PROGRAMACIÓN

Desarrollo de Aplicaciones Multiplataforma

Manual del módulo

UD 3 – BUCLES

Ander Gonzalez Ortiz

3.1. INTRODUCCIÓN

Un bucle es un tipo de estructura que contiene un bloque de instrucciones que se ejecuta repetidas veces; cada ejecución o repetición del bucle se llama iteración.

El uso de bucles simplifica la escritura de programas, minimizando el código duplicado. Cualquier fragmento de código que sea necesario ejecutar varias veces seguidas es susceptible de incluirse en un bucle. Java dispone de los bucles: `while`, `do-while` y `for`.

3.2. BUCLES CONTROLADOS POR CONDICIÓN

El control del número de iteraciones se lleva a cabo mediante una condición. Si la evaluación de la condición es cierta, el bucle realizará una nueva iteración.

3.2.1. While

Al igual que la instrucción `if`, el comportamiento de `while` depende de la evaluación de una condición. El bucle `while` decide si realizar una nueva iteración basándose en el valor de la condición. Su sintaxis es:

```
while (condición) {
    bloque de instrucciones
    …
}
```

El comportamiento de este bucle (véase Figura 3 .1) es:

1. Se evalúa `condición`.
2. Si la evaluación resulta `true`, se ejecuta el bloque de instrucciones.
3. Tras ejecutarse el bloque de instrucciones, se vuelve al primer punto.
4. Si, por el contrario, la condición es `false`, terminamos la ejecución del bucle.

Por ejemplo, podemos mostrar los números del 1 al 3 mediante un bucle `while` controlado por la variable `i`, que empieza valiendo 1, con la condición `i <= 3`:

```
int i = 1; //valor inicial
while (i <=   3) { //el bucle iterará mientras i sea menor o igual que 3
    System.out.println(i); //mostramos i
    i++; //incrementamos i
}
```

Veamos una traza de la ejecución:

1. Se declara la variable `i` y se le asigna el valor 1.
2. La instrucción `while` evalúa la condición (i < = 3): ¿es 1 ≤ 3? Cierto.
3. Se ejecuta el bucle de instrucciones: `System.out.println` e `i++`. Ahora la `i` vale 2.
4. La instrucción `while` vuelve a evaluar la condición: ¿es 2 ≤ 3? Cierto.
5. Se ejecuta el bloque de instrucciones: `System.out.println` e `i++`. Ahora la `i` vale 3.

6. La instrucción `while` vuelve a evaluar la condición: ¿es 3≤ 3? Cierto.
7. Se ejecuta el bloque de instrucciones: `System.out.println` e `i++`. Ahora la `i` vale 4.
8. La instrucción `while` vuelve a evaluar la condición: ¿es 4 ≤ 3? Falso.
9. Se termina el bucle y se pasa a ejecutar la instrucción siguiente.

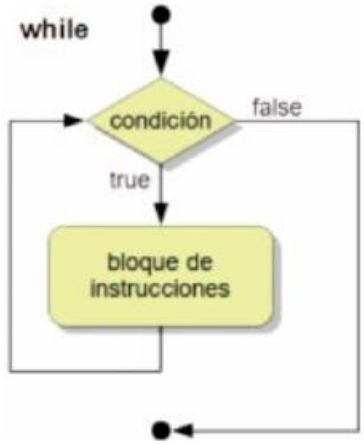

Figura 3.1. *Funcionamiento del bucle while.*

Un bucle `while` puede realizar cualquier número de iteraciones, desde cero, cuando la primera evaluación de la condición resulta falsa, hasta infinitas, en el caso de que la condición sea siempre cierta. Esto es lo que se conoce como bucle infinito.

Veamos un ejemplo de un bucle `while` que nunca llega a ejecutarse:

```
int cuentaAtras = -8; //valor negativo
while (cuentaAtras >= 0) {
     …
}
```

En este caso, independientemente del bloque de instrucciones asociado a la estructura `while`, no se llega a entrar nunca en el bucle, debido a que la condición no se cumple ni siquiera la primera vez. Se realizan cero iteraciones. En cambio,

```
int cuentaAtras = 10;
while (cuentaAtras >= 0) {
    System.out.println(cuenatAtras)
}
```

Dentro del bloque de instrucciones no hay nada que modifique la variable `cuentaAtras`, lo que hace que la condición permanezca idéntica, evaluándose siempre `true` y haciendo que el bucle sea infinito.

3.2.2. do-while

Disponemos de un segundo bucle controlado por una condición: el bucle do-while, muy similar a while, con la diferencia de que primero se ejecuta el bloque de instrucciones y después se evalúa la condición para decidir si se realiza una nueva iteración. Su sintaxis es:

```
do {
      bloque de instrucciones
      …
} while (condición);
```

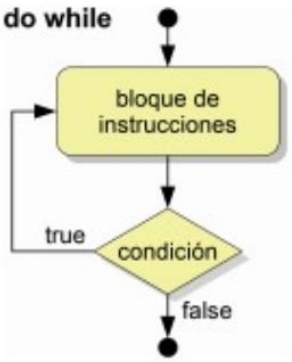

Figura 3.2. *Representación del flujo de ejecución de un bucle do-while.*

La Figura 3.2 describe su comportamiento. Se compone de los siguientes puntos:

1. Se ejecuta el bloque de instrucciones.
2. Se evalúa `condición`.
3. Según el valor obtenido, se termina el bucle o se vuelve al punto 1.

Como ejemplo, vamos a escribir el código que muestra los números del 1 al 10 utilizando un bucle `do-while`, en vez de un bucle `while`:

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <=   10);
```

Debemos recordar que es el único bucle que termina en punto y coma (`;`). Mientras el bucle `while` se puede ejecutar de 0 a in finitas veces, el `do-while` lo hace de 1 a infinitas veces. De hecho, la única diferencia con el bucle `while` es que `do-while` se ejecuta, al menos, una vez.

3.3. BUCLES CONTROLADOS POR CONTADOR: FOR

El bucle `for` permite controlar el número de iteraciones mediante una variable (que suele recibir el nombre de contador). La sintaxis de la estructura `for` es:

```
for (inicialización; condición; incremento) {
     bloque de instrucciones
     …
}
```

Donde,

- **Inicialización:** es una lista de instrucciones, separadas por comas, donde generalmente se inicializan las variables que van a controlar el bucle. Se ejecutan una sola vez antes de la primera iteración.
- **Condición:** es una expresión booleana que controla las iteraciones del bucle. Se evalúa antes de cada iteración: el bloque de instrucciones se ejecutará solo cuando el resultado sea `true`.
- **Incremento:** es una lista de instrucciones, separadas por comas, donde se suelen modificar las variables que controlan la condición. Se ejecuta al final de cada iteración.

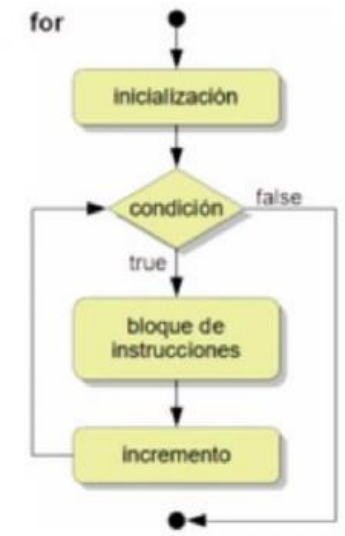

Figura 3.3. *Secuencia del flujo de control de un bucle for.*

El funcionamiento de `for` se describe en la Figura 3.3. Consiste en los siguientes puntos:

1. Se ejecuta la inicialización; esto se hace una sola vez al principio.
2. Se evalúa la condición: si resulta `false`, salimos del bucle y continuamos con el resto del programa; en caso de que la evaluación sea `true`, se ejecuta todo el bloque de instrucciones.
3. Cuando termina de ejecutarse el bloque de instrucciones, se ejecuta el incremento.
4. Se vuelve de nuevo al punto 2.

Aunque `for` está controlado por una condición que, en principio, puede ser cualquier expresión booleana, la posibilidad de configurar la inicialización y el incremento de las variables que controlan el bucle permite determinar de antemano el número de iteraciones.

Veamos un ejemplo donde solo se usa la variable `i` para controlar el bucle:

```
for (int i = 1; i < 2; i++) {
      System.out.priln("La i vale"+ i);
}
```

A continuación, se muestra una traza de la ejecución del bucle anterior:

1. Primero se ejecuta la inicialización: `i=l`;

2. Evaluamos la condición: ¿es cierto que `i` ≤ 2? Es decir: ¿1 ≤ 2?
3. Cierto. Ejecutamos el bloque de instrucción: `Systern.out.println(...)`
4. Obtenemos el mensaje: «La `i` vale 1».
5. Terminado el bloque de instrucciones, ejecutamos el incremento: (`i++`) `i` vale 2.
6. Evaluamos la condición: ¿es cierto que `i` ≤ 2? Es decir: ¿2 ≤ 2?
7. Cierto. Ejecutamos `Systern.out.println(...)`
8. Obtenemos el mensaje: «La `i` vale 2».
9. Ejecutamos el incremento: (`i++`) la `i` vale 3.
10. Evaluamos la condición: ¿es cierto que `i` ≤ 2? Es decir: ¿3 ≤ 2?
11. Falso. El bucle termina y continúa la ejecución de las sentencias que siguen a la estructura `for`.

3.4. SALIDAS ANTICIPADAS

Dependiendo de la lógica que implementar en un programa, puede ser interesante terminar un bucle antes de tiempo y no esperar a que termine por su condición (realizando todas las iteraciones). Para poder hacer esto disponemos de:

- `break`: interrumpe completamente la ejecución del bucle.
- `continue`: detiene la iteración actual y continúa con la siguiente.

Cualquier programa puede escribirse sin utilizar `break` ni `continue`; se recomienda evitarlos, ya que rompen la secuencia natural de las instrucciones. Veamos un ejemplo:

```
i = 1;
while (i <= 10) {
    System.out.println("La i vale" + i);
    if (i == 2) {
        break;
    }
    i++;
}
```

En un primer vistazo da la impresión de que el bucle ejecutará 10 iteraciones, pero cuando está realizando la segunda (`i` vale 2), la condición de if se evalúa como cierta y entra en juego `break`, que interrumpe completamente el bucle, sin que se ejecuten las sentencias restantes de la iteración en curso ni el resto de las iteraciones. Tan solo se ejecutan dos iteraciones y se obtiene:

La `i` vale 1

La `i` vale 2

Veamos otro ejemplo:

```
i = 0;
while (i < 10) {
   i++;
   if (i % 2 == 0) {
       continue;
   }
```

```
   System.out.println("La i vale " + i);
}
```

Cuando la condición `i % 2 == 0` sea cierta, es decir, cuando i es par, la sentencia `continue` detiene la iteración actual y continúa con la siguiente, saltándose el resto del bloque de instrucciones. `System.out.println` solo llegará a ejecutarse cuando `i` sea impar, o dicho de otro modo: en iteraciones alternas. Se obtiene la salida por consola:

La i vale 1
La i vale 3
La i vale 5
La i vale 7
La i vale 9

3.5. BUCLES ANIDADOS

En el uso de los bucles es muy frecuente la anidación, que consiste en incluir un bucle dentro de otro, como describe la Figura 3.4.

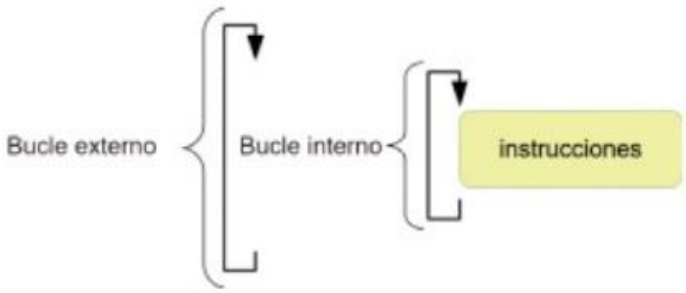

Figura 3.4. *Bucles anidados.*

Al hacer esto se multiplica el número de veces que se ejecuta el bloque de instrucciones de los bucles internos. Los bucles anidados pueden encontrarse relacionados cuando las variables de los bucles más externos intervienen en el control de la iteración de un bucle interno; o independientes, cuando no existe relación alguna entre ellos.

3.5.1. Bucles independientes

Cuando los bucles anidados no dependen, en absoluto, unos de otros para determinar el número de iteraciones, se denominan bue/es anidados independientes. Veamos un ejemplo sencillo, la anidación de dos bucles for:

```
for (i = 1; i <= 4; i++) {
     for (j = 1; j <= 3; j++) {
          System.out.println(“Ejecutando...");
     }
}
```

El bucle externo, controlado por la variable `i`, realizará cuatro iteraciones, donde `i` toma los valores 1, 2, 3 y 4. En cada una de ellas, el bucle interno, controlado por `j`, realizará tres iteraciones, tomando `j` los valores 1, 2 y 3. En total, el bloque de instrucciones se ejecutará doce veces.

Anidar bucles es una herramienta que facilita el procesado de tablas multidimensionales (Unidad 6). Se utiliza cada nivel de anidación para manejar el índice de cada dimensión. Sin embargo, el uso descuidado de bucles anidados puede convertir un algoritmo en algo ineficiente, disparando el número de instrucciones ejecutadas.

3.4.2. Bucles dependientes

Puede darse el caso de que el número de iteraciones de un bucle interno no sea independiente de la ejecución de los bucles exteriores, y dependa de sus variables de control.

Decimos entonces que son bue/es anidados dependientes. Veamos el siguiente fragmento de código, a modo de ejemplo, donde la variable utilizada en el bucle externo (`i`) se compara con la variable (`j`) que controla el bucle más interno. En algunas ocasiones, la dependencia de los bucles no se aprecia de forma tan clara como en el ejemplo:

```
for (i = 1; i <= 3; i++) {
    System.out.println("Bucle externo, i=" +i);
    j = l;
    while (j <= i) {
        System.out.println("...Bucle interno, j=" + j);
        j++;
    }
}
```

que proporciona la salida:

Bucle externo, i=1
...Bucle interno, j=1
Bucle externo, i=2
...Bucle interno, j=1
...Bucle interno, j=2
Bucle externo, i=3
...Bucle interno, j=1
...Bucle interno, j=2
...Bucle interno, j=3

- Durante la primera iteración del bucle `i`, el bucle interno realiza una sola iteración.
- En la segunda iteración del bucle externo, con `i` igual a 2, el bucle interno realiza dos iteraciones.
- En la última vuelta, cuando `i` vale 3, el bucle interno se ejecuta tres veces.

La variable `i` controla el número de iteraciones del bucle interno y resulta un total de 1 + 2 + 3 = 6 iteraciones. Los posibles cambios en el número de iteraciones de estos bucles hacen que, a priori, no siempre sea tan fácil conocer el número total de iteraciones.