Capítulo 4. Control del flujo de programa

A. J. Pérez

Control del flujo de programa

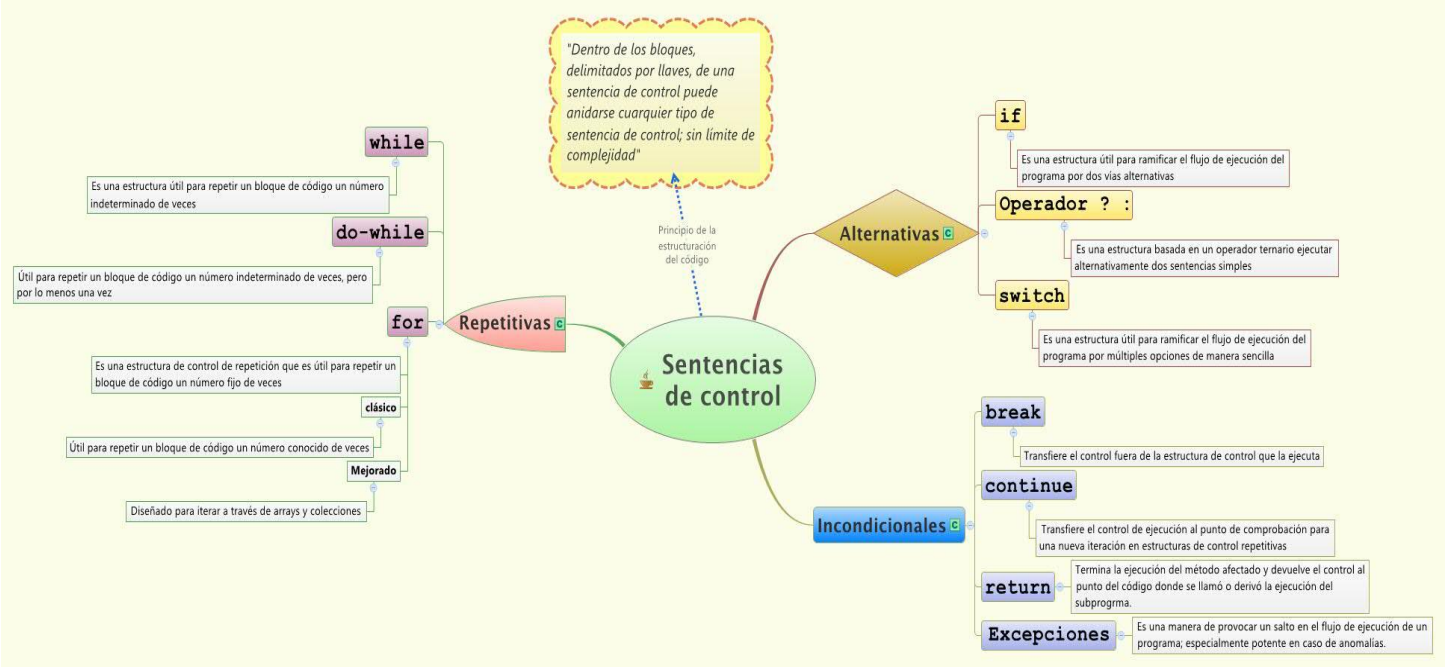
Teoría de la estructuración

El control del flujo de ejecución es la manera que tiene un lenguaje de programación de hacer que la ejecución de un programa avance, se ramifique o itere en función de los cambios de estado de los datos.

Bohm y Jacopini demostraron que cualquier programa se puede escribir con tres estructuras de control fundamentales: la *estructura secuencial*, la *estructura alternativa* y la *estructura de iteración*.

Las estructuras *secuencial, alternativas e iterativas* serían los elementos más básicos y explícitos del control de flujo de ejecución de un programa. En Java, y otros lenguajes orientados a la productividad, existen otros mecanismos, menos evidentes y más avanzados, relacionados con el *control de flujo incondicional* que serán tratados aparte; son:

- El mecanismo de las llamadas a subprogramas (métodos)
- El mecanismo de las excepciones.
- el mecanismo de los asertos para las pruebas.



Estructura secuencial

La estructura secuencial está integrada en todos los lenguajes algorítmicos como Java. A menos que se le indique lo contrario, un programa se ejecuta procesando las instrucciones una después

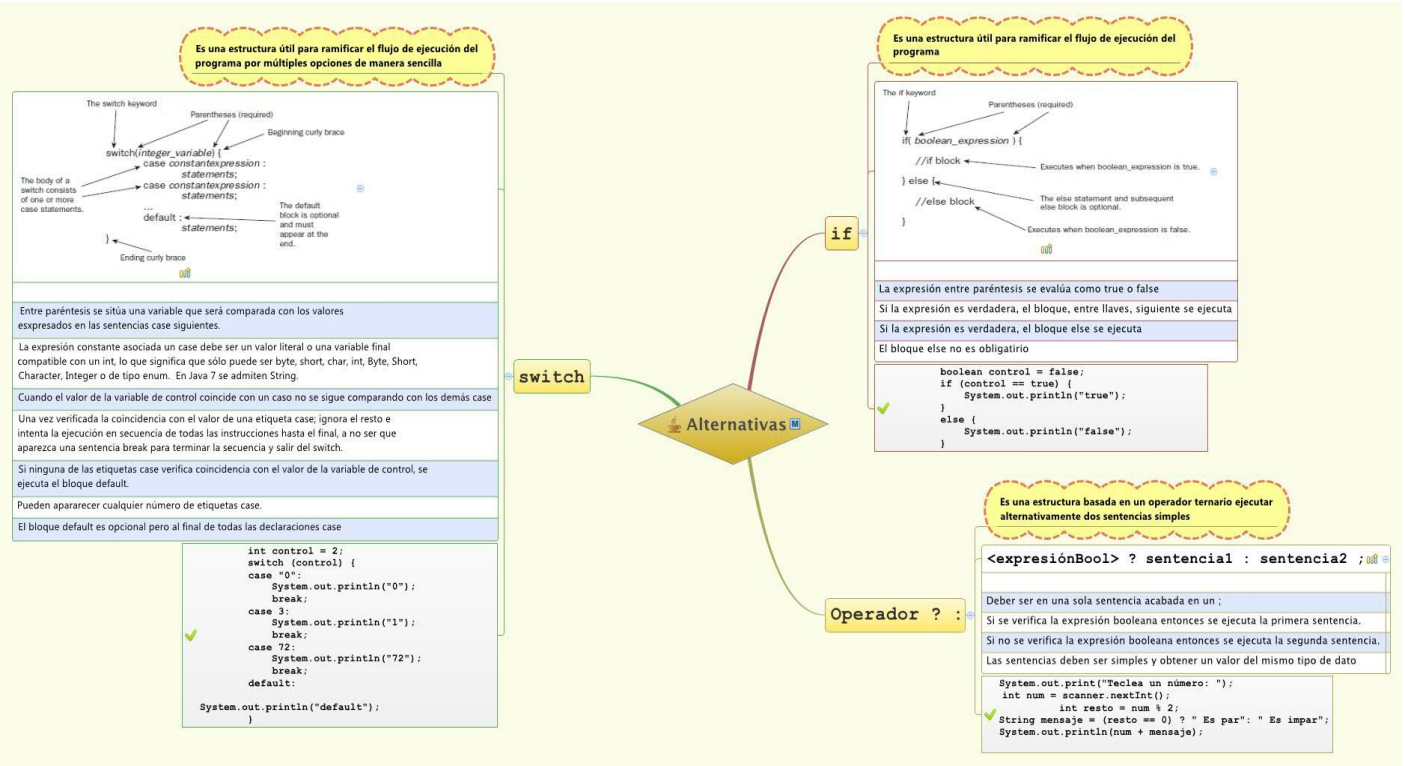
de otra, en el orden exacto en que están expresadas; o sea, en secuencia.

Se pueden establecer cualquier cantidad de acciones en una *estructura de secuencia*. además cada una de esas acciones puede ser a su vez se una subsecuencia. Es el principio básico de la estructuración.

Estructuras alternativas, condicionales o selectivas

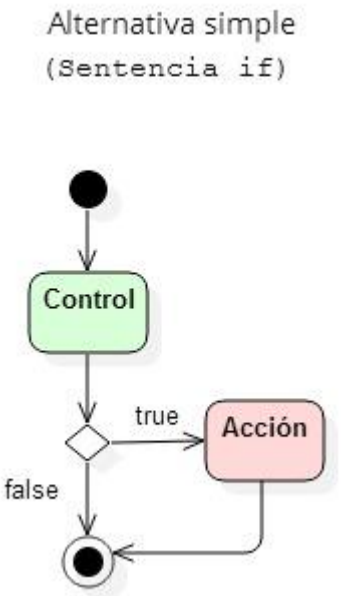
En los programas, se necesitará con frecuencia que algunas sentencias se ejecuten condicionalmente y otras sean omitidas; Java proporciona tres mecanismos para conseguir este control y decidir qué partes del código ejecutar:

- Alternativa simple: **`if`**
- Alternativa doble: **`if-else`**
- alternativa múltiple: **`switch`**
- Decisión in-line: con el operador **`? :`**



Alternativa simple: if

Es una estructura de control útil para la ejecución o no, de un bloque de instrucciones. La construcción **`if`** permite la ejecución o no de un bloque de código dependiendo del estado que toma una *expresión booleana* que actúa de *control*.



```
if (<expresiónBool>) {
     <bloqueSentencias> ;
}
```

- El *bloque de sentencias* pueden ser una instrucción simple o un conjunto delimitado por llaves **{ }**. Si fuese una única instrucción; las llaves se pueden omitir.
- Una *expresión booleana* es cualquier expresión más o menos compleja formada por la combinación de *operadores de relación* y *operadores lógicos* de la que se obtiene un resultado **`true`** o **`false`**. Podría ser una variable simple declarada como **`boolean`**.

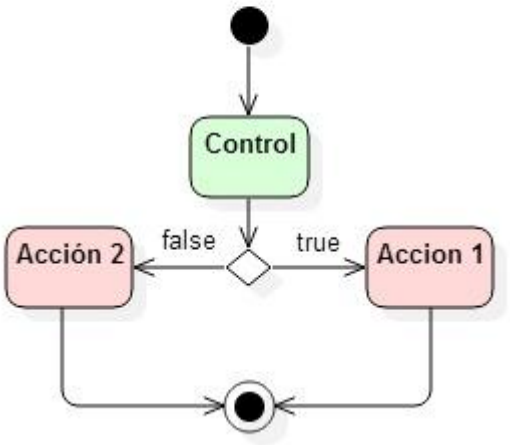
```
//...

//Pide la edad y determina si es menor de edad

Scanner teclado = new Scanner(System.in);
int edad;

System.out.print("¿Cuál es tu edad: ");
edad = teclado.nextInt();

if (edad < 18) {
    System.out.println("¡Eres menor de edad !");
}

//...
```

Si sólo se tienen que incluir una instrucción asociada al **`if`**, las llaves son opcionales.

Alternativa doble: if-else

Es una estructura de control útil para bifurcar el flujo de ejecución del programa. La construcción **`if-else`** permite la ejecución alternativa de dos bloques de código dependiendo del estado que toma una *expresión booleana* que actúa de *control*.

Alternativa doble
(Sentencia if - else)




```
if (<expresiónBool>) {
      <bloqueSentencias1> ;
}

else {
      <bloqueSentencias2> ;
}
```

- Cada uno de los *bloques de sentencias* pueden ser una instrucción simple o un conjunto delimitado por llaves **{ }**. Si fuese una única instrucción; las llaves se pueden omitir.
- Una *expresión booleana* es cualquier expresión más o menos compleja formada por la combinación de *operadores de relación* y *operadores lógicos* de la que se obtiene un resultado **true** o **false**. Podría ser una variable simple declarada como **boolean**.

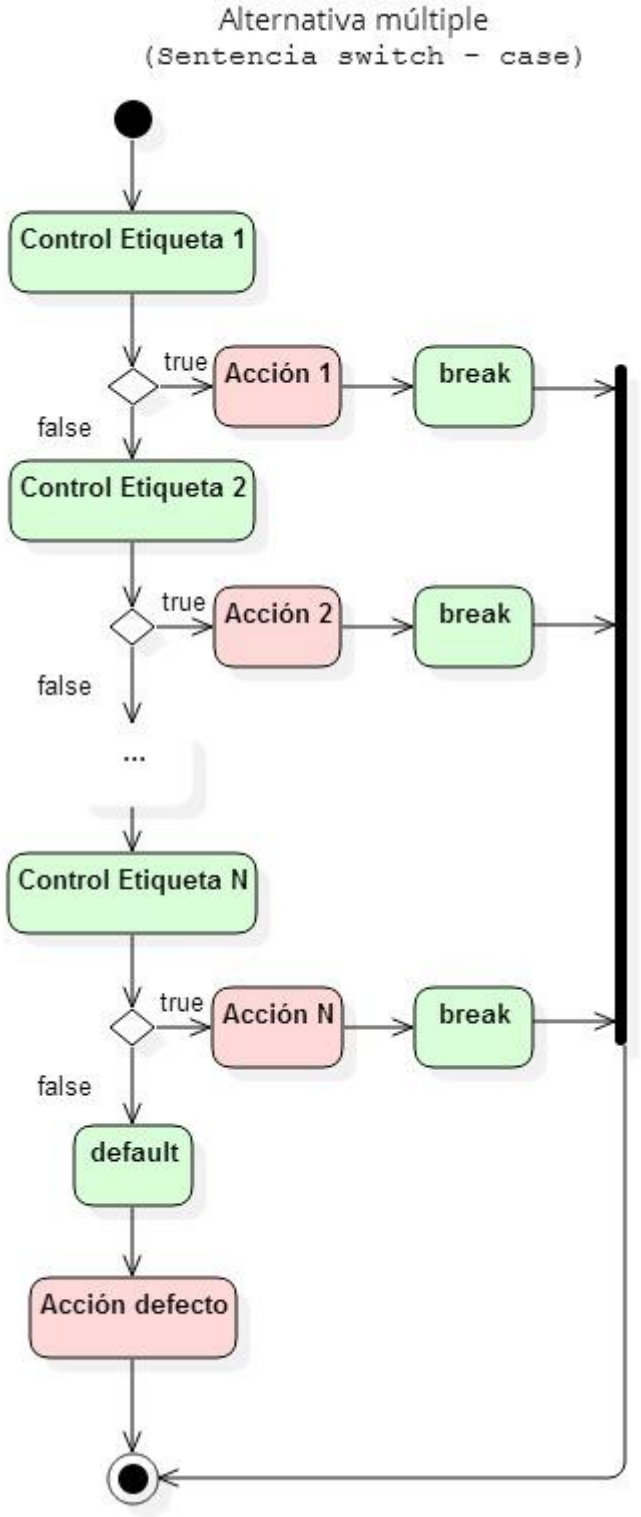
```
//...

//Pide la edad y determina si es mayor de edad

Scanner teclado = new Scanner(System.in);

System.out.print("¿Cuál es tu edad: ");
int edad = teclado .nextInt();

if (edad < 18) {
   System.out.println("¡Eres menor de edad !");
}
else {
   System.out.println("¡Eres mayor de edad !");
}
//...
```

Si se tienen que incluir varias instrucciones asociadas al **`if`** o **`else`**, es obligatorio utilizar las llaves, como en esta versión del ejemplo:

```
//...

//Pide la edad y determina si es mayor de edad

boolean acceso = false;
Scanner teclado = new Scanner(System.in);

System.out.print("¿Cuál es tu edad: ");
int edad = scanner.nextInt();

if (edad >= 18) {
	System.out.println("¡Eres mayor de edad !");
	acceso = true;
	System.out.println("Acceso permitido...");
}
else {
	System.out.println("¡Eres menor de edad !");
	System.out.println("Acceso no permitido...");
}

//...
```

Alternativa múltiple: switch - case

La sentencia `switch` proporciona una forma limpia de dirigir la ejecución a partes diferentes del código en base al valor de una variable o expresión. Esta es la forma general de la sentencia **`switch`**:




```
switch (<expresiónInt>) {
      case valor1:
            sentencias1;
            break;
      case valor2:
```

```
                    sentencias2;
                    break;
              case valorN:
                    sentenciasN;
                    break;
              default:
                    sentenciasDefault;
        }
```

Pueden aparecer cualquier número de etiquetas **`case`**.

El valor de la expresión se compara con cada uno de los *valores literales* de las etiquetas **`case`**. La *expresión* asociada un **`case`** debe ser un *valor literal* o una *variable* ***`final`*** o constante simbólica compatible con un **`int`**, lo que significa que sólo puede ser **`byte`**, **`short`**, **`char`**, **`int`**, `Byte`, `Short`, `Character`, `Integer` o de tipo **`enum`**.

A partir de Java 7 se admiten también `String`.

Una vez verificada la coincidencia con el valor de una *etiqueta `case`*; ignora el resto e intenta la ejecución en secuencia de todas las instrucciones hasta el final, a no ser que aparezca una sentencia `break` para terminar la secuencia y salir del `switch`.

Si ninguna de las *etiquetas* ***`case`*** verifica coincidencia con el valor de la *variable de control*, se ejecuta el bloque **`default`** que es opcional y siempre colocado al final.

```
//...

/* Pide un numero (entre 1 y 7) que representa el día de la semana
   y muestra su nombre. Si se introducen valores no válidos muestra
   un mensaje de error.
*/
Scanner teclado = new Scanner(System.in);
int dia;
String nombre;

System.out.print("Teclea el número del día de la semana: ");
dia = teclado.nextInt();

switch(dia) {
      case 1:
            nombre = "Lunes";
            break;
      case 2:
            nombre = "Martes";
            break;
      case 3:
            nombre = "Miércoles";
            break;
      case 4:
            nombre = "Jueves";
```

```
                break;
        case 5:
                nombre = "Viernes";
                break;
        case 6:
                nombre = "Sábado";
                break;
        case 7:
                nombre = "Domingo";
                break;
        default:
                nombre = "Número de día incorrecto... debe ser entre 1 y 7";
}
System.out.println(nombre);

//...
```

Decisión in-line

El `if` *in-line* tiene la siguiente estructura:

```
<expresiónBool> ? sentencia1 : sentencia2 ;
```

Se interpreta:

Si `expresiónBool` es **`true`** entonces se ejecuta `sentencia1`.

Si `expresiónBool` es **`false`** entonces se ejecuta `sentencia2`.

```
//...

// Pide un número y dice si es par o impar

Scanner teclado = new Scanner(System.in);

System.out.print("Teclea un número: ");
int num = teclado.nextInt();

int resto = num % 2;                    // resto de la división entera

// utiliza un if in-line y el operador == de comparación
String mensaje = (resto == 0) ? " Es par": " Es impar";

        /* equivalente
                if (resto == 0)
                        mensaje = " Es par";
                else
                        mensaje = " Es impar";
        */

// muestra
System.out.println(num + mensaje);
```

```
//...
```

Repetitivas o iterativas

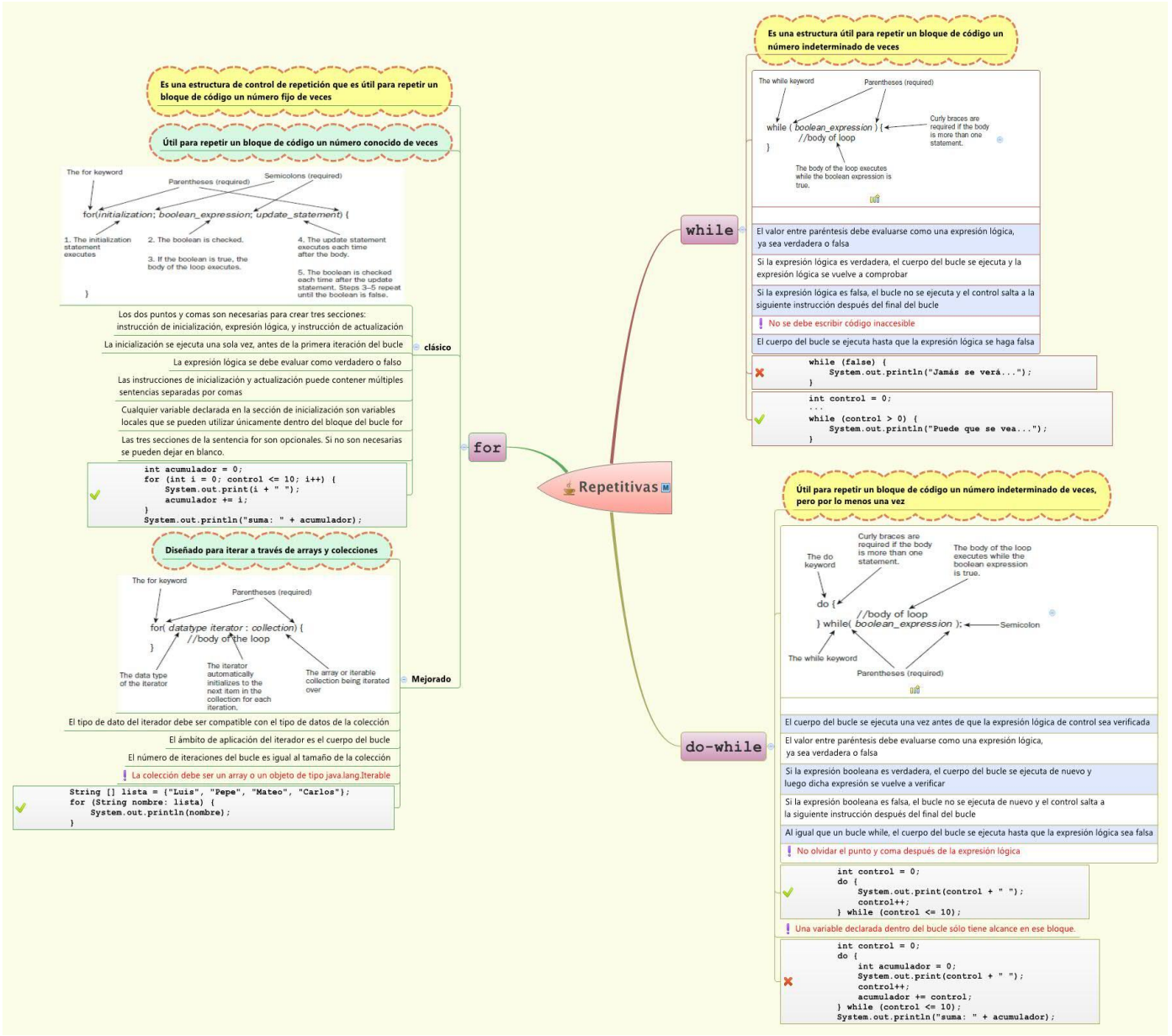
En Java, **un bucle es ejecutar repetidamente el mismo bloque de código mientras que se cumpla una condición de continuación**.

Hay cuatro aspectos en cualquier bucle:

- *Inicialización*.
  Hace referencia al código que establece las condiciones iniciales de un bucle.
- *Cuerpo*.
  Incluye y afecta a la serie de instrucciones que serán ejecutadas repetidas veces.
- *Actualización de Iteración*.
  Afecta a las instrucciones de código que se ejecutan después de cada ejecución del *cuerpo* y antes de intentar otro ciclo del bucle. Se actualizan, a menudo, contadores o variables implicadas en el *control de continuación*. Es siempre parte del cuerpo.
- *Control de continuación*.
  Fundamentalmente consiste en establecer una expresión booleana que se verifica en cada pasada del bucle para verificar si se debe continuar iterando o terminar.

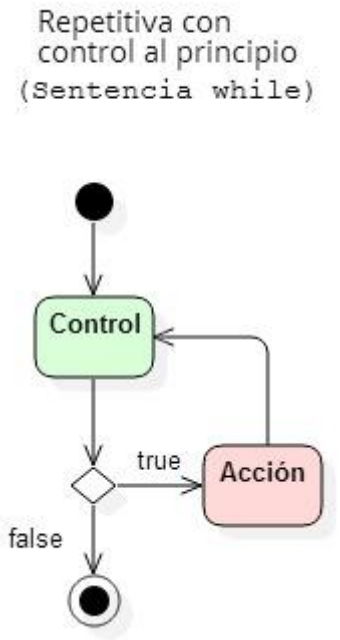
Java tiene tres construcciones para bucles:

- **`Repetitiva: while`**
- **`Repetitiva: do-while`**
- **`Repetitiva: for`**



Repetitiva: while

Es una estructura de flujo de programa, con control al principio, que permite repetir un bloque de código un número indeterminado de veces; según sea verdadera o falsa una expresión booleana que actúa de control.




```
while (<expresiónControl>) {
      sentencias1;
}
```

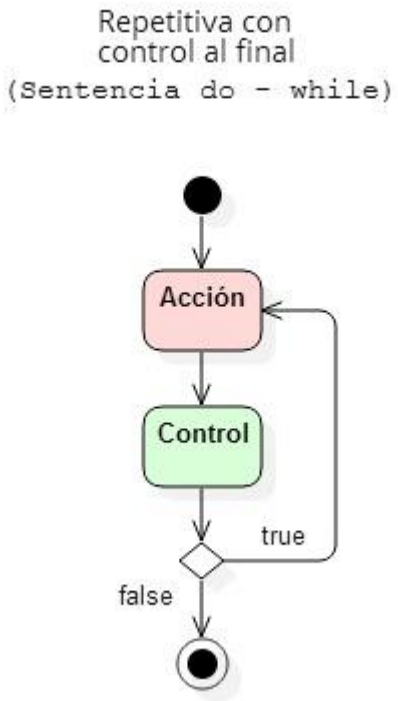
- Si la *expresión lógica de control* es verdadera, el *cuerpo del bucle* se ejecuta y la *expresión lógica de control* se vuelve a comprobar.
- El cuerpo del bucle se ejecuta mientras que la *expresión lógica de control* se mantenga verdadera.
- Si la *expresión lógica de control* es falsa, el bucle termina y el control de ejecución de programa salta a la siguiente instrucción después del final del bucle.

```
//...

// Muestra múltiplos de 3 menores de 1000

int multiplo = 3;                          //Inicialización
System.out.println("Múltiplos de 3 menores de 1000...");

while (multiplo <= 1000) {                 //Control de continuación
       System.out.println(multiplo);       //Cuerpo
       multiplo += 3;                      //Actualización de iteración
}

//...
```

Repetitiva: do-while

Es una estructura de flujo de programa, con control al final, que permite repetir un bloque

de código un número indeterminado de veces; asegurando la ejecución del cuerpo del bucle al menos una vez, incluso si la expresión booleana de *control* fuese `false` la primera vez.



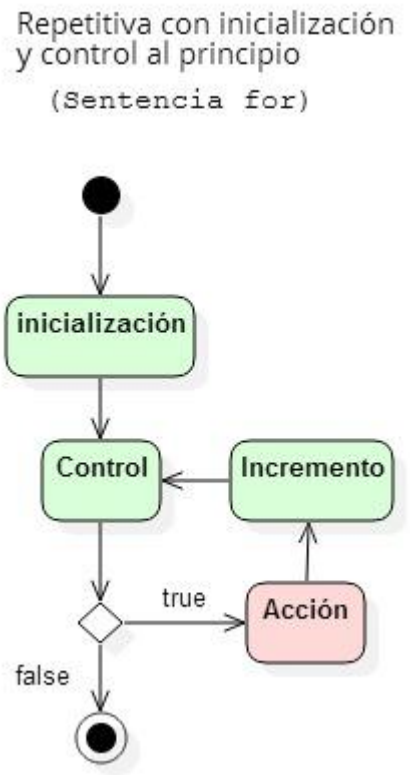

```
do {
      sentencias1;
} while (<expresiónControl>);
```

Si la expresión booleana que controla la continuación es verdadera, el *cuerpo del bucle* se ejecuta de nuevo, se actualizan las *variables de iteración* y vuelve a verificarse el *control de continuación*. Al igual que un bucle **`while`**, el cuerpo del bucle se ejecuta mientras que la *expresión lógica de control* sea verdadera.

```
//...
// Muestra números menores de 1000 que no son múltiplos de 5

int num = 1;                              //Inicialización

do {
      if (num % 5 != 0) {                 //Cuerpo
            System.out.println(num);
      }

num++;                                    //Actualización de iteración
}
while (i <= 1000);                        //Control de continuación

//...
```

Repetitiva: for

Es una estructura de flujo de programa, con control e inicialización al principio, que normalmente permite repetir un bloque de código un número conocido de veces.



Tiene dos formas, una clásica y otra renovada a partir de Java 5. La sintaxis clásica es:

```
for (<inicialización>; <expresiónControl>; <incremento> ) {
                sentencias1;
}
```

- La *inicialización* se ejecuta una sola vez, antes de la primera iteración del bucle.
- La *expresión lógica de control* se debe evaluar como verdadero o falso.
- Cualquier variable declarada en la sección de *inicialización* es una variable local que se pueden utilizar únicamente dentro del bloque del bucle **for**.
- La sección de *incremento* o *actualización* se ejecuta siempre después de la ejecución del cuerpo, antes de comprobar *expresión lógica de control.* Se pueden actualizar las variables que se considere oportuno; no sólo la implicada en el control.
- Las tres secciones de la sentencia **for** son opcionales. Si no son necesarias se pueden dejar en blanco manteniendo los **;**

```
//...
// Muestra 1000 números en filas de 20 como máximo

for (int i = 0; i < 1000; i += 20) {
```

```
        for (int j = i; j < 20 + i; j++) {
                System.out.print(j + "\t");
        }

        System.out.println("->");
}
//...
```

Las secciones de *inicialización* e *incremento* pueden contener múltiples sentencias separadas por comas.

```
//...
int a, b;
for (a = 1, b = 4; a < b; a++, b--) {
        System.out.println("a = " + a);
        System.out.println("b = " + b);
}

//...
```

El bucle anterior sólo se ejecuta dos veces y su salida es la siguiente:

```
a = 1;
b = 4
a = 2
b = 3
```

Otra forma es como bucle **for** *each* que tiene la siguiente sintaxis:

```
            for (<iterador> : <estructuraDeDatos> ) {

                  sentencias1;
            }
```

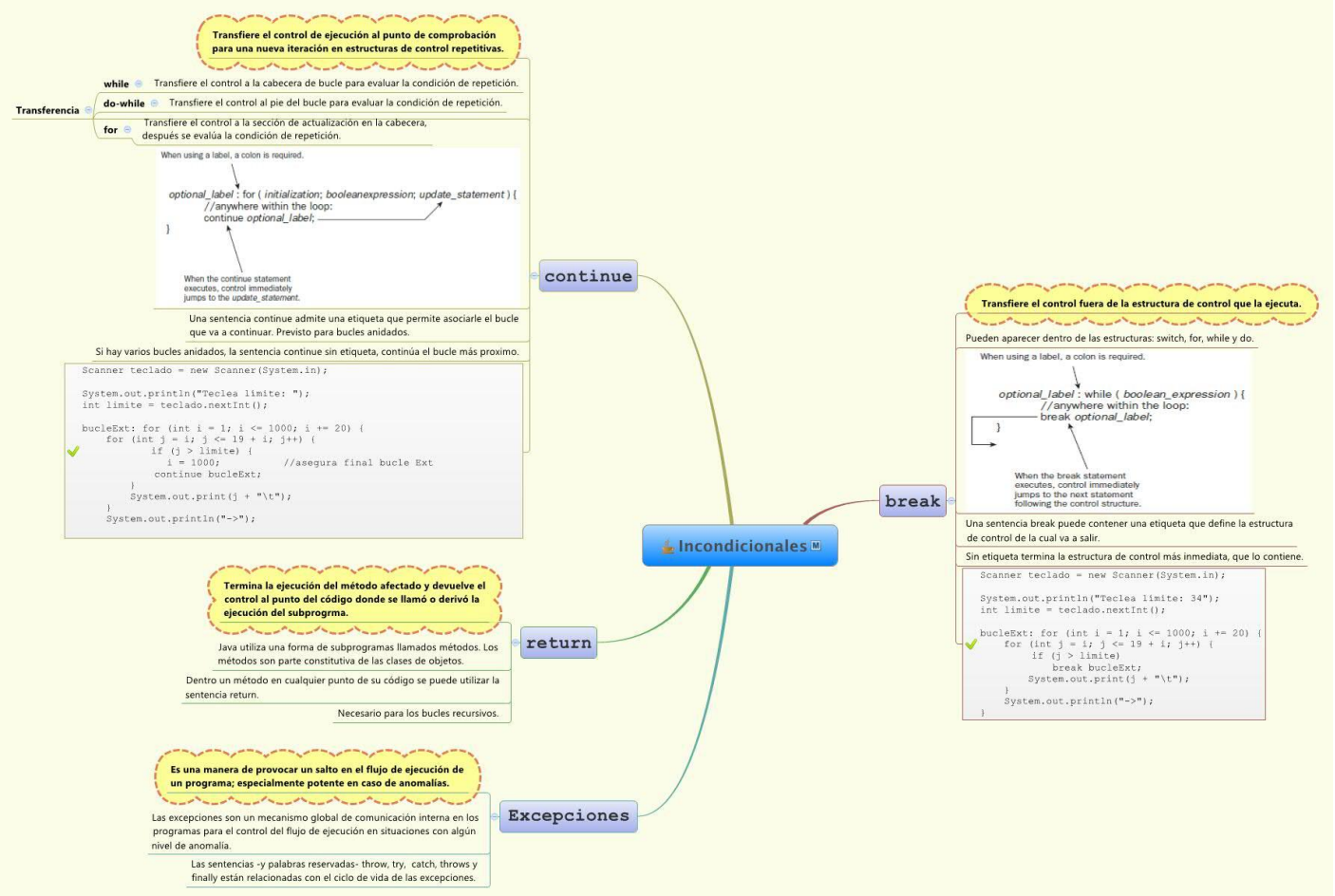
Diseñado para iterar sobre arrays y colecciones de datos. Deben cumplirse las siguientes condiciones:

- El tipo de dato del iterador debe ser compatible con el tipo de datos de la colección o array.
-
- El ámbito de aplicación del iterador es el cuerpo del bucle.
-
- El número de iteraciones del bucle es igual al tamaño de la *colección* o *array*.
-
- La colección debe ser un *array* o un objeto de tipo `java.lang.Iterable`

```
//...
// Muestra una lista de nombres
```

```
String [] lista = {"Luis", "Pepe", "Mateo", "Carlos"};

for (String nombre: lista) {
      System.out.println(nombre);
}

//...
```

Incondicionales



break

Transfiere el control de ejecución de programa fuera de la *estructura de control* que la ejecuta. Termina la estructura de control y la ejecución prosigue pasado el final del bloque de la *estructura de control* interrumpida.

El término **`break`** se refiere al acto de salir de un bloque de código. Pueden aparecer asociadas a las estructuras: **`switch`**, **`for`**, **`while`** y **`do-while`**.

La sentencia **`break`** de Java está también prevista para los casos en los que se necesita, arbitrariamente, ejecutar una porción de código. Una sentencia **`break`** puede ir con una *etiqueta*

que define la estructura de control de la cual va a salir. Sin etiqueta termina la estructura de control más inmediata, que lo contiene.

```
//...
/* Muestra 1000 números en filas de 20 como máximo
   pregunta límite para mostrar
   Utiliza break con una etiqueta
   Sin etiqueta no terminaría bien, sólo rompería bucle interno
*/
Scanner teclado = new Scanner(System.in);

System.out.println("Teclea límite: ");
int limite = teclado.nextInt();

bucleExt: for (int i = 0; i < 1000; i += 20) {
	for (int j = i; j < 20 + i; j++) {
		if (j > limite) {
			break bucleExt;
		}
		System.out.print(j + "\t");
	}
	System.out.println("->");
}

//...
```

continue

Transfiere la ejecución al *punto de control* de una estructura repetitiva para intentar una nueva iteración. Es aplicable a todas las estructuras repetitivas.

- En los bucles **`while`** transfiere el control a la cabecera de bucle para evaluar la condición de repetición.
- En los bucles **`do-while`** transfiere el control al pie del bucle para evaluar la condición de repetición.
- En el bucle **`for`** transfiere el control a la *sección de actualización* en la cabecera, después se evalúa la condición de repetición.
- Una sentencia **`continue` admite una etiqueta que permite asociar el bucle que va a continuar**; esto está **previsto para *bucles anidados***. Sin etiqueta, continua el bucle más próximo.

```
//...
// Muestra 1000 números en filas de 20 como máximo
// pregunta límite para mostrar
// Utiliza continue con una etiqueta
// Sin etiqueta continuaría el bucle interno

Scanner teclado = new Scanner(System.in);

System.out.println("Teclea límite: ");
int limite = teclado.nextInt();
```

```
bucleExt: for (int i = 0; i < 1000; i += 20) {
       for (int j = i; j < 20 + i; j++) {
              if (j > limite) {
                     i = 1000;                          //asegura final bucle Ext
                     continue bucleExt;
              }
              System.out.print(j + "\t");
       }
       System.out.println("->");
}

//...
```

return

Java utiliza una forma de subprogramas llamados *métodos*. Los métodos forman parte constitutiva de las clases de objetos. Dentro un método en cualquier punto de su código se puede utilizar la sentencia **`return.`**

La sentencia **`return` detiene la ejecución del método de manera incondicional y devuelve el control de ejecución de programa al punto donde fue llamado el método**. La utilización de **`return`** se tratará en un capítulo aparte.

Recursividad

La recursividad es un medio para implementar, implícitamente, bucles en la ejecución de los programas. **Aprovecha la propiedad que poseen los métodos de poderse llamar a sí mismos**.

La recursividad es una especialidad dentro de la *algoritmia* que proporciona soluciones relativamente simples en muchos casos. Se hará una introducción cuando se trate el capítulo correspondiente a los *subprogramas* y los *métodos*.

Excepciones

Otra manera de provocar un salto en el flujo de ejecución de un programa es utilizar las excepciones. Las sentencias -y palabras reservadas- **`throw`**, **`try`**, **`catch`**, **`throws`** y **`finally`** están relacionadas con el ciclo de vida de las excepciones.

Las excepciones son un mecanismo global de comunicación interna en los programas para encaminar el flujo de ejecución cuando se producen anomalía y errores. La gestión de excepciones en todos sus aspectos se tratarán en un capítulo aparte.

Asertos

Un aserto es una instrucción que contiene una expresión booleana que será verdadera, con

certeza, en un momento dado de la ejecución.

Es un mecanismo disponible en los lenguajes de programación que permite verificar lo que se considera invariante -seguro- en el funcionamiento de un programa. Los asertos permiten acotar los puntos donde hay posibilidad de errores. Son empleados frecuentemente para la depuración y pruebas de los programas.

Ejercicios

1. (*) Escribe un programa simple que, utilizando sólo las instrucciones de salida:

```
System.out.print("* ");
System.out.print(" ");
System.out.println();
```

Muestre el siguiente patrón.

```
* * * * * * * * *
 * * * * * * * * *
* * * * * * * * *
 * * * * * * * * *
* * * * * * * * *
 * * * * * * * * *
* * * * * * * * *
 * * * * * * * * *
```

Observa que una llamada al método `System.out.println()` sin argumento hace que el programa salte una línea en la salida.

2. (*) Escribe un programa simple que muestre en pantalla los múltiplos de 2 (es decir: 2, 4, 8, 16, 32, 64, etc.). El bucle utilizado no debe terminar (es decir, se debe crear un bucle infinito). ¿Qué ocurre cuando ejecuta este programa?

3. (*) Escribe un programa simple que lea tres valores distintos de cero introducidos por el usuario, y que determine e imprima si podrían representar los lados de un triángulo.

4. (*)Escribe un programa simple que lea tres enteros distintos de cero, determine e imprima si estos enteros podrían representar los lados de un triángulo rectángulo.

5. (**) Escribe un programa simple que reciba como entrada un entero que contenga sólo 0s y 1s (es decir, un entero binario), y que muestre su equivalente decimal.

> Sugerencia: El operador módulo permite obtener los dígitos del número binario en sucesivas pasadas de derecha a izquierda. En el sistema decimal, el dígito más a la derecha tiene un valor posicional de 1 y el siguiente dígito a la izquierda tiene un valor posicional de 10, después 100, después 1000, etc. El número decimal 234 puede interpretarse como 4 * 1 + 3 * 10 + 2 * 100. En el sistema binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la izquierda tiene un valor posicional de 2, luego 4, luego 8, etcétera. El equivalente decimal del número binario 1101 es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, o 1 + 0 + 4+ 8, o 13.

6. (**) Una compañía desea transmitir datos a través del teléfono, pero le preocupa que sus teléfonos puedan estar intervenidos. Escriba un programa simple que cifre los datos, de manera que éstos puedan transmitirse con más seguridad. Todos los datos se transmiten como enteros de cuatro dígitos.

El programa debe leer un entero de cuatro dígitos introducido por el usuario y cifrarlo de la siguiente manera:

- Reemplazando cada dígito con el resultado de sumar 7 al dígito y obtener el módulo de 10.
- Luego intercambiar el primer dígito con el tercero, e intercambiar el segundo dígito con el cuarto.
- Después imprima el entero cifrado.

7. (**) Escribe un programa simple que reciba como entrada un entero de cuatro dígitos cifrado con el programa del ejercicio anterior, y que lo descifre para formar el número original.

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en linea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- CARRERES, J. *Manual de Java*. [en línea] http://www.oocities.org/collegepark/quad/8901/indice.html
- DEITEL, P. J. y DEITEL, H. M. *Cómo programar en Java.* Ed.Pearson Educación, 7ª edición. Méjico, 2008. ISBN: 978-970-26-1190-5