

Unidad 4: Sentencias de control

Sentencia IF

Es un tipo de sentencia que habilita la selección de varios caminos alternativos en la ejecución del programa.

Para ello se van a utilizar las sentencias de selección. Java provee varias como son:

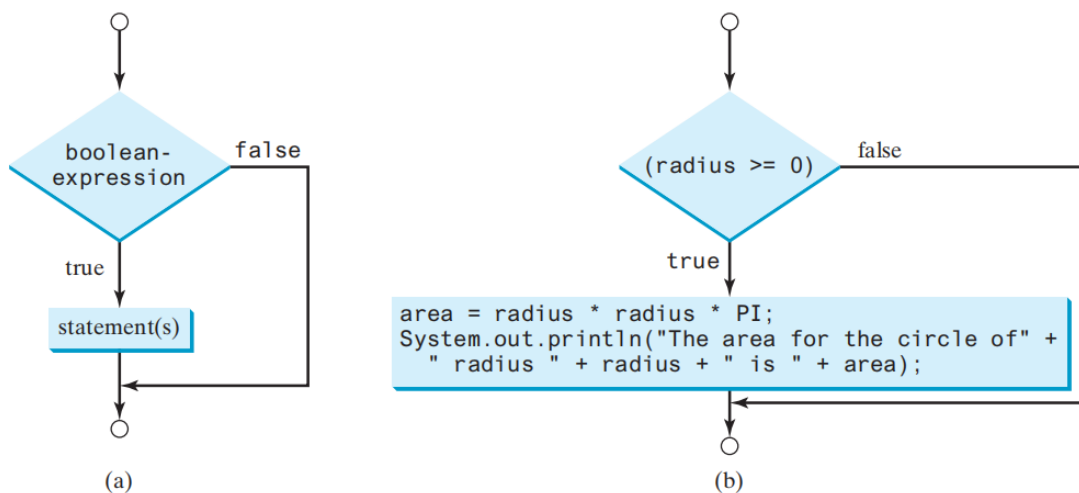
- `if`
- `if-else`
- `switch`

La sentencia `if` de una sólo vía tiene la siguiente sintaxis:

```
if (boolean-expression) {  
    statement1;  
    statement2;  
    statement3;  
    ...  
    statementn;  
}
```

- Sólo se va a ejecutar el `if` siempre y cuando `boolean-expression` sea **true**.

Se puede ver en el siguiente diagrama de flujo:



- En b se puede ver que si el `radius >= 0` pasa a realizar el bloque de código, si no, sigue con el flujo del programa.
- Obsérvese que no se puede ejecutar nunca el bloque de sentencias dentro de un **if** si la condición no es **true**.

Algunas observaciones a `if`:

```
if i > 0 {  
    System.out.println("i is positive");  
}
```

(a) Wrong

```
if (i > 0) {  
    System.out.println("i is positive");  
}
```

(b) Correct

Si sólo hay una sentencia:

```
if (i > 0) {  
    System.out.println("i is positive");  
}
```

(a)

Equivalent

```
if (i > 0)  
    System.out.println("i is positive");
```

(b)

Escribe el siguiente código:

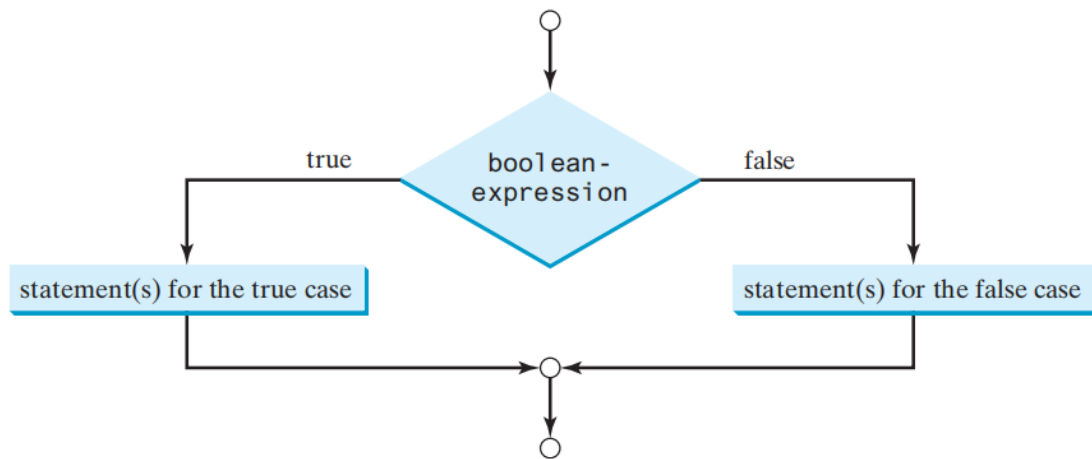
```
// SimpleIfDemo.java  
1 import java.util.Scanner;  
2  
3 public class SimpleIfDemo {  
4     public static void main(String[] args) {  
5         Scanner input = new Scanner(System.in);  
6         System.out.print("Enter an integer: ");  
7         int number = input.nextInt();  
8  
9         if (number % 5 == 0)  
10            System.out.println("HiFive");  
11  
12        if (number % 2 == 0)  
13            System.out.println("HiEven");  
14    }  
15 }
```

Sentencias if-else

La sentencia **if-else** funciona de la siguiente forma:

```
if (boolean-expression) {  
    statement(s)-for-the-true-case;  
}  
else {  
    statement(s)-for-the-false-case;  
}
```

El diagrama de flujo queda de la siguiente forma:



En este caso si `boolean-expression` es **true** pasa por la parte del `if` mientras que si es **false** pasa por la parte del `else`.

Veamos este ejemplo:

```

if (radius >= 0) {
    area = radius * radius * PI;
    System.out.println("The area for the circle of radius " +
        radius + " is " + area);
}
else {
    System.out.println("Negative input");
}
  
```

Otro ejemplo:

```

if (number % 2 == 0)
    System.out.println(number + " is even.");
else
    System.out.println(number + " is odd.");
  
```

Ejercicios:

- Escribe una sentencia if que aumenta una cantidad en 3% si es mayor que 90 y en caso contrario aumenta un 1%
- Cuál es la salida si `number` es 30 y si es 35.

```

if (number % 2 == 0)
    System.out.println(number + " is even.");
System.out.println(number + " is odd.");
  
```

(a)

```

if (number % 2 == 0)
    System.out.println(number + " is even.");
else
    System.out.println(number + " is odd.");
  
```

(b)

Sentencias if-else anidadas y alternativa múltiple

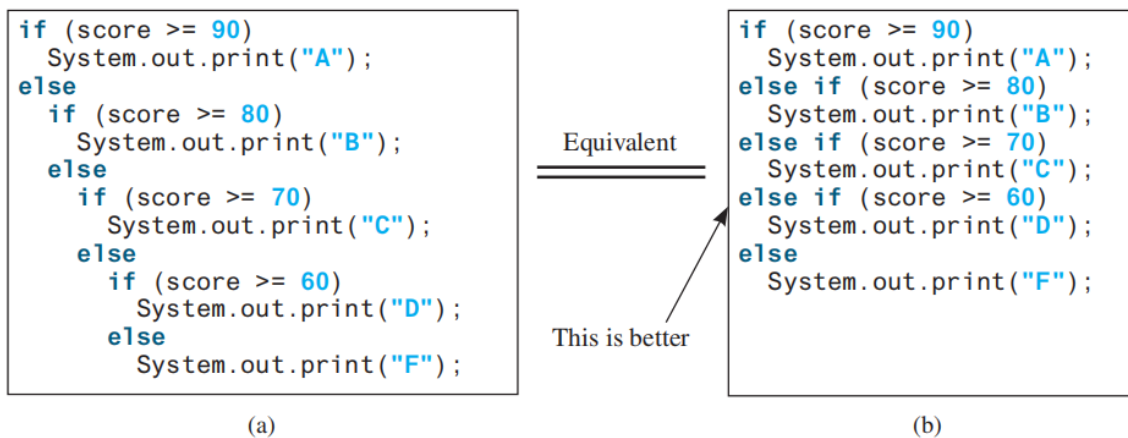
Una sentencia `if-else` puede estar dentro de otra y así sucesivamente, esto se llama anidación o que una sentencia está anidada

Veamos el ejemplo:

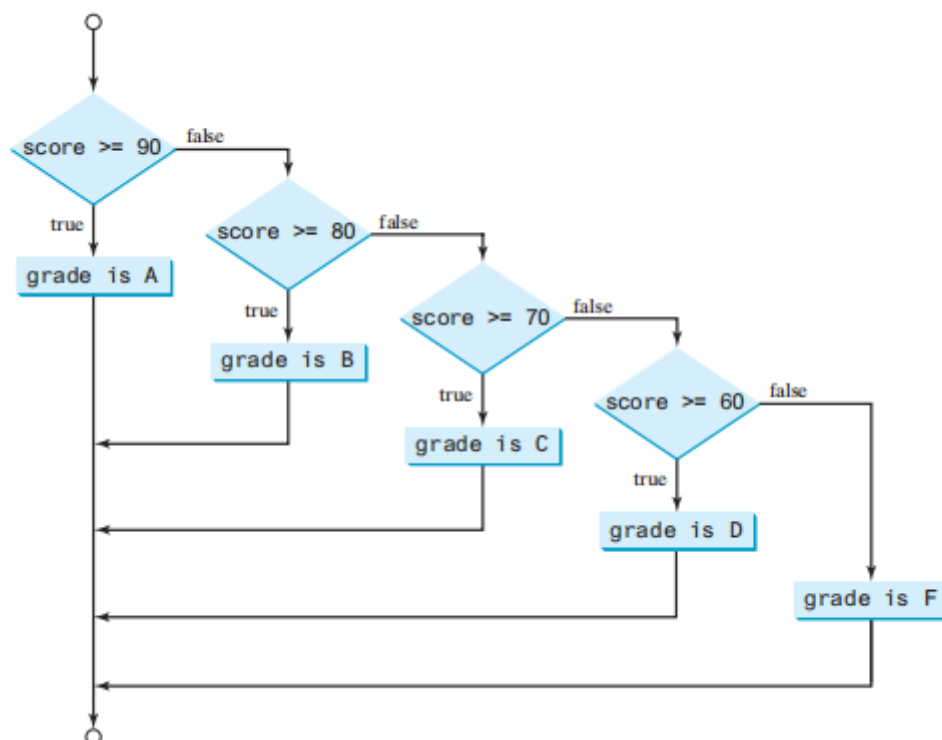
```
if (i > k) {  
    if (j > k)  
        System.out.println("i and j are greater than k");  
}  
else  
    System.out.println("i is less than or equal to k");
```

- La sentencia del primer if sólo si $i > k$ pasará a la segunda sentencia y evaluará $j > k$, si es cierto, imprimirá el mensaje de que i y j es mayor que k .
- Si $i > k$ es false no entrará en el if anidado y pasará a imprimir que **i es menor o igual a k** puesto que sólo sabemos que NO es mayor.
- Tampoco sabemos la relación entre i y j .

Para ello podemos utilizar la **alternativa múltiple**:



La forma b es mejor por ser más legible. Un diagrama que muestre la salida del programa anterior es la siguiente:



Operadores Lógicos

Los operadores lógicos o booleanos son:

- `||`: OR
- `&&`: AND
- `!`: NOT
- `XOR`: OR Exclusiva

TABLE 3.3 Boolean Operators

Operator	Name	Description
<code>!</code>	not	Logical negation
<code>&&</code>	and	Logical conjunction
<code> </code>	or	Logical disjunction
<code>^</code>	exclusive or	Logical exclusion

TABLE 3.4 Truth Table for Operator `!`

p	!p	Example (assume age = 24, weight = 140)
true	false	<code>!(age > 18)</code> is false, because <code>(age > 18)</code> is true.
false	true	<code>!(weight == 150)</code> is true, because <code>(weight == 150)</code> is false.

TABLE 3.5 Truth Table for Operator `&&`

p ₁	p ₂	p ₁ && p ₂	Example (assume age = 24, weight = 140)
false	false	false	
false	true	false	<code>(age > 28) && (weight <= 140)</code> is false, because <code>(age > 28)</code> is false.
true	false	false	
true	true	true	<code>(age > 18) && (weight >= 140)</code> is true, because <code>(age > 18)</code> and <code>(weight >= 140)</code> are both true.

TABLE 3.6 Truth Table for Operator `||`

p ₁	p ₂	p ₁ p ₂	Example (assume age = 24, weight = 140)
false	false	false	<code>(age > 34) (weight >= 150)</code> is false, because <code>(age > 34)</code> and <code>(weight >= 150)</code> are both false.
false	true	true	
true	false	true	<code>(age > 18) (weight < 140)</code> is true, because <code>(age > 18)</code> is true.
true	true	true	

TABLE 3.7 Truth Table for Operator ^

p ₁	p ₂	p ₁ ^ p ₂	Example (assume age = 24, weight = 140)
false	false	false	(age > 34) ^ (weight > 140) is false, because (age > 34) and (weight > 140) are both false.
false	true	true	(age > 34) ^ (weight >= 140) is true, because (age > 34) is false but (weight >= 140) is true.
true	false	true	
true	true	false	

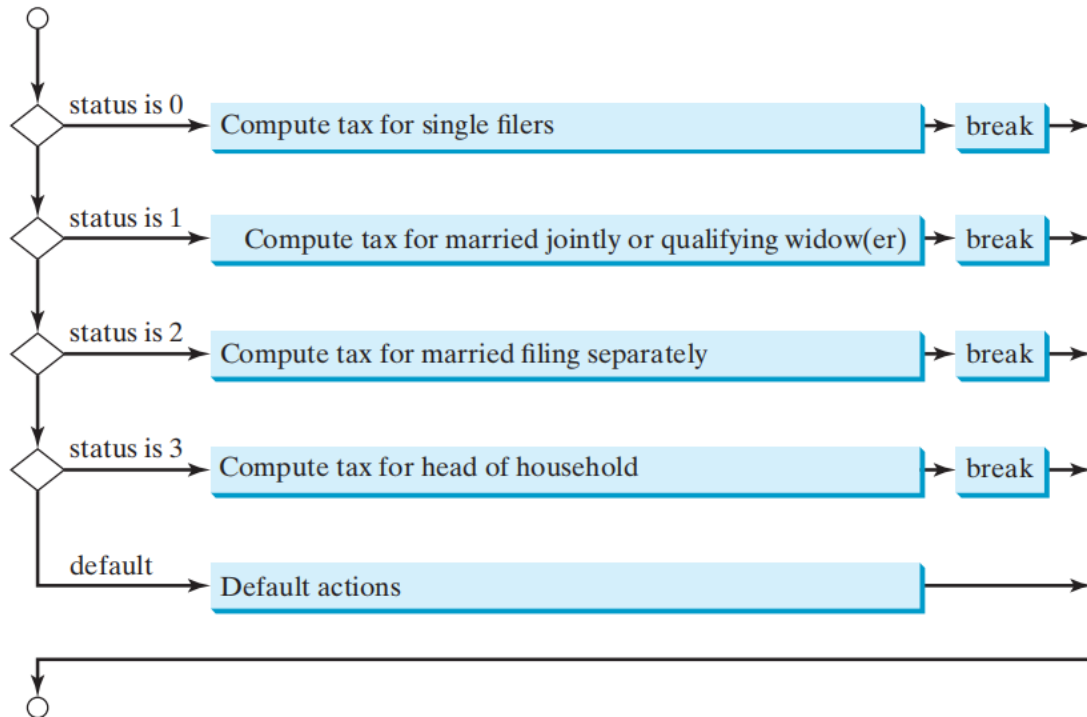
Sentencia `switch`

La sentencia `switch` ejecuta sentencias basadas en el valor de una variable o una expresión.

```
switch (status) {  
    case 0: compute tax for single filers;  
    break;  
    case 1: compute tax for married jointly or qualifying widow(er);  
    break;  
    case 2: compute tax for married filing separately;  
    break;  
    case 3: compute tax for head of household;  
    break;  
    default: System.out.println("Error: invalid status");  
    System.exit(1);  
}
```

La sentencia `switch` se utiliza para tratar varias opciones alternativas ya que las `if` y las `if-else` son menos claras de leer.

Aquí se muestra el diagrama de flujo de la sentencia anterior. Nótese que no son en sí sentencias sino ejemplos de lo que podría ser la sentencia.



Aquí mostramos lo que sería la **sintaxis** de switch

```

switch (switch-expression) {
  case value1: statement(s)1;
  break;
  case value2: statement(s)2;
  break;
  ...
  case valueN: statement(s)N;
  break;
  default: statement(s)-for-default;
}

```

La sentencia `<switch>` tiene las siguientes reglas:

- La expresión `switch-expression` debe darnos un valor **char, byte, short, int** o también de tipo **String** y debe estar siempre entre `()`.
- Los valores `value1...valueN` deben tener el mismo tipo de datos como valor de `switch-expression` y son **expresiones constantes**. No pueden tener variables como `1+x`.
- Cuando el valor de `switch-expression` coincide con algún valor `case`, se ejecuta la sentencia para el caso y luego con `break` sale del switch para la siguiente sentencia a ejecutar.
- El caso `default`, el cual es opcional, puede ser usado para llevar acciones cuando no hay valores `case` no coincidentes.
- Siempre es recomendable poner un `break` al finalizar la sentencias del case.

En el siguiente caso se imprimirá **Weekday** en los casos 1 a 5. En los casos 0 y 6 se imprime **Weekend**.

```
switch (day) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5: System.out.println("Weekday"); break;
    case 0:
    case 6: System.out.println("Weekend");
}
```

Operador condicional

El operador condicional evalúa una expresión basada en una condición:

```
if (x > 0)
    y = 1;
else
    y = -1;
```

Lo anterior se puede poner como:

```
y = (x > 0) ? 1 : -1;
```

Esto se llama **operador condicional** también llamado como **operador ternario** de la siguiente sintaxis:

boolean-expression ? expression1 : expression2

- Si el resultado de `boolean-expression` es `true` se le asigna el valor de `expression1`
- Si el resultado de `boolean-expression` es `false` se le asignará el valor de `expression2`

Ejemplos:

```
max = (num1 > num2) ? num1 : num2;
```

```
System.out.println((num % 2 == 0) ? "num es par" : "num es impar");
```

Operadores de precedencia generales

Ahora se muestran todos los operadores de precedencia, véase que ahora que hemos visto todos los operadores, la tabla se organiza de la siguiente forma:

TABLE 3.8 Operator Precedence Chart

<i>Precedence</i>	<i>Operator</i>
	<code>var++</code> and <code>var--</code> (Postfix)
	<code>+</code> , <code>-</code> (Unary plus and minus), <code>++var</code> and <code>--var</code> (Prefix)
	(type) (Casting)
	<code>!</code> (Not)
	<code>*</code> , <code>/</code> , <code>%</code> (Multiplication, division, and remainder)
	<code>+</code> , <code>-</code> (Binary addition and subtraction)
	<code><</code> , <code><=</code> , <code>></code> , <code>>=</code> (Relational)
	<code>==</code> , <code>!=</code> (Equality)
	<code>^</code> (Exclusive OR)
	<code>&&</code> (AND)
	<code> </code> (OR)
	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> (Assignment operators)

Sentencias repetitivas. Loops o bucles.

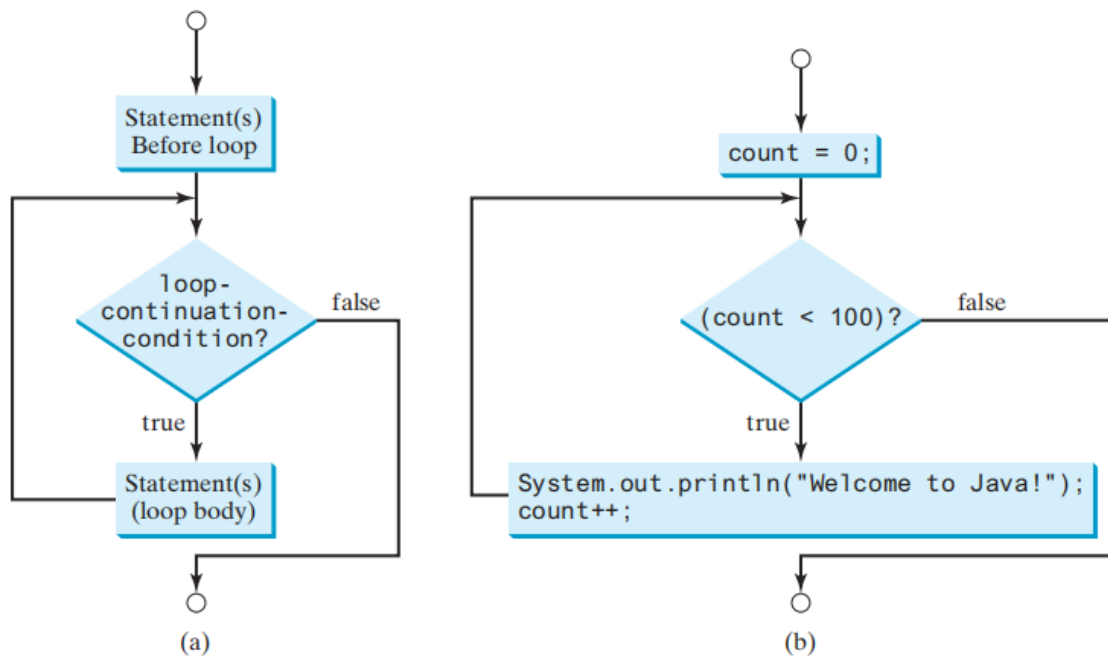
while

El bucle `while` ejecuta sentencias repetidamente mientras la condición es `true`.

Esta es la sintaxis de `while`

```
while (loop-continuation-condition) {  
    // Loop body  
    Statement(s);  
}
```

- El loop `while` ejecuta las sentencias dentro de las `{}` mientras `loop-continuation-condition` sea verdadera.
- `loop-continuation-condition` es una expresión booleana que en algún momento será falsa, si no, estaremos en un **bucle infinito**.
- Tendrás que controlar la condición dentro del bucle para que sea falsa en algún momento.



loop-continuation-condition

```

int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}
    
```

} loop body

En el bucle while anterior se llama **loop controlado por contador** y nos hacemos valer de una variable `count` la cual iniciamos y luego incrementamos dentro del loop hasta que su iteración llegue a `count == 100` donde ya sale del while para no ejecutar más las sentencias interiores.

Pureba el siguiente ejemplo:

```

int sum = 0, i = 1;
while (i < 10) {
    sum = sum + i;
    i++;
}
System.out.println("sum is " + sum); // sum is 45
    
```

Qué pasa con el siguiente ejemplo:

```

int sum = 0, i = 1;
while (i < 10) {
    sum = sum + i;
}
    
```

Lo anterior es un **bucle infinito** ya que `i < 10` siempre true.

Se puede utilizar una técnica de salida del bucle introduciendo un carácter por teclado:

```

char continueLoop = 'Y';
while (continueLoop == 'Y') {
    // Ejecuta las sentencias del bucle
    ...
    // Imprime la confirmación del usuario
    System.out.print("Introduce Y para continuar y N para salir: ");
    continueLoop = input.next().charAt(0);
}

```

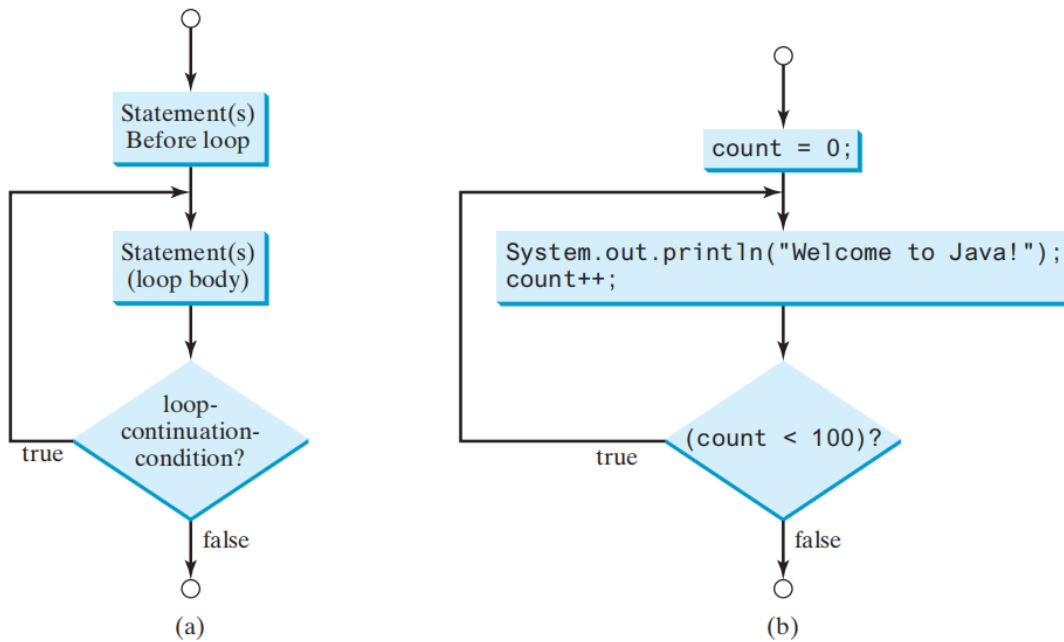
El bucle do-while

El bucle do-while es igual que el while exceptuando que primeramente se ejecuta el cuerpo del bucle y luego chequea si la condición de continuación es cierta.

```

do {
    // Loop body;
    Statement(s);
} while (loop-continuation-condition);

```



Si tenemos el bucle while:

```

int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}

```

Lo podemos transformar a:

```

int count = 0;
do {
    System.out.println("Welcome to Java!");
    count++;
} while (count < 100);

```

- Hay que tener en cuenta que el `do-while` se ejecuta al menos una vez y que la elección de uno u otro dependerá de la situación

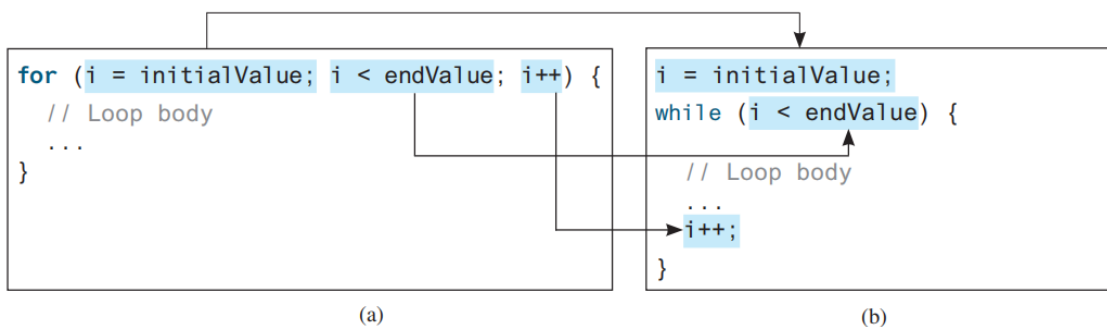
```
do {
    // Read the next data
    System.out.print("Enter an integer (the input ends if it is 0): ");
    data = input.nextInt();
    sum += data;
} while (data != 0);
```

Bucle for

El bucle for viene de un caso particular del while:

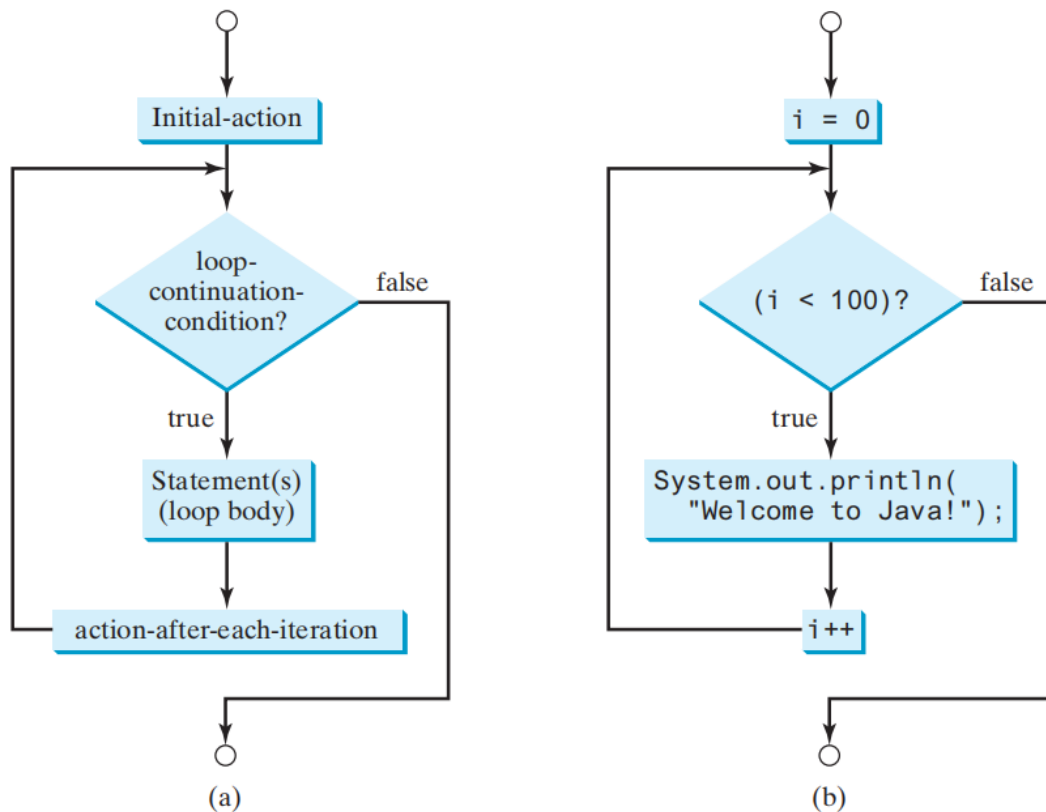
```
i = initialValue; // Initialize loop control variable
while (i < endValue) {
    // Loop body
    ...
    i++; // Adjust loop control variable
}
```

Este caso es muy utilizado así que se creó un tipo de bucle para hacerlo más simple:



- Sintaxis:

```
for (initial-action; loop-continuation-condition; action-after-each-iteration) {
    // Loop body;
    Statement(s);
}
```



- El bucle for tiene una inicialización inicial, una expresión booleana y luego un incremento.
- El orden debe ser el que se indica en los parámetros del for.
- El funcionamiento: inicializa, chequea la condición e incrementa. Luego ejecuta las sentencias.
- Después de ejecutarlas continuá la condición incrementada y verifica si la condición es true, si es así, incrementa y ejecuta las sentencias así hasta que la condición sea false, momento en que sale del bucle.

El siguiente bucle for imprime **Welcome to Java!** 100 veces:

```
int i;
for (i = 0; i < 100; i++) {
    System.out.println("Welcome to Java!");
}
```

También se puede declarar i directamente:

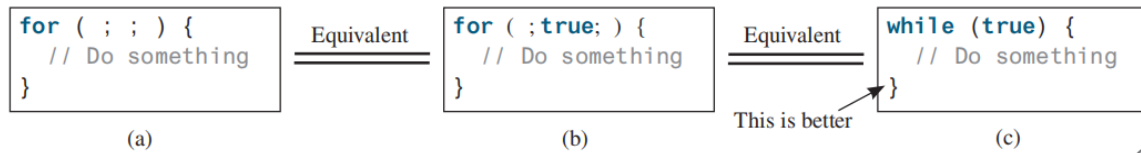
```
for (int i = 0; i < 100; i++) {
    System.out.println("Welcome to Java!");
}
for (int i = 0; i < 100; i++)
    System.out.println("Welcome to Java!");
```

Se pueden omitir las llaves con una sola sentencia.

También se pueden utilizar varias variables en el for:

```
for (int i = 0, j = 0; i + j < 10; i++, j++) {
    // Do something
}
```

Se pueden hacer loops infinitos:



Bucles `for` anidados

Los bucles `for` pueden estar dentro de forma anidada:

```
for (int i = 1; i <= 9; i++) {  
    for (int j = 1; j <= 9; j++) {  
        System.out.printf("%4d", i * j);  
    }  
}
```

Esto será muy útil a la hora de recorrer matrices.

Sentencias `break` y `continue`

Hemos visto como el `break` lo utilizamos en `switch` para romper la ejecución de la sentencia y no ejecutar más opciones.

También podemos utilizar `break` para bucles si queremos cortar la ejecución del mismo:

```
1 public class TestBreak {  
2     public static void main(String[] args) {  
3         int sum = 0;  
4         int number = 0;  
5  
6         while (number < 20) {  
7             number++;  
8             sum += number;  
9             if (sum >= 100)  
10                break;  
11        }  
12  
13        System.out.println("The number is " + number);  
14        System.out.println("The sum is " + sum);  
15    }  
16 }
```


```
The number is 14  
The sum is 105
```

Mientras que la sentencia `continue` lo que hace es que rompe la iteración pero pasa a la siguiente. Cuando ponemos `continue` ya no se ejecutan más sentencias de la interacción pero sí la siguiente iteración:

```

1 public class TestContinue {
2     public static void main(String[] args) {
3         int sum = 0;
4         int number = 0;
5
6         while (number < 20) {
7             number++;
8             if (number == 10 || number == 11)
9                 continue;
10            sum += number;
11        }
12
13        System.out.println("The sum is " + sum);
14    }
15 }

```



The sum is 189

Se puede reescribir el código sin `continue` ni `break`

```

int factor = 2;
while (factor <= n) {
    if (n % factor == 0)
        break;
    factor++;
}
System.out.println("The smallest factor other than 1 for "
    + n + " is " + factor);

```

Y sin `break`:

```

boolean found = false;
int factor = 2;
while (factor <= n && !found) {
    if (n % factor == 0)
        found = true;
    else
        factor++;
}
System.out.println("The smallest factor other than 1 for "
    + n + " is " + factor);

```