MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN

Estructuras de control

3

ÍNDICE

/ 1. Introducción y contextualización práctica 3

/ 2. Diagramas de flujo 4

/ 3. Sentencias de selección 4

3.1. Condicional if-else 5

3.2. Condicional switch 6

/ 4. Sentencias de repetición 6

4.1. Bucle while 7

4.2. Bucle do-while 8

4.3. Bucle for 8

4.4. Equivalencia entre bucles 9

/ 5. Condiciones múltiples 10

/ 6. Caso práctico 1: "¿Cómo puedo organizar el código?" 11

/ 7. Caso práctico 2: "¿Condición múltiple o anidamiento?" 12

/ 8. Sentencias de salto 12

8.1. Instrucciones goto, break, continue y return 13

/ 9. Resumen y resolución del caso práctico de la unidad 14

/ 10. Bibliografía 14

OBJETIVOS

Reconocer las estructuras de selección.

Reconocer las estructuras de repetición.

Reconocer las estructuras de salto.

Conocer cuándo utilizar la estructura de selección adecuada.

Conocer cuándo utilizar la estructura de repetición adecuada

/ 1. Introducción y contextualización práctica

En este tema hablaremos sobre las diferentes estructuras de control que existen y cuál es su sintaxis correcta para poder utilizarlas.

También vamos a hablar sobre la equivalencia entre sentencias de control repetitivas, lo cual nos mostrará la gran flexibilidad de éstas a la hora de resolver un problema.

Por último, hablaremos sobre las instrucciones de salto y de cómo tener precaución a la hora de utilizarlas únicamente en su contexto correcto.

Todos estos conceptos los iremos desarrollando aún más en todos los temas restantes, así que no te preocupes, que tenemos mucho tiempo por delante.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

Fig. 1. El camino que sigue un programa no es siempre el mismo.

Audio Intro. "Las instrucciones de control".
https://bit.ly/3cx2QFU

/ 2. Diagramas de flujo

Un diagrama de flujo es la representación gráfica de un algoritmo o proceso que muestra la secuencia lógica o los pasos que se deben llevar a cabo para realizar una tarea. Para que sea útil debe proporcionar una información clara, ordenada y concisa de todos los pasos a seguir.

Estos diagramas utilizan símbolos con significados definidos que representan los pasos del algoritmo, y marcan el flujo de ejecución mediante flechas que conectan los puntos de inicio y de fin del proceso.

En la figura se pueden ver los símbolos utilizados en este tipo de diagramas.

Un diagrama de flujo es útil en todo aquello que necesite una previa organización antes de su desarrollo, por ejemplo, en la realización de un programa informático, donde es imprescindible primero establecer los pasos a seguir, independientemente del lenguaje de programación que usemos después.

Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Linea de Flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada / Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida
	Proceso	Representa cualquier tipo de operación
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso

Fig. 2. Principales símbolos utilizados en los diagramas de flujo.

Una vez que tenemos nuestro diagrama de flujo solo tendremos que conocer las órdenes del lenguaje que realizan esas tareas que se especifican en el diagrama y codificarlo en el lenguaje que estemos utilizando.

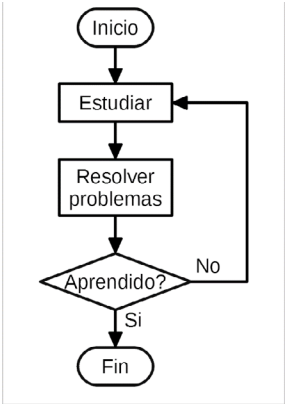

Fig. 3. Ejemplo de diagrama de flujo.

/ 3. Sentencias de selección

Como hemos visto en los temas anteriores, las sentencias son líneas de código que componen un programa y que se ejecutan secuencialmente, es decir, todas las veces que se ejecute el programa, siempre se ejecutarán las instrucciones en el mismo orden y proporcionarán el mismo resultado.

Sin embargo, esto no siempre es así, ya que un programa puede ejecutarse varias veces y proporcionar resultados distintos según los datos de entrada que se le proporcionen. Esto es debido a las estructuras de control de flujo.

Mediante estas estructuras podremos hacer que nuestros programas sigan un camino u otro dependiendo del valor de un dato, pudiendo, por ejemplo, mostrar un error por pantalla si ocurre cierta condición.

Un concepto muy importante con respecto a las estructuras de control es el de bloque de código. Un bloque nos va a indicar dónde podemos escribir código respetando la estructura del lenguaje de programación que utilicemos. Los bloques van a estar limitados por una serie de delimitadores, también especificados por el lenguaje de programación que se use, por ejemplo, en Java los limitadores de bloques de código serán las llaves ({ }), mientras que en Python los limitadores de bloque son los saltos de línea y las tabulaciones.

Disponemos de las siguientes estructuras de selección:

- **Sentencias if**: Nos permiten decidir si queremos ejecutar o no un fragmento de código o no dependiendo de si se cumple una cierta condición.
- **Sentencias if-else:** Similar al anterior, permiten ejecutar un bloque de código si se cumple una condición u otro si no se cumple.
- **Sentencias switch:** Nos permite ejecutar un bloque diferente de código para cada valor posible que pueda tomar una expresión.

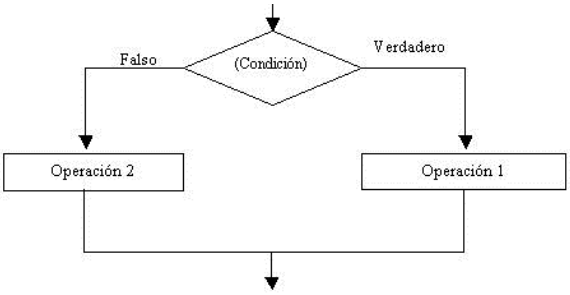

Fig. 4. Estructura if-else.

3.1. Condicional if-else

La sentencia *if-else* nos **permite ejecutar un bloque de código u otro según se cumpla una cierta condición**. Esta sentencia evalúa el valor de la condición y ejecutará el bloque correspondiente según la condición sea verdadera o falsa. Existen tres combinaciones principales:

- **If:** Si se cumple la condición, se ejecuta el bloque de código indicado.
- **If-else:** Si se cumple la condición, se ejecuta el primer bloque de código; en caso contrario, se ejecuta el bloque asociado al else.
- **If-else-if:** Si se cumple la condición, se ejecuta el primer bloque de código; en caso contrario se evalúa la segunda condición y se ejecuta su bloque asociado si es preciso. A esta combinación también se le puede añadir un else al final que se ejecutará si no se cumple ninguna de las condiciones anteriores.

Uno de los aspectos cruciales en este tipo de estructuras es diseñar bien las condiciones, ya que es un punto del código donde se suelen acumular los errores y se hacen difíciles de detectar.

En el ejemplo de la figura podemos observar cómo el programa mostrará un mensaje dependiendo de la condición que se cumpla.

```
if (hora < 12) {
  System.out.println("Buenos días.");
} else if (hora < 20) {
  System.out.println("Buenas tardes.");
}  else {
  System.out.println("Buenas noches.");
}
```

Fig. 5. Ejemplo de if-else.

3.2. Condicional switch

Al igual que las sentencias *if-else*, las sentencias *switch* son bloques que nos permitirán ejecutar código según se cumpla una condición.

Existen situaciones donde nos encontraremos con la necesidad de utilizar múltiples bloques if-else-if que hacen que el código sea poco legible. En estos casos puede ser una mejor opción utilizar una estructura de selección switch. Hay que tener en cuenta que no podremos hacer comparaciones, solamente comprobar si el valor del switch es igual a cada uno de los casos que indiquemos.

La diferencia fundamental está en que **estas son sentencias multiramificadas**, es decir, en una **sentencia if-else solo teníamos dos caminos posibles**, cuando la condición fuese verdadera o falsa, teniendo que usar el anidamiento en caso de querer más ramificaciones. Por el contrario, en una **sentencia switch podremos poner tantos casos como queramos** sin tener que usar el anidamiento. Esto no quiere decir que las sentencias *switch* no se puedan anidar, también se puede.

Quizás la característica más significativa de las sentencias *switch* sea la posibilidad de definir en última instancia un caso especial que se ejecutará cuando no se cumplan ninguna de las propiedades anteriores.

Es importante recordar que siempre que se termine un caso del bloque *switch* hay que terminar con la instrucción *break*.

Por ejemplo, en la siguiente figura podemos ver el código necesario para identificar los días de la semana según su número, mostrando un error en caso de introducir un día incorrecto.

```
switch( numero )
{
    case 1:
        System.out.println("Lunes");
        break;
    case 2:
        System.out.println("Martes");
        break;
    case 3:
        System.out.println("Miércoles");
        break;
    case 4:
        System.out.println("Jueves");
        break;
    case 5:
        System.out.println("Viernes");
        break;
    case 6:
        System.out.println("Sábado");
        break;
    case 7:
        System.out.println("Domingo");
        break;
    default:
        System.out.println("Error. Día incorrecto.");
        break;
}
```

Fig. 7. Ejemplo de bloque switch.

/ 4. Sentencias de repetición

Las sentencias de repetición, o también conocidas como bucles, **son sentencias que nos van a permitir repetir la ejecución de un bloque de código completo un número determinado de veces**.

Podremos hacer también que un cierto código se repita automáticamente un número de veces, sin tener que preocuparnos por repetir el código escrito en el programa.

El número de veces que se repetirá el bloque completo de código **vendrá dado por una cierta condición (o por varias)** que se irá comprobando cada vez que el bucle vaya a dar una nueva vuelta de repetición.

Al igual que puede ocurrir con las sentencias condicionales, hay que tener mucho cuidado con las condiciones a la hora de crear un bucle, ya que puede darse que la condición siempre sea cierta y tengamos, lo que se conoce, como un bucle infinito.

Siempre que tengamos un bucle infinito tendremos que revisar la condición del mismo, ya que el fallo estará ahí.

Otro error que podemos cometer a la hora de crear el bucle es que su condición sea siempre falsa, por lo cual **no se entrará al bloque nunca, y no se ejecutará el bloque de código**. En este caso, el fallo probablemente también esté en la propia condición.

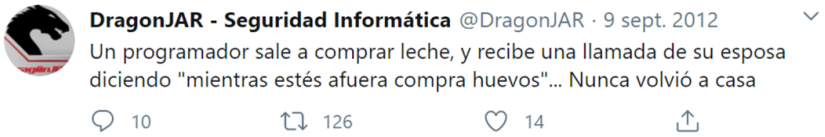

Fig. 8. Si no se diseñan bien las condiciones aparecerán bucles infinitos.

4.1. Bucle while

El bucle *while* es una estructura de repetición que nos permitirá repetir un bloque de código 0 o más veces.

Está controlado por una condición de parada y su funcionamiento es el siguiente:

1. Comprueba si la condición es verdadera o falsa.
2. Si la condición es verdadera ejecuta el bloque de código.
    - Cuando termina de ejecutar el bloque vuelve al punto uno.
3. Si la condición es falsa no entra al bucle y ejecuta la siguiente instrucción.

La condición que tenga el bucle, se ha de devolver un valor booleano (verdadero o falso), para que así, éste pueda realizar la comprobación de la misma y continuar si su valor es verdadero o no entrar y continuar con la siguiente instrucción si su valor es falso.

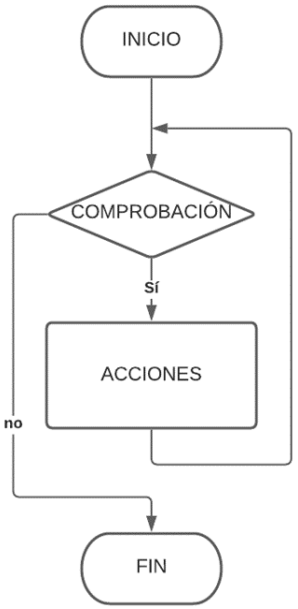

Fig. 9. Diagrama de flujo de un bucle while.

A continuación, podemos ver un ejemplo de bucle *while* que mostrará los números del 1 al 50.

```
int contador = 1;
while( contador <= 50 )
{
    System.out.println("El número es: " + contador);
    contador++;
}
```

Fig. 10. Ejemplo de bucle while.

Si comparamos el bucle while y el do-while, vemos que el primero comprueba y luego ejecuta el cuerpo del bucle, mientras que el segundo ejecuta el cuerpo del bucle y luego comprueba si debe salir o continuar ejecutando el bucle. Esta es la razón por la que el do-while se ejecuta al menos una vez, mientras que while puede no llegar a ejecutarse nunca.

4.2. Bucle do-while

El bucle *do-while* **es una estructura de repetición que nos permitirá repetir un bloque de código 1 o más veces**.

A diferencia del bucle *while*, el *do-while* como mínimo va a ejecutarse una vez.

Está controlado por una condición de parada y su funcionamiento es el siguiente:

1. Entra al bucle y ejecuta el bloque de código.
2. Al terminar la ejecución del bloque comprueba si la condición es verdadera o falsa.
   - Si la condición es verdadera vuelve al punto 1.
   - Si la condición es falsa no entra al bucle y ejecuta la siguiente instrucción

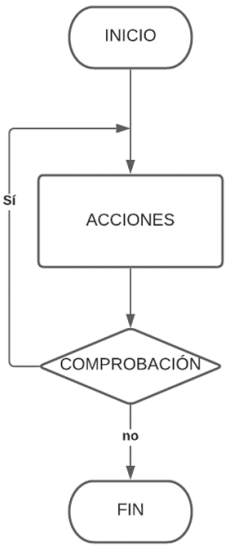

Fig. 11. Diagrama de flujo de un bucle do-while.

Al igual que ocurre con el bucle *while*, la condición que tenga el bucle **ha de devolverse en un valor booleano, verdadero o falso**, para que así el bucle pueda realizar la comprobación de la misma y continuar si su valor es verdadero o no entrar y continuar con la siguiente instrucción si su valor es falso. A continuación, podemos ver un ejemplo de bucle *do-while* que mostrará los números del 1 al 50.

```
int contador = 1;
do
{
    System.out.println("El número es: " + contador);
    contador++;
} while( contador <= 50 );
```

Fig. 12. Ejemplo de bucle do-while.

Si comparamos el bucle while y el do-while, vemos que el primero comprueba y luego ejecuta el cuerpo del bucle, mientras que el segundo ejecuta el cuerpo del bucle y luego comprueba si debe salir o continuar ejecutando el bucle. Esta es la razón por la que el do-while se ejecuta al menos una vez, mientras que while puede no llegar a ejecutarse nunca.

4.3. Bucle for

El bucle *for* **es una estructura de repetición que nos permitirá repetir un bloque de código una cantidad de veces ya conocida y fija**.

Está controlado por tres partes:

1. Un contador, inicializado normalmente a cero.
2. Una condición de parada, que tendrá que ver con el contador.
3. Un incremento del contador, puede ser de uno en uno, adelante, hacia atrás, o de un número x en x.

Su funcionamiento es el siguiente:

1. Se crea e inicializa el contador.
2. Se comprueba la condición de parada.
3. Si la condición de parada es verdadera se ejecuta el bloque de código.
    - Al terminar la ejecución del bloque de código se realiza el incremento del contador y se vuelve al punto 2.
4. Si la condición de parada es falsa se sale del bucle y ejecuta la siguiente instrucción del programa.

Podemos crear un bucle *for* sin poner ninguna de las tres partes que lo controlan, es ese caso, tendremos un bucle infinito.

Este bucle va a ser muy útil cuando sepamos de antemano las veces que vamos a repetir un bloque de código, esto no quiere decir que haya que usar el bucle *for* forzosamente para esto, como veremos más adelante.

Vídeo 1. "Tabla de multiplicar con bucle for".
https://bit.ly/2U6vV4D

4.4. Equivalencia entre bucles

Hemos visto que tenemos tres tipos de bucles: el bucle *while*, el *do-while* y el *for*, pero, entonces si vamos a realizar un programa donde necesitemos uno o varios bucles, ¿cuál elegimos?

La respuesta es sencilla, **podemos utilizar el que queramos ya que los bucles son equivalentes entre ellos**, esto quiere decir que cualquier ejercicio que contenga uno o varios bucles podrá realizarse tanto con un *while*, como con un *do-while*, como con un *for*.

Evidentemente habrá que hacer algunos cambios si utilizamos un bucle u otro, como es lógico, pero en la práctica el resultado que obtengamos debería ser el mismo.

Esto conlleva que podamos elegir un bucle y trabajar siempre con él. Un desarrollador en concreto puede sentirse más cómodo con el bucle while por ejemplo, y puede decidir usarlo siempre como primera opción. El hecho de que podamos elegir, nos permitirá decidirnos por uno u otro, ya que nos encontraremos problemas donde será más sencilla una implementación con un bucle determinado que con los demás.

Esto se aprenderá con la práctica, pero las siguientes ideas nos servirán a modo orientativo mientras tanto:

- **For:** cuando tengamos un número de iteraciones conocidas, por ejemplo, sumar los 100 primero números enteros.
- **While:** cuando el número de iteraciones sea desconocido porque dependa de una condición, por ejemplo, solicitar números al usuario hasta que sumen un valor determinado. No tendría sentido utilizar un for porque no sabemos cuántos números tendremos que leer
- **Do-While:** cuando se tenga que ejecutar al menos una vez, por ejemplo, solicitar números al usuario hasta que introduzca el 0. Puede que lo introduzca a la primera o a la cuarta vez, pero tendrá que introducir al menos uno.

Cuando tengamos algo más de experiencia veremos que hay casos que se prestan a resolverse muchísimo más fácil con un bucle determinado, pero, en el nivel en el que nos encontramos todavía, no será tan obvio ya que los problemas no serán tan complicados.

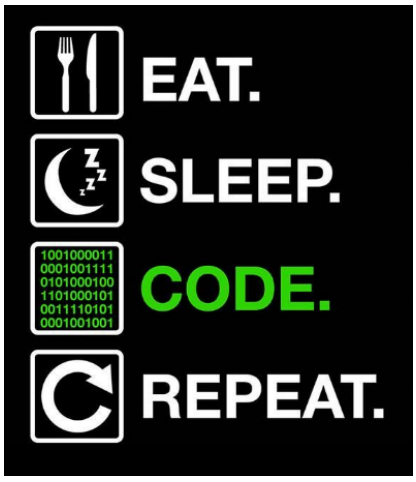

Fig. 13. Dependiendo de la tarea será mejor utilizar uno u otro tipo de bucle.

Audio 1. "Las equivalencias entre bucles".
https://bit.ly/2yZKSOP

/ 5. Condiciones múltiples

Hasta ahora, hemos visto que tanto las estructuras condicionales como las estructuras repetitivas utilizan condiciones para comprobar si se tienen que ejecutar o no.

Estas condiciones pueden ser simples (una única condición) o múltiples (varias condiciones).

Para poder operar con más de una condición a la vez utilizaremos los operadores lógicos vistos en la unidad anterior:

- **Operador AND**, representado como **&&**.
- **Operador OR**, representado como **||**.
- **Operador NOT**, representado como **!**.

Recordemos que el operador AND devolverá verdadero solo si todas las condiciones aplicadas son verdaderas, **el operador OR devolverá falso sólo si todas las condiciones aplicadas son falsas**, es decir, si hay solo una verdadera ya será verdadero, y el operador NOT devolverá el valor contrario a la condición aplicada.

La gran ventaja que nos brindan estos operadores lógicos es que **podremos operar tantas condiciones con ellos como queramos**, podemos operar con 2, con 3, con 4 condiciones...

El hecho de operar con varias condiciones y con varios operadores lógicos al mismo tiempo **puede hacer que la condición final sea un poco complicada de ver y entender y llevarnos a un fallo que no comprendamos**, así que, de primeras, tendremos que utilizar las condiciones múltiples con mucho cuidado.

Este tipo de condiciones también se pueden utilizar en los bucles, no es algo exclusivo de los condicionales.

```java
public static void main(String[] args)
{
    Scanner teclado_int = new Scanner(System.in);
    int numero;

    System.out.println("Introduce un número");
    numero = teclado_int.nextInt();

    if (numero >= 0 && numero % 2 == 0)
    {
        System.out.println("La condición se cumple.");
    }
}
```

Fig. 14. Ejemplo de condición múltiple que comprueba que un número sea mayor o igual que 0 y múltiplo de 2 a la vez.

/ 6. Caso práctico 1: "¿Cómo puedo organizar el código?"

Planteamiento. Pilar y José están realizando un programa en el que tienen que utilizar varios bloques *if-else* anidados. Este ejercicio les está siendo muy complicado, ya que no entienden bien dónde queda cada instrucción en cada bloque. Ellos están escribiendo el código como siempre, comenzando a escribir todo desde el principio de la línea.

Nudo. ¿Qué piensas sobre ello? ¿Crees que es apropiado escribir los códigos de los bloques de esa forma?

Desenlace. Cuando utilicemos bloques de código, a partir de ahora, vamos a tener a las estructuras de control como algo básico en cualquier programa, por lo que hay que tener en cuenta una regla muy importante: **Tabular el código**. Cuando tenemos un bloque de código, por ejemplo, el de una instrucción *if* y el de una instrucción *else*, vamos a poder diferenciarlo muy fácilmente si tabulamos las instrucciones que están dentro del bloque. En el caso de que tengamos más de un bloque de código anidado tendremos que aplicar tantas tabulaciones como bloques tengamos, dejando así el código muy fácil de leer y entender. Observa las tabulaciones del siguiente ejemplo:

```java
if( numero >= 0 )
{
    // Bloque del if número 1
    if( numero == 0 )
    {
        // Bloque del if número 2
        System.out.println("El número es cero.");
    }
    else
    {
        // Bloque del else del if 2
        System.out.println("El número es mayor que cero.");
    }
}
else
{
    // Bloque el else del if 1
    System.out.println("El número menor que cero.");
}
```

Fig. 15. Bloques if-else anidados y correctamente tabulados.

/ 7. Caso práctico 2: "¿Condición múltiple o anidamiento?"

Planteamiento: Pilar y José están realizando un ejercicio que consiste en comprobar que un usuario y una contraseña sean correctos, es decir, una validación. Para ello Pilar cree que la mejor forma de hacerlo es con dos *if* anidados, cada uno con su propia condición, mientras que José piensa que sería más fácil hacerlo con un único *if* y una condición múltiple.

Nudo: ¿Qué piensas al respecto? ¿Crees que tiene razón Pilar o que la razón la tiene José? ¿Podrían tener razón los dos? ¿Acaso hay más de una forma de realizar el ejercicio?

Desenlace: Cuando en un programa necesitemos comprobar más de una condición, como es el caso del ejercicio de validación de usuario y contraseña que están realizando Pilar y José, no existe una manea única de hacerlo, sino que existen varias formas, así que, respondiendo la duda de nuestros amigos, los dos tienen razón, la comprobación se podrá hacer tanto con dos *if* anidados, uno para comprobar si el usuario es correcto y otro para la contraseña, o con un único *if* con una condición múltiple unida mediante el operador lógico AND, ya que tienen que ser verdad las dos validaciones.

Fig. 16. Diferentes caminos para llegar a la misma solución.

En este caso en concreto podremos poner una condición múltiple que compruebe si el usuario y (operador AND) la contraseña son correctos:

if (usuario.equals("valor") && contrasena.equals("valor"))

/ 8. Sentencias de salto

Las sentencias de salto en programación **permiten pasar el control de ejecución de un punto a otro del programa de forma incondicional**.

Estas sentencias, se utilizaban en los primeros lenguajes de programación, como el Ensamblador, el Pascal... y permitían emular el funcionamiento de las sentencias condicionales o repetitivas que tenemos hoy día.

Actualmente, es totalmente desaconsejable su uso, salvo muy ciertas ocasiones que veremos más adelante. Por ejemplo, en Java, en las sentencias *switch* al terminar un caso tendremos que utilizar la sentencia de salto *break* obligatoriamente, ya que así lo dice la sintaxis del lenguaje.

La razón de no aconsejar su uso en programas de hoy en día es debido a que provocan una mala estructuración del código fuente, incrementando la complejidad del mismo, ya que no es sencillo seguir el flujo del programa, y por lo tanto dificultando su depuración.

Como hemos adelantado antes, en lenguajes como el Ensamblador, eran totalmente necesarias estas instrucciones, ya que este no poseía el concepto de estructuras condiciones y repetitivas como pasa con los lenguajes que tenemos hoy. Esto hacía que este lenguaje fuera muy complicado de aprender, leer, entender y modificar.

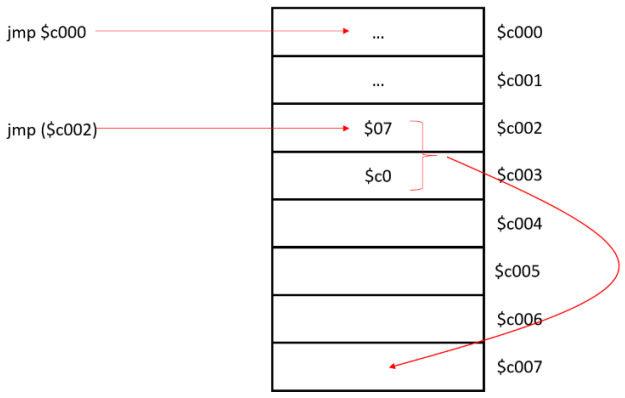

Fig. 17. Ejemplo de instrucciones de salto en ensamblador.

8.1. Instrucciones goto, break, continue y return

Existen varias instrucciones de salto, como son:

- Instrucción *goto*
- Instrucción *break*
- Instrucción *continue*
- Instrucción *return*

La instrucción **goto nos permite transferir el control de ejecución del programa a otro punto identificado por una etiqueta previamente creada**. Esta instrucción hacía que los programas fueran un auténtico caos, haciendo que la modificación de los mismos para mejorarlos fuera prácticamente imposible.

La instrucción *break* **es utilizada para salir de un condicional o de un bucle de forma abrupta, pudiendo provocar fallos en los programas por el hecho de usarla incorrectamente**. En Java la podemos utilizar, pero solo debemos hacerlo en las instrucciones switch.

La instrucción *continue* se utiliza en los bucles y su funcionamiento consiste en hace que el bucle pase a la siguiente vuelta directamente, aumentando así su contador.

La instrucción *return* se utiliza en los procedimientos que veremos más adelante, y su funcionamiento consiste en devolver el control de ejecución al punto donde se llamó al procedimiento.

En Java existen break, continue y return. Esta última sirve para devolver valores al volver de la invocación de funciones, mientras que las otras dos rompen el flujo de los bucles, como se ilustra en la figura.

```
// Ejemplo A: continue
for (int i = 0; i < 10; i++) {
    if (i == 4) {
        continue;
    }
    System.out.println(i);
}

// Ejemplo B: break
for (int i = 0; i < 10; i++) {
    if (i == 4) {
        break;
    }
    System.out.println(i);
}
```

Fig. 18. Estas sentencias rompen el flujo del bucle.

Vídeo 2. "Parando un bucle de forma correcta".
https://bit.ly/36WGUms

/ 9. Resumen y resolución del caso práctico de la unidad

A lo largo de este tema, hemos visto que existe la posibilidad de **controlar el flujo de ejecución de un programa** haciendo que se ejecuten unos bloques u otros según queramos.

Las sentencias condicionales nos permiten tomar dos posibles caminos, el camino de que la condición sea cierta y el de que la condición sea falsa, pero no ambos a la vez.

Con las sentencias repetitivas, conseguimos que un bloque de código se ejecute muchas veces, tantas como indique una cierta condición, sin necesidad de repetir el bloque en código, haciendo mucho más pequeños nuestros programas de esta forma.

Con las sentencias de salto, podemos transferir el control de ejecución de un punto a otro del programa, pero son muy peligrosas.

Por último, hemos visto que podemos **utilizar los operadores lógicos para combinar condiciones** y hacerlas más compactas, aunque más difícil de entender.

Resolución del caso práctico de la unidad

Cuando estamos aprendiendo nuestro primer lenguaje de programación, o si ya tenemos una base y queremos aprender otro lenguaje distinto, una de las cosas más importantes que tenemos que tener en cuenta son las estructuras de control.

Las estructuras de control son similares en todos los lenguajes de programación, pero cuidado que similares no quiere decir que sean iguales, sino que su filosofía es la misma, pero cambia la sintaxis, la cual nos la proporcionará el lenguaje de programación en concreto que estemos usando.

Puede ocurrir que haya lenguajes de programación con estructuras de control únicas, además de tener ligeras variaciones en una misma estructura de control de un lenguaje a otro.

/ 10. Bibliografía

Sentencias de control — Programación. (s. f.). Recuperado 16 de mayo de 2020, de http://progra.usm.cl/apunte/materia/sentencias-de-control.html

Estructuras de control. (2020, mayo 13). Recuperado 16 de mayo de 2020, de https://es.wikipedia.org/wiki/Estructuras_de_control

Bucle while. (2020, abril 16). Recuperado 16 de mayo de 2020, de https://es.wikipedia.org/wiki/Bucle_while

Bucle do. (2020, marzo 3). Recuperado 16 de mayo de 2020, de https://es.wikipedia.org/wiki/Bucle_do

Sentencias de salto (Programación) - EcuRed. (s. f.). Recuperado 16 de mayo de 2020, de https://www.ecured.cu/Sentencias_de_salto_(Programaci%C3%B3n)