnteros(Number number) {
    if (number instanceof Integer data) {
        System.out.println(data.compareTo(5));
    }
}
```

La variable ***data*** en este ejemplo se conoce como la variable de patrón. Ten en cuenta que este código también evita cualquier ***ClassCastException*** potencial porque la operación de casting se ejecuta solo si el operador ***instanceof*** implícito devuelve verdadero.

La coincidencia de patrones incluye expresiones que se pueden usar para filtrar datos, como en el siguiente ejemplo:

```java
void printIntegersGreaterThan5(Number number) {
    if (number instanceof Integer data && data.compareTo(5) > 0) {
        System.out.print(data);
    }
}
```

Podemos aplicar varios filtros, o patrones, para que la sentencia ***if*** se ejecute solo en circunstancias específicas. Observa que estamos usando la variable de patrón en una expresión en la misma línea en la que se declara.

El tipo de la variable de patrón debe ser un subtipo de la variable en el lado izquierdo de la expresión. Tampoco puede ser del mismo tipo. Sin embargo, esta regla no existe para las expresiones tradicionales del operador ***instanceof***. Considera los siguientes dos usos del operador ***instanceof***:

```
1 Integer value = 123;
2 if (value instanceof Integer) {}
3 if (value instanceof Integer data) {} // NO COMPILA
```

Mientras que la segunda línea compila, la última no compila porque la coincidencia de patrones requiere que el tipo de la variable de patrón (***value)*** sea estrictamente un subtipo de ***Integer***.

Alcance de flujo: El compilador aplica el alcance del flujo cuando trabaja con la coincidencia de patrones. El alcance del flujo significa que la variable solo está dentro del alcance cuando el compilador puede determinar definitivamente su tipo.

```
1 void printIntegersOrNumbersGreaterThan5(Number number) {
2     if (number instanceof Integer data || data.compareTo(5) > 0)
3         System.out.print(data);
4 }
```

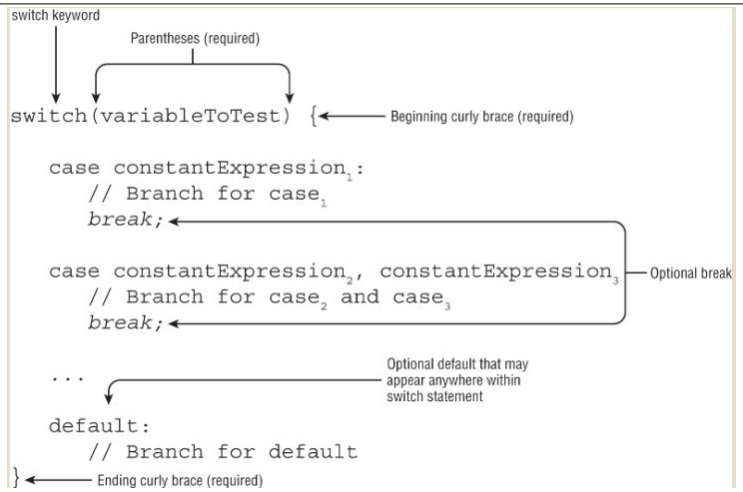
El problema radica en que la variable "data" solo está definida si la condición ***number instanceof Integer*** es verdadera. Si ***number*** no es un ***Integer***, "**data**" no existe y al tratar de usarla en la segunda parte de la condición (|| ***data.compareTo(5) > 0***) causará un error de compilación. El compilador no puede garantizar que "data" tenga un valor válido en todos los casos, por lo que la considera fuera de alcance y no compila el código. Por otra parte el siguiente ejemplo compila:

```
1 void printOnlyIntegers(Number number) {
2     if (!(number instanceof Integer data))
3         return;
4     System.out.print(data.intValue());
5 }
```

esto porque si la condición se cumple el código debe irse a la ultima linea de código para finalizar la función y para ello se debe asegurar la penúltima linea. esto provoca que si el ***if*** no se cumple la penúltima linea si se ejecute.

Sentencia SWITCH:

En este tipo de sentencia entra en la primera coincidencia que se encuentre y de no haber ejecuta la sentencia ***default***



a partir de Java 14 se puede combinar casos con una coma

```
1 switch (animal){
2  case 1, 2: System.out.println("hola");
3  case 3: System.out.println("adios");
4 }
```

Tener en cuenta que no es obligatorio tener casos dentro de una sentencia ***switch***, por ejemplo:

switch(month){} // compilara

utilizando la sentencia ***break***; podemos saltar las ejecuciones restantes después de haber ejecutado la sentencia asociado a la primera ubicación del ***case***. Por ejemplo en el siguiente código:

```
1  public void printSeason(int month) {
2          switch (month) {
3          case 1, 2, 3: System.out.print("Winter");
4          case 4, 5, 6: System.out.print("Spring");
5          default: System.out.print("Unknown");
6          case 7, 8, 9: System.out.print("Summer");
7          case 10, 11, 12: System.out.print("Fall");
8          }
9  }
10 Capitulo1 a1 = new Capitulo1();
11 a1.printSeason(5); //se imprime: SpringUnknownSummerFall
```

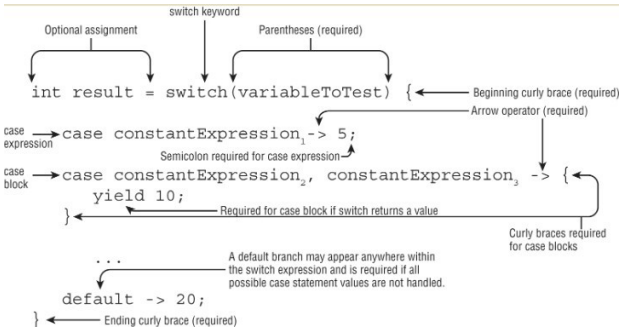
las sentencias ***switch*** soportan los datos de tipo:

- ***int*** y ***Integer***
- ***byte*** y ***Byte***
- ***short*** y ***Short***
- ***char*** y ***Character***
- ***String***
- ***enum*** (son un grupo de constantes como los días de la semana o meses del año)
- ***var*** (si el tipo fue resuelto previamente)

Los valores aceptados para los ***cases*** deben tener en cuenta; Primero, los valores en cada sentencia ***case*** deben ser constantes conocidas en tiempo de compilación y del mismo tipo de dato que el "**valor**" del ***switch***. Esto significa que solo puedes usar literales, constantes de enumeraciones (***enum***) o variables constantes "***final***" del mismo tipo de dato. Por ejemplo:

```
1  final int getCookies() { return 4; }
2  void feedAnimals() {
3     final int bananas = 1;
4     int apples = 2;
5     int numberOfAnimals = 3;
6     final int cookies = getCookies();
7     switch(numberOfAnimals) {
8        case bananas:      // compila, por utilizar final
9        case apples:       // no compila,  por no usar final
10        case getCookies():  // no compila, el método se ejecuta en tiempo de ejecución
11        case cookies: // no compila, el método se ejecuta en tiempo de ejecución
12        case 3 * 5:  // compila, porque el valor se determina en tiempo de compilación.
13     }
14 }
```

Expresiones *switch*: Estas expresiones ofrecen una forma más concisa y expresiva de escribir código condicional en comparación con las instrucciones ***switch*** tradicionales.



Hay que tener en cuenta que puede haber casos en los que esta nueva estructura requiera si o si una rama ***default***

```
1  String dayOfWeek = switch (day) {
2     case 1 -> "Lunes";
3     case 2 -> "Martes";
4     case 3, 4, 5 -> "Día laborable";
5     case 6, 7 -> "Fin de semana";
6     default -> "Día inválido";
7  };
8
9  System.out.println(dayOfWeek);
```

Todas las ramas de una expresión ***switch*** que no lancen una excepción deben devolver un tipo de dato consistente (si la expresión ***switch*** devuelve un valor). Esto significa que si una expresión ***switch*** está diseñada para devolver un valor, todas las ramas que no terminen en una excepción deben devolver un valor del mismo tipo.

Se requiere una rama ***default*** a menos que se cubran todos los casos o que no se devuelva ningún valor. Esto significa que si hay valores posibles para la expresión ***switch*** que no están cubiertos por ninguna rama ***case***, se debe incluir una rama ***default*** para manejar esos casos.

Si la expresión ***switch*** devuelve un valor, entonces cada rama que no sea una expresión debe usar la palabra clave ***yield*** para devolver un valor.

En el siguiente ejemplo no compila porque los 3 últimos valores devuelven un tipo que no es compatible con el resultado que se espera.

```
1  int a1 = 10;
2  int size switch(a1) {
3   case 5 -> 1;
4   case 10 -> (short)2;
5   default -> 5;
6   case 20 -> "3"; // no compila
7   case 40 -> 4L; // no compila
8   case 50 -> null; // no compila
9  }
```

Una expresión ***switch*** admite tanto una expresión como un bloque en las ramas ***case*** y ***default***. Al igual que un bloque regular, un bloque case está encerrado entre llaves ({ }). También incluye una instrucción ***yield*** si la expresión ***switch*** devuelve un valor. Por ejemplo, el siguiente código utiliza una mezcla de expresiones ***case*** y bloques:

```java
int fish = 5;
int length = 12;
var name = switch(fish) {
    case 1 -> "Goldfish";
    case 2 -> { yield "Trout"; }
    case 3 -> {
        if (length > 10) yield "Blobfish";
        else yield "Green";
    }
};
```

yield: Se utiliza dentro de un bloque para especificar el valor que se devuelve desde ese caso. Es como una versión de return para las expresiones ***switch***.

```java
int fish = 5;
int length = 12;
var name = switch(fish) {
 case 1 -> "Goldfish";
 case 2 -> {} // no compila por no utilizar yield
 case 3 -> {
 if(length > 10) yield "BLod";
 } // no compila porque no hay un yield por defecto
}
```

por ultimo, las expresiones ***switch***, indican que todas las posibilidades deban cubrirse como se evidencia en el siguiente ejemplo:

```java
String type = switch(canis){
 case 1 -> "dog";
 case 2 -> "wolf";
 case 3 -> "coyote";
}; // no compila ya que no se considera los caso 5, 4, -1, etc ya que estos van a devolver null, lo cual no es aceptado por Java
```

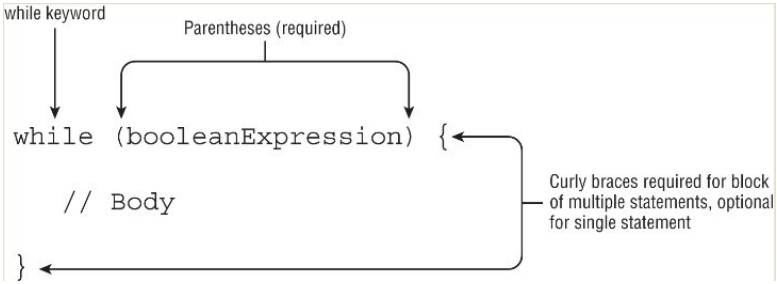
las posibles soluciones para estos casos son:

Primera solución (más común): Esta solución se refiere a utilizar la expresión ***default***.

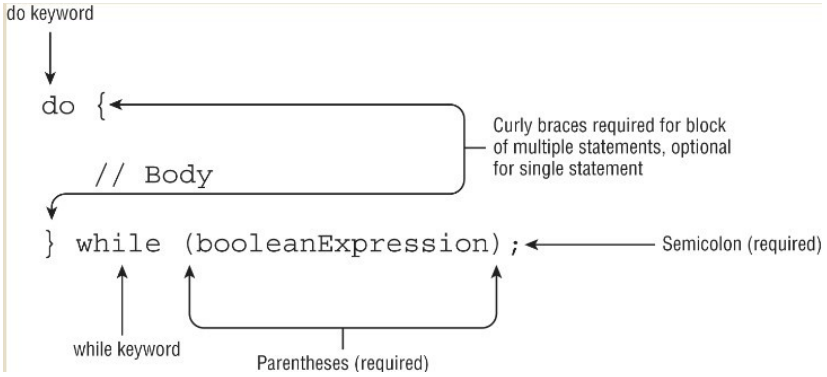
Segunda solución: Esta solución es más específica para ***enums***, que son tipos de datos que representan un conjunto fijo de constantes.
Se utiliza cuando queremos evaluar el valor de un ***enum*** y realizar acciones diferentes según el valor.

```java
enum Season {WINTER, SPRING, SUMMER, FALL}

String getWeather(Season value) {
   return switch(value) {
      case WINTER -> "Cold",
      case SPRING -> "Rainy",
      case SUMMER -> "Hot",
      case FALL -> "Warm"
   };
}
```

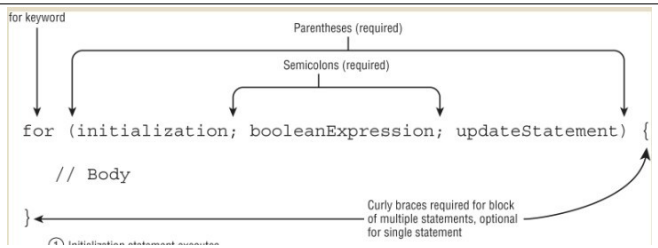
Loops en Java: se realizan a través de la sentencia ***while***, durante la ejecución se debe aumentar el contador de la condición



do-while es un tipo de sentencia loop que permite una primera ejecución de código



for-loop es un tipo de loop que permite ejecutarse para una cantidad especifica de repeticiones




```java
for (int i = 0; i < 10; i++) {
  System.out.println("a" + i); // en estos loops la primera impresión sera a0, no a1 ya que después de imprimir recién ejecuta el incremento en "i"
}
```

- LOOP infinito

```java
for(;;)
 System.out.println("hola mundo"); // esto si compilara
```

- agregando múltiples parámetros al ***for*** (si compila)

```java
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++){
 System.out.println(y + "");
}
System.out.println(x + "");
```

- redefiniendo una variable en el bloque inicial

```java
int x = 0;
for (int x = 4; x < 5; x++){
 System.out.println("hola mundo"); // no compila, la variable ya fue declarada
}
```

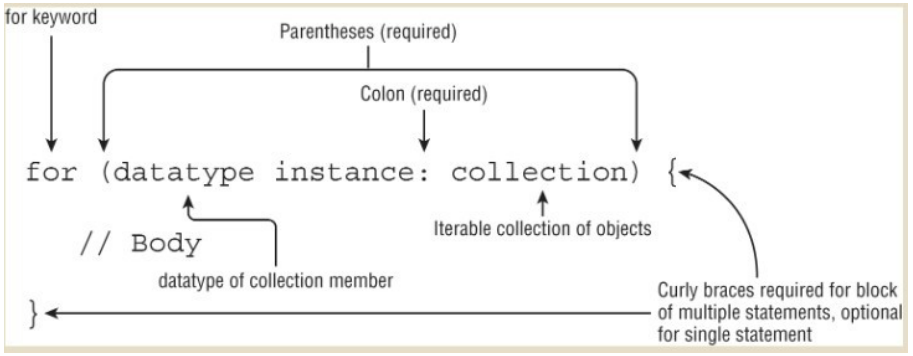
- utilizando tipos de dato incompatibles

```java
int x = 0;
for (long y = 0, int z = 4, x < 5; x++){ // no compila, todas las variables en el bloque de iniciación deben ser del mismo tipo
 System.out.println(y + "");
}
```

- utilizando variables del loop fuera de el

```java
for (long y = 0, x = 4; x < 5 && y < 10; x++, y++)
 System.out.println(y + "");
System.out.println(x); // no compila porque se usa una variable que solo vive en el loop.
```

for-each loop: es un loop especializado en recorrer ***Arrays*** y otras clases de tipo ***Collection***.



La declaración de un bucle ***for-each*** se compone de una sección de inicialización y un objeto sobre el que iterar. El lado derecho del bucle ***for-each*** debe ser uno de los siguientes:

- Un arreglo de Java integrado.
- Un objeto cuyo tipo implemente java.lang.Iterable (o extienda de la interfaz Collection)

El lado izquierdo debe incluir una declaración para una instancia de una variable cuyo tipo sea compatible con el tipo del arreglo o colección en el lado derecho de la declaración.

```java
public void printNames(String[] names) {
    for(int counter = 0; counter < names.length; counter++) {
        System.out.println(names[counter]);
    }
}

public void printNames(String[] names) {
    for(var name : names) {
        System.out.println(name);
    }
}
```

bucles anidados: Un bucle anidado es un bucle que contiene otro bucle, incluyendo bucles while, do/while, for y for-each. Por ejemplo, considera el siguiente código que itera sobre un arreglo bidimensional, que es un arreglo que contiene otros arreglos como sus miembros.

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray: myComplexArray) {
    for(int i=0; i<mySimpleArray.length; i++) {
        System.out.print(mySimpleArray[i]+"\t");
    }
    System.out.println();
}
```

obteniendo el siguiente resultado:

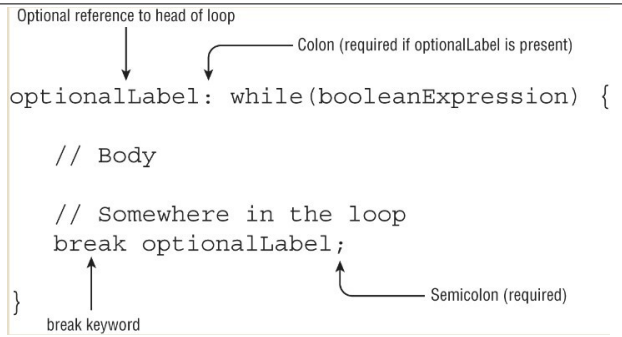
```
5 2 1   3
3 9 8   9
5 7 12 7
```

Etiquetas Opcionales: Una cosa que intencionalmente omitimos cuando presentamos las sentencias ***if***, ***switch*** y los bucles es que todos ellos pueden tener etiquetas opcionales. Una etiqueta es un puntero opcional al inicio de una sentencia que permite que el flujo de la aplicación salte a ella o se rompa desde ella. Es un único identificador seguido de dos puntos (:). Por ejemplo, podemos agregar etiquetas opcionales a uno de los ejemplos anteriores:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};

OUTER_LOOP: for(int[] mySimpleArray : myComplexArray) {

    INNER_LOOP: for(int i=0; i<mySimpleArray.length; i++) {
        System.out.print(mySimpleArray[i]+"\t");
    }

    System.out.println();
}
```

Las etiquetas siguen las mismas reglas de formato que los identificadores. Para una mejor legibilidad, comúnmente se expresan en mayúsculas con **snake_case**, utilizando guiones bajos entre palabras. Cuando se trabaja con un solo bucle, las etiquetas no añaden ningún valor, pero como verás en la siguiente sección, son extremadamente útiles en estructuras anidadas.

Sentencia break: Como viste al trabajar con sentencias ***switch***, una sentencia ***break*** transfiere el flujo de control hacia fuera de la sentencia que lo encierra. Lo mismo ocurre con una sentencia ***break*** que aparece dentro de un bucle ***while***, ***do/while*** o ***for***, ya que terminará el bucle antes de tiempo



El parámetro de etiqueta opcional nos permite salir de un bucle externo de nivel superior. En el siguiente ejemplo, buscamos la primera posición de índice (x, y) de un número dentro de un arreglo bidimensional no ordenado

```
public class FindInMatrix {
    public static void main(String[] args) {
        int[][] list = {{1, 13}, {5, 2}, {2, 2}};

        int searchValue = 2;
        int positionX = -1;
        int positionY = -1;

        PARENT_LOOP: for (int i = 0; i < list.length; i++) {
            for (int j = 0; j < list[i].length; j++) {
                if (list[i][j] == searchValue) {
                    positionX = i;
                    positionY = j;
                    break PARENT_LOOP; // esta sentencia rompe tanto el bucle interno como el externo, terminando la búsqueda.
                }
            }
        }

        if (positionX == -1 || positionY == -1) {
            System.out.println("Value " + searchValue + " not found");
        } else {
            System.out.println("Value " + searchValue + " found at: (" + positionX + "," + positionY + ")");
        }
    }
}
```

si remplazamos el bucle interno por lo siguiente:

```
if (list[i][j] == searchValue) {
    positionX = i;
    positionY = j;
    break;
}
```

El programa solo saldría del bucle interno actual, y continuaría buscando en las siguientes filas del arreglo, incluso si ya encontró el valor.

obtenemos: ***Value 2 found at: (2, 0)***

y si removemos la sentencia ***break***

```
if (list[i][j] == searchValue) {
    positionX = i;
    positionY = j;
}
```

obtenemos: ***Value 2 found at: (2, 1)***

sentencia *continue*: La sentencia ***continue*** te permite controlar el flujo de ejecución dentro de un bucle, permitiendo saltar a la siguiente iteración sin ejecutar el resto del código de la iteración actual. Esto es útil cuando quieres omitir ciertas partes del bucle en función de determinadas condiciones

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue; // Salta a la siguiente iteración si i es par
    }
    System.out.println(i);
}
```

Este código imprimirá solo los números impares del 0 al 9, ya que cuando ***i*** es par, la sentencia ***continue*** salta a la siguiente iteración.

La sentencia ***continue*** se aplica al bucle interno más cercano que está en ejecución. Sin embargo, podemos usar etiquetas opcionales para anular este comportamiento.

```
public class CleaningSchedule {
    public static void main(String[] args) {
        for (char stables = 'a'; stables <= 'd'; stables++) {
            CLEANING: for (int leopard = 1; leopard <= 2; leopard++) {
                if (stables == 'b' || leopard == 2) {
                    continue CLEANING; // Salta al siguiente establo o leopardo
                }
                System.out.println("Cleaning: " + stables + "," + leopard);
            }
        }
    }
}

// Salida: El programa imprimirá:
// Cleaning: a,1
// Cleaning: c,1
// Cleaning: d,1
```

pero si solo dejamos ***continue*** (eliminamos **cleaning**) obtendremos:

```
// Salida: El programa imprimirá:
// Cleaning: a,1
// Cleaning: a,3
// Cleaning: c,1
// Cleaning: c,3
// Cleaning: d,1
// Cleaning: d,3
```

Y si removemos el ***continue*** y las sentencias ***if*** obtendríamos

```
// Cleaning: a,1
// Cleaning: a,2
// Cleaning: a,3
// Cleaning: b,1
// Cleaning: b,2
// Cleaning: b,3
// Cleaning: c,1
// Cleaning: c,2
// Cleaning: c,3
// Cleaning: d,1
// Cleaning: d,2
// Cleaning: d,3
```

Sentencia *return*: en el siguiente ejemplo podremos ver como la sentencia ***return*** funciona como una mejor alternativa a las etiquetas y ***break***.

```
public class FindInMatrixUsingReturn {
   private static int[] searchForValue(int[][] list, int v) {
      for (int i = 0; i < list.length; i++) {
         for (int j = 0; j < list[i].length; j++) {
            if (list[i][j] == v) {
               return new int[] {i, j};
            }
         }
      }
      return null;
   }

   public static void main(String[] args) {
      int[][] list = {{1, 13}, {5, 2}, {2, 2}};
      int searchValue = 2;

      int[] results = searchForValue(list, searchValue);

      if (results == null) {
         System.out.println("Value " + searchValue + " not found");
      } else {
         System.out.println("Value " + searchValue + " found at: (" +
               results[0] + "," + results[1] + ")");
      }
   }
}
```

código inalcanzable: Un aspecto de ***break***, ***continue*** y ***return*** a tener en cuenta es que cualquier código colocado inmediatamente después de ellos en el mismo bloque se considera inalcanzable y no se compilará. Por ejemplo:

```
int checkDate = 0;
while (checkDate < 10) {
   checkDate++;
   if (checkDate > 100) {
      break;
      checkDate++; // no se compilar por considerarse código inalcanzable
   }
}
```

```
int minute = 1;
WATCH: while (minute > 2) {
   if (minute++ > 2) {
      continue WATCH;
      System.out.print(minute); // no se compilar por considerarse código
inalcanzable
   }
}

int hour = 2;
switch (hour) {
   case 1: return; hour++; // no se compilar por considerarse codigo inalcanzable
   case 2:
}
```

	Support labels	Support break	Support continue	Support yield
while	Yes	Yes	Yes	No
do/while	Yes	Yes	Yes	No
for	Yes	Yes	Yes	No
switch	Yes	Yes	No	Yes