

UNIDAD 2.

Estructuras de control.



Índice

1. Ficha unidad didáctica.....	1
2. Contenidos.....	2
2.1. Introducción.....	2
2.2. Concepto de algoritmo.....	2
2.2.1 Representación de algoritmos.....	4
2.3. Estructuras de selección.....	6
2.3.1 Selección simple.....	7
2.3.2 Selección alternativa.....	8
2.3.2.1 Operador ternario.....	9
2.3.3 Selección encadenadas y anidadas	10
2.3.4 Selección múltiple.....	12
2.4. Estructuras de repetición.....	15
2.4.1 Concepto de bucle.....	16
2.4.2 While.....	17
2.4.3 Do.. While.....	18
2.4.4 For.....	20
2.5. Estructuras de salto.....	23
2.5.1 Ruptura o salto.....	23
2.5.2 Continuar.....	24
2.6. Ejemplos.....	25
2.6.1 Ejemplo primo.....	26
2.6.2 Ejemplo diagonal.....	27
2.6.3 Ejemplo marco.....	29
2.6.4	30
3. Actividades y ejercicios.....	30
4. Bibliografía.....	31

1. Ficha unidad didáctica.

OBJETIVOS DIDÁCTICOS	
<p>OD1: Caracterizar los diferentes tipos de estructuras de control. OD2: Diferenciar y utilizar correctamente la diferentes estructuras de repetición. OD3: Usar el control de excepciones en las estructuras de control. OD4: Utilizar sentencias condicionales complejas. OD5: Reconocer la importancia de documentar el trabajo. OD6: Comprender y utilizar las características de cada lenguaje. OD7: Realizar pruebas y depuración de programas con estructuras de control. EV3: Evaluar la importancia del trabajo en grupo en el ámbito empresarial. EV4: Conocer a las principales mujeres con importancia histórica en la programación. RL1: Utilizar de forma adecuada y ergonómica el mobiliario de oficina, evitando posturas incorrectas que conllevan lesiones. ID1. Habituarse a la lectura y comprensión de documentación técnica en inglés, estándar de facto en la programación.</p>	
RESULTADOS DE APRENDIZAJE	RA3
CONTENIDOS	
<p>Estructuras de selección. Estructuras de repetición. Estructuras de salto. Codificación, edición y compilación de programas con estructuras de control.</p>	
ORIENTACIONES METODOLÓGICAS	
<p>Relacionar las actividades con el entorno socio-económico para despertar el interés por el módulo. Se realizan actividades de introducción que permiten a los alumnos descubrir la necesidad de los contenidos tratados en el mundo laboral. Las actividades se dirigen a potenciar la documentación de trabajos realizados y lectura de documentación técnica.</p>	
CRITERIO DE EVALUACIÓN	<p>CE3a. Se ha escrito y probado código que haga uso de estructuras de selección. CE3b. Se han utilizado estructuras de repetición. CE3c. Se han reconocido las posibilidades de las sentencias de salto. CE3e. Se han creado programas ejecutables utilizando diferentes estructuras de control. CE3f. Se han probado y depurado los programas. CE3g. Se ha comentado y documentado el código.</p>

2. Contenidos.

2.1. Introducción.

En la unidad anterior se ha trabajado con las variables, sus tipos y operaciones. El siguiente paso es poder tomar decisiones y diferentes caminos en función de los valores de entrada del programa y del valor de las variables. Estas decisiones forman parte del paradigma de la programación estructurada, junto con las funciones o subrutinas (estas últimas se tratarán en temas posteriores).

Prácticamente todos los lenguajes utilizados en el mundo empresarial poseen las estructuras de control que se tratan en esta unidad o sus equivalentes en programación lógica o funcional.

- **Las estructuras de control se basan en la ejecución de una o un conjunto de instrucciones en función de la evaluación de una condición lógica que puede tomar el valor cierto o falso.**

Al usar las variables, las estructuras de control y las subrutinas (estas últimas no son estrictamente necesarias) aparece el concepto de algoritmo y los diferentes tipos técnicas de diseño de los mismos que ayudan a su definición y comprensión.

2.2. Concepto de algoritmo.

- **Se define algoritmo como un conjunto de instrucciones ordenadas, no ambiguas, con un punto inicial y final, con una finalidad definida y que finaliza en un tiempo finito.**



De la definición anterior se extraen los siguientes puntos:

- Las instrucciones no han de ser ambiguas. La ambigüedad provoca que el ordenador no pueda ejecutar la correcta.
- Posee un punto de entrada y un punto de finalización. El de entrada es claro, en caso de Java el punto de entrada es el método public static void main, el punto de

finalización no está tan claro, si bien el caso ideal y recomendable es que tenga uno y solo uno, puede darse casos en que se pueda finalizar antes.

- El algoritmo ha de tener una finalidad definida: Calcular la raíz cuadrada, realizar la amortización de un préstamo, obtener la matrículas existentes en una imagen.
- Ha de finalizar en un tiempo finito. Uno de los factores determinantes en los algoritmos es la del tiempo de ejecución, que ha de ser lo menor posible. Por ejemplo en los algoritmos de ordenación dependiendo de la elección de uno u otro los tiempos pueden ser manejables o tardar cientos de años si los elementos a ordenar son muchos. A la cantidad de tiempo que tarda un algoritmo en ejecutarse se le denomina **complejidad temporal** (existe la espacial pero dadas las características de los equipos actuales ha dejado de ser importante) y la unidad de medida es el orden. (constante, logarítmica, lineal, exponencial....), existiendo medidas para el caso mejor, caso peor y caso promedio, tomándose como referencia el caso peor O .

$O(1)$	Orden constante
$O(\log n)$	Orden logarítmico
$O(n)$	Orden lineal
$O(n \log n)$	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^k)$	Orden polinómico
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

 [Video complejidad temporal.](#)



Suponiendo problemas de los siguientes tamaños: 10, 1000, 100000 y 1000000 indicar el tiempo estimado con algoritmos que resuelven el problema con complejidad lineal, cuadrático, exponencial y factorial.

n	$T(n)$	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10		$3.3 \cdot 10^{-6}$	10^{-5}	$3.3 \cdot 10^{-5}$	10^{-4}	0.001	0.001	3.63
50		$5.6 \cdot 10^{-6}$	$5 \cdot 10^{-5}$	$2.8 \cdot 10^{-4}$	0.0025	0.125	intratable	intratable
100		$6.6 \cdot 10^{-6}$	10^{-4}	$6.6 \cdot 10^{-4}$	0.01	1	intratable	intratable
10^3		10^{-5}	0.001	0.01	1	1000	intratable	intratable
10^4		$1.3 \cdot 10^{-5}$	0.01	0.13	100	10^6	intratable	intratable
10^5		$1.6 \cdot 10^{-5}$	0.1	1.6	10^4	intratable	intratable	intratable
10^6		$2 \cdot 10^{-5}$	1	19.9	10^6	intratable	intratable	intratable

Tiempos empleados para el cálculo de algoritmos con distintos ordenes, considerando que el computador en cuestión ejecuta 1 Millón de operaciones por segundo (1MHz).

2.2.1 Representación de algoritmos.

Uno de los defectos de muchos desarrolladores y desarrolladoras es la de implementar directamente los algoritmos en el equipo y con un lenguaje concreto. Esto hace que se diseñen algoritmos no óptimos, con fallos y difíciles de mantener y depurar, eso si funcionan.

Para facilitar la tarea de diseño de algoritmo se desarrollaron diferentes técnicas, destacando.

- **Pseudocódigo.** Lenguaje de alto nivel informal que permite describir un algoritmo de forma muy parecida a como se describiría en una conversación, pero que no puede ser ejecutado en un ordenador. Por supuesto la especificación ha de cumplir con los principios de un algoritmo en especial la no existencia de ambigüedad. Las características son:
 - Fácil de aprender.
 - Resolución del problema independientemente de la máquina o lenguaje concreto.
 - Paso a lenguaje concreto relativamente sencillo.
 - Problemas en la formalización y estandarización.
 - Dificultad en resolución de problemas medianos y grandes.

ALGORITMO Sumar;

VAR

```
ENTERO Numero1, Numero2, Resultado;
```

INICIO

```
ESCRIBIR("Dime dos números para sumar: ");
```

```
LEER(Numero1, Numero2);
```

```
Resultado <- Numero1 + Numero2;
```

```
ESCRIBIR("La suma es: ", Resultado);
```

FIN



Definir un algoritmo en pseudocódigo que indique si un número es par o impar.

¿Existe una única solución? ¿De qué depende de que sea una buena o mala solución?

Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Línea 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

- Diagrama de flujo. Permite definir algoritmos de forma gráfica usando símbolos predefinidos. Los símbolos principales son inicio, fin, entrada/salida, decisión proceso, flujo. Se caracterizá por su facilidad de lectura, comprensión y modificación, siempre que su tamaño no sea excesivo. **Muy útil al iniciarse en el desarrollo de programas y algoritmia.**

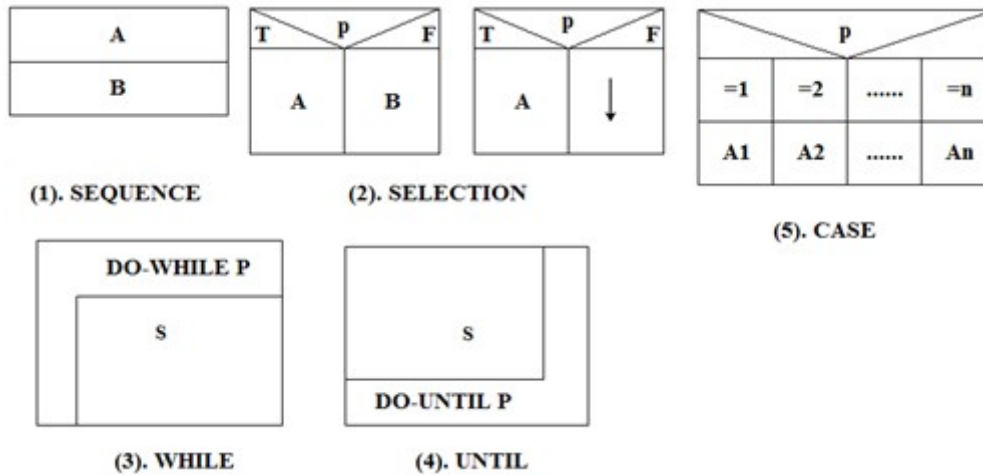


[Vídeo de humor sobre algoritmos y diagrama de flujos.](#)



Escribir el algoritmo de si un número es par o impar con diagrama de flujos.

- Diagrama de Chapin o NS. Similiar al diagrama de flujos, pero con una notación más compacta, permite ver la profundidad de las instrucciones en el diseño estructurado, ver el algoritmo de arriba hacia abajo.



Escribir el algoritmo de si un número es par o impar con diagrama NS.

Existen otras como las notaciones tabulares de diseño, útiles cuando las decisiones a tomar poseen múltiples factores.

Tablas de Decisión

		Reglas →								
Condiciones	{	1	2	3	4	5	6	7	8	9
Acciones	{									

2.3. Estructuras de selección.

También conocida como decisión o bifurcación, permite determinar que instrucciones se van a ejecutar en **función de la evaluación de una expresión lógica (recordar que una expresión lógica al evaluarse tiene como resultado un valor cierto o false (true/false)).**

Existen 2 tipos de instrucciones de selección (si y selección), aunque la primera se puede dividir a su vez en 2. simple y alternativa.

2.3.1 Selección simple.

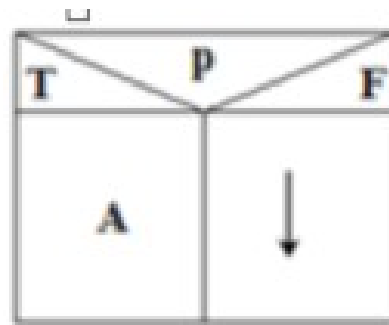
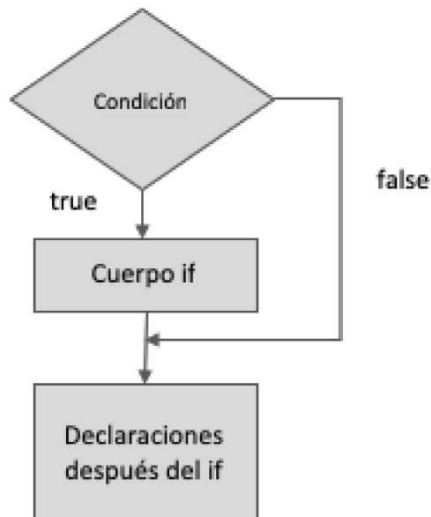
Permite ejecutar o no las instrucciones indicadas en un fragmento de código en función de que la evaluación de una expresión lógica sea cierta o falsa.

- **Recordar que las expresiones lógicas utilizan los operadores ==,<=,<,>=,> &&, ||, !, true y false.**

La sintaxis de la expresión SI en Java es:

```
if (condicion) {
    instrucciones si se cumple la condición ;
}
```

Las representaciones en diagrama de flujos y diagrama NS son:



Un ejemplo:

```
import java.util.*;
class EjemploIf {
    public static void main (String args[]){
        int edad=0;
        Scanner input= new Scanner (System.in);
        System.out.println("Introducir la edad:");
        edad=input.nextInt();
        //edad>=18 se evalua con cierto o falso y dependiendo de ese valor se
        //ejecuta o no la instrucción o instrucciones entre las llaves
        if(edad>=18){
            System.out.println("Puedes entrar");
        }
    }
}
```

}



Modificar el código anterior para que pueda entrar si es mayor de edad o está acompañado o acompañada por su padre y/o su madre y dibujar el diagrama de flujo del algoritmo.

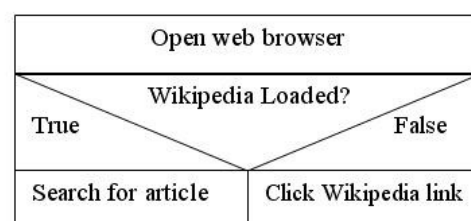
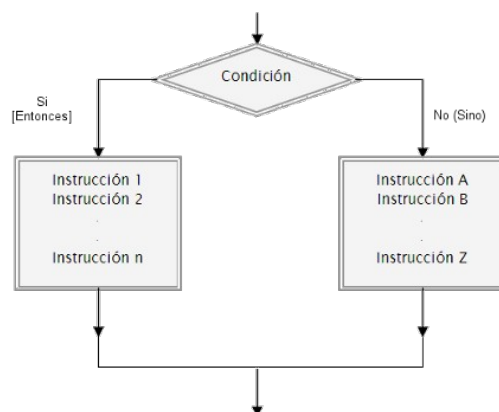
2.3.2 Selección alternativa.

En muchos de los casos al evaluar una expresión se han de ejecutar instrucciones en caso de que la evaluación sea positiva, pero también instrucciones en caso de que sea negativa. Para ello existe la palabra reservada **else**:

La sintaxis es la siguiente:

```
if (condición){
    //instrucciones si la condición es cierta
}else{
    //instrucciones si la condición es falsa.
}
```

Las representaciones en diagrama de flujos y NS son:



Un ejemplo de algoritmo para indicar si un número es par o impar:

```
import java.util.*;
class EjemploIf {
public static void main (String args[]){
    int num=0;
    Scanner input= new Scanner (System.in);
    System.out.println("Introducir un número:");
```

```
num=input.nextInt();  
if((num & 2)==0){  
    System.out.println("Es par");  
}  
else{  
    System.out.println("Es impar");  
}  
}
```

- En los casos de que solo se desee ejecutar una instrucción no es necesario insertar las llaves.



Modificar el algoritmo anterior para que además de ser positivo ha de ser divisible entre 5, dibuja el diagrama NS del algoritmo.

2.3.2.1 Operador ternario.

Es común el uso de la instrucción if para la asignación de un valor a una variable en función de ciertos valores:

```
if(variable2<10){  
    variable1=0;  
}else{  
    variable1=1;  
}
```

Esto hace que el código se haga extenso, aunque su comprensión es sencilla, aunque existe el operador ternario, que permite asignar en una única instrucción el valor a una variable en función de otras:

La sintaxis en Java es la siguiente:

```
variable= (condición)?valor_si_cierto: valor_si_falso;
```

El ejemplo anterior:

```
variable1=(variable2<10)?0:1;
```



Escribir un programa que a partir de la nota indique si el alumno/a ha superado el módulo o no, ha de usarse el operador ternario.

2.3.3 Selección encadenadas y anidadas .

En algunas ocasiones las “decisiones” que se han de tomar no son simplemente cierto o falso, sino que dentro o bien se han de tomar diversas decisiones en el mismo nivel o dentro de una decisión se ha de tomar a su vez otra decisión dentro.

Es posible introducir sentencias if-else encadenadas, una a continuación de otras, denominándose encadenadas.

Un ejemplo extraído de la documentación oficial:

```
class IfElseDemo {  
    public static void main(String[] args) {  
        int testscore = 76;  
        char grade;  
        if (testscore >= 90) {  
            grade = 'A';  
        } else if (testscore >= 80) {  
            grade = 'B';  
        } else if (testscore >= 70) {  
            grade = 'C';  
        } else if (testscore >= 60) {  
            grade = 'D';  
        } else {  
            grade = 'F';  
        }  
        System.out.println("Grade = " + grade);  
    }  
}
```

Cuando se tienen instrucciones de tipo if-else unas dentro de otras se dice que se encuentran anidadas.

Un ejemplo de if-else anidados:

```
class IfElseDemo {  
    public static void main(String[] args) {
```

```
int temperatura=10;
if (temperatura > 15) {
    if (temperatura > 25) {
        // Si la temperatura es mayor que 25 ...
        System.out.println("A la playa!!!");
    } else {
        System.out.println("A la montaña!!!");
    }
} else if (temperatura < 5) {
    if (nevando) {
        System.out.println("A esquiar!!!");
    }
} else {
    System.out.println("A descansar... zZz");
}
}
```

- **Observar que para facilitar la lectura y la comprensión se utiliza la tabulación de código ya que si no se utiliza puede resultar tremendamente complicado entender y modificar el código fuente.**

En Aules se dispone de una guía de estilo en el lenguaje Java en PDF de la Universidad de Castilla la Mancha que establece ciertas reglas para la escritura de código, por ejemplo en caso de los if-else:

Formato de líneas

1. No usar más de 80 caracteres por línea (imagen de tarjeta). De esta forma se pueden visualizarlas líneas completas con un editor de texto o en una hoja impresa tamaño DIN A4.
2. Cuando la línea sea mayor de 80 caracteres, divídala en varias partes, cada una sobre una línea. Salte de línea al final de una coma o al final de un operador. Si se trata de una expresión con paréntesis salte, si es posible, a línea nueva después de finalizar el paréntesis. Por ejemplo, casos válidos serían,

```
public void metodoHaceAlgo(int valor1, double valor2,
    int valor3){
```

```
resultado = aux* (final-inicial+desplazamiento)
+ referencia;
```

3. Use líneas en blanco como elemento de separación entre bloques de código conceptualmente

diferentes.

4. Sangre adecuadamente cada nuevo bloque de sentencias. Entre dos y cuatro espacios en blanco son suficientes para cada nivel de sangrado. De esta forma se aprecia visualmente la diferencia de nivel entre bloques de sentencias, sin rebasar, normalmente, los 80 caracteres por línea. A la hora de sangrar, y para evitar problemas de compatibilidad entre editores de texto, use espaciosy no tabuladores. Por ejemplo,

```
public double calcularDescuento (double total) {
    int aux=0; // Se sangra dos espacios
    if (total>LIMITE) {
        total = total * 0.9; // Se sangra otros dos espacios
    }
    return total;
}
```

- El punto 4 es un tanto extremo y depende mucho de la empresa y/o desarrollador/a, lo importante es que se pueda ver de forma rápida y clara las instrucciones a las que afecta el bloque, un bloque es las instrucciones encerradas entre llaves {} y en este punto son las afectadas por el if-else.

2.3.4 Selección múltiple.

La lectura y gestión de código con if-else encadenados hace que el código sea poco legible, en caso de que se tenga que evaluar una variable y se tengan que ejecutar diferentes bloques en función del valor se dispone de la sentencia de selección múltiple **switch**.

El formato es el siguiente:

```
1. switch (expresión) {
2.     case x:
3.         instrucciones;
4.         break;
5.     case y:
```

```
6.         instrucciones;  
7.         break;  
8.     case z:  
9.         instrucciones;  
10.    default:  
11.        instrucciones  
12.    }
```

Analizando el código se tiene en la línea 1 la palabra `switch` y entre paréntesis la expresión a evaluar, esta expresión a de ser de tipo `byte`, `short`, `char`, e `int` en de tipos primitivos y sus equivalente en Clases (se tratará en temas posteriores).

- **A partir de Java 7 es posible utilizar cadenas (“fragmentos de texto”) en la expresión.**

- En las líneas 2, 5 y 8 se indica el valor que ha de tomar la expresión para que se ejecuten las instrucciones asociadas.
- En la línea 10 se tiene la sentencia **default** que cuyas instrucciones se ejecutarán en caso de que la expresión de la línea 1 no coincida con ninguno de los valores de `case`.
- Al final de cada bloque se tiene la opción de insertar la instrucción **break** que hace que se dejen de ejecutar el código que está por debajo (líneas 4 y 7). En el ejemplo anterior en los casos `x` e `y` solo se ejecuta el código de cada caso ya que al llegar al `break` sale del `switch`, en el caso de `z` (línea 8), al no tener `break` al final hace que se ejecute también el código de `default`.

- Los errores por no insertar la instrucción `break` en los bloques `case` son uno de los errores más comunes en programación.

Un ejemplo de uso de la sentencia `switch`:

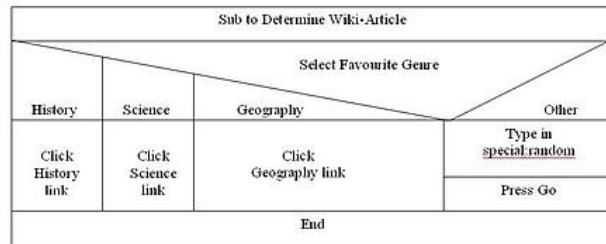
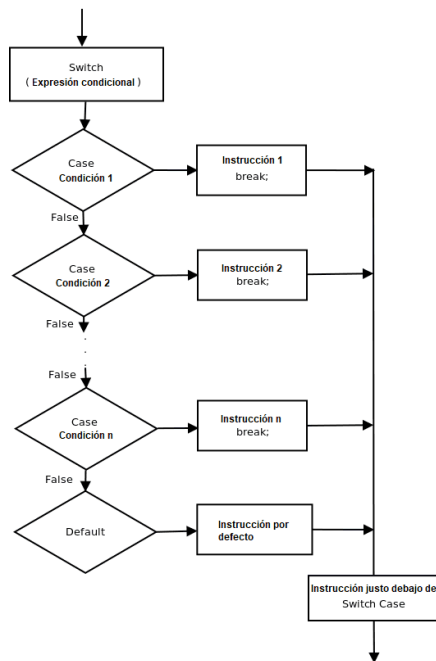
```
public class SwitchDemo {  
    public static void main(String[] args) {  
        int month = 8;  
        String monthString;  
        switch (month) {  
            case 1: monthString = "January";  
                    break;  
            case 2: monthString = "February";  
                    break;  
            case 3: monthString = "March";  
                    break;  
            case 4: monthString = "April";  
                    break;  
            case 5: monthString = "May";  
                    break;  
            case 6: monthString = "June";  
                    break;  
            case 7: monthString = "July";  
                    break;  
            case 8: monthString = "August";  
                    break;  
            case 9: monthString = "September";  
                    break;  
            case 10: monthString = "October";  
                    break;  
            case 11: monthString = "November";  
                    break;  
            case 12: monthString = "December";  
                    break;  
            default: monthString = "Invalid month";  
                    break;  
        }  
        System.out.println("Month: " + monthString);  
    }  
}
```

```
case 2: monthString = "February";  
        break;  
case 3: monthString = "March";  
        break;  
case 4: monthString = "April";  
        break;  
case 5: monthString = "May";  
        break;  
case 6: monthString = "June";  
        break;  
case 7: monthString = "July";  
        break;  
case 8: monthString = "August";  
        break;  
case 9: monthString = "September";  
        break;  
case 10: monthString = "October";  
case 11: monthString = "November";  
        break;  
case 12: monthString = "December";  
        break;  
default: monthString = "Invalid month";  
        break;  
}  
System.out.println(monthString);  
}  
}
```



El código anterior tiene un fallo, encontrarlo. ¿En qué caso daría error? ¿Cómo solucionarlo?

Las representaciones gráfica en diagrama de flujos y NS son:



Diseñar un algoritmo que solicite una fecha en formato día, mes y año e indique el día correspondiente al año (no se tiene en cuenta los bisiestos).

2.4. Estructuras de repetición.

En múltiples ocasiones es necesario repetir un conjunto de instrucciones un número de veces finito, para poder realizar estas acciones se definen las instrucciones de repetición que ejecutan un número de veces definido un bloque o conjunto de instrucciones. Por ejemplo pensar en el algoritmo que imprima todas las tablas de multiplicar desde el 0 hasta el 15, utilizando lo vistos hasta este momento se tiene un programa muy grande que repite una y otra vez el mismo código.



[Estructuras de control en C. UPV](#)

Al usar las instrucciones de repetición se crean los llamados **bucles**. Las instrucciones son:

- While,
- Do..While
- For

En pseudocódigo:

- Repetir mientras.
- Repetir hasta.
- Para.

2.4.1 Concepto de bucle.

Un bucle es la ejecución de un bloque (instrucciones que se encuentran entre las llaves a continuación de la instrucción de inicio de bucle, **de forma indefinida mientras la expresión lógica (se evaluó a cierto o falso) sea cierta**, en el momento que la expresión se evaluó a falso, finaliza el bucle.



La expresión puede ser tan sencilla como $a \neq 0$ o tan compleja como sea necesario usando los operadores $!=$, $==$, $<=$, $>=$, $<$, $>$, $!$, $\&\&$ o $||$.

Los tres tipos de sentencias son muy similares, diferenciándose en el momento en que se realiza la evaluación de la condición (principio y final), indicar si se ejecuta alguna instrucción al entrar en el bucle, y si se realiza alguna instrucción al finalizar el bucle.

De forma implícita o explícita los pasos de ejecución de un bucle son:

1. Instrucciones de inicio del bucle, se realiza una única vez (For).
2. Posible evaluación de la condición (depende del tipo de bucle) (While y for), en caso de ser falsa sale del bucle(punto 6)
3. Ejecución del código del bucle.
4. Posible ejecución de instrucciones en cada iteración del bucle.(For)
5. Posible evaluación de la condición del bucle(depene del tipo del bucle) (Do..While), en caso de cierta paso 1, si es falsa punto 6. En caso de while y for vuelve al paso 1 sin realizar ninguna acción.

6. Fin del bucle.

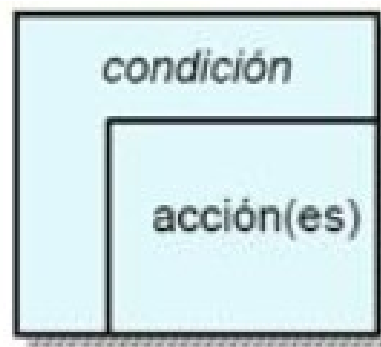
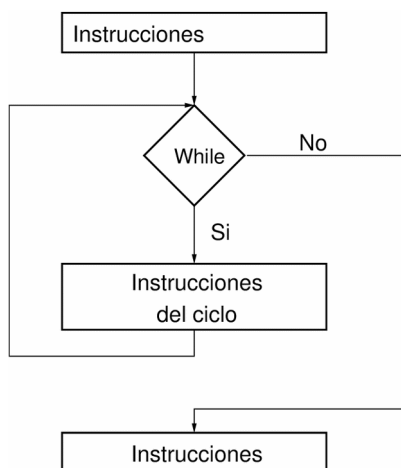
2.4.2 While.

Los bucles se ejecutan de forma indefinida mientras no se cumpla la condición, en el caso de los bucles While se **evalua cada vez que comienza el ciclo**, destacar que la **primera vez que se llega al inicio del bucle se produce la evaluación, pudiendo a ser falsa desde ese primer instante y por tanto el bucle no ejecutarse nunca**.

La sintaxis es la siguiente:

```
while (condición){
    instrucciones del bucle
}
```

Los símbolos del diagrama de flujo y diagrama NS son:



En el caso de que la condición nunca sea falsa se trata de un bucle infinito, que hace que el algoritmo no pueda avanzar. **La pregunta qué se hace a continuación es cómo hacer que la condición se evalúe a false en algún momento siendo la respuesta que en las instrucciones internas del ciclo se tienen que producir cambios en las variables que están implicadas en la condición para que se modifique.**

Por ejemplo, imprimir la tabla de multiplicar del 5:

```
package com.mycompany.tabla_multiplicar;
public class Multiplicar{
    public static void main(String args[]) {
        int elemento=5;
```

```
int j=0;
System.out.println("Tabla de multiplicar del número 5");
while (j<=10){
    System.out.println(elemento+"x"+j+"="+elemento*j);
    j++;
}
}
```

Los bucles, al igual que la sentencia **if** se pueden anidar, y se pueden utilizar las mismas variables en los bucles internos y externos.



Pensar como imprimir por pantalla las tablas de multiplicar del 1 al 10.



[Estructura while. UPM.](#)

```
public class Multiplicar2 {
    public static void main(String args[]) {
        int i = 0;
        int j = 0;
        int maximo=10;
        while (i <= maximo) {
            System.out.println("Tabla de multiplicar del número"+i);
            while (j <= maximo) {
                System.out.println(i + "x" + j + "=" + i * j);
                j++;
            }
            j=0;
            i++;
        }
    }
}
```

2.4.3 Do.. While.

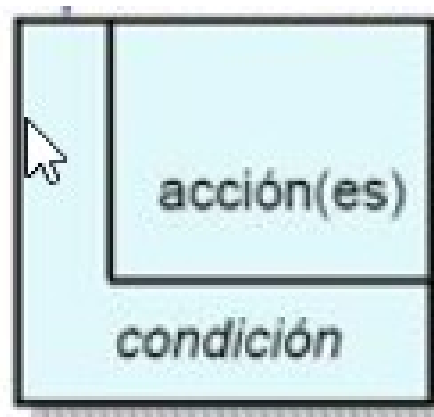
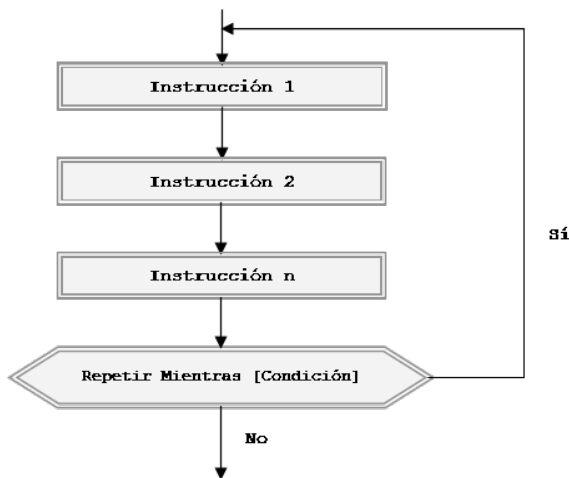
Esta estructura es similar a la anterior, pero con la diferencia de que la evaluación de la condición para seguir en el bucle se realiza **al finalizar la ejecución de las**

instrucciones. Por tanto las instrucciones del bucle se van a ejecutar SIEMPRE AL MENOS UNA VEZ.

Su sintaxis es:

```
do {  
    instrucciones;  
}while(condicion) ;
```

La representación en diagrama de flujos y NS es:



En el siguiente ejemplo se tiene un programa que solicita que va solicitando 10 números desde la terminal, finalizando cuando se llegue a los 10 números o la suma sea 100, implementado con la estructura do-while:

```
package ieslaencanta.pedro.tema2;  
import java.util.*;  
public class EjemploDoWhile {  
  
    public static void main(String args[]) {  
        int i = 0;  
        int tope=9  
        int suma=0;  
        Scanner input= new Scanner(System.in);  
        do{  
            System.out.println("Introducir un número a sumar:");  
            suma+= input.nextInt();  
  
        }  
    }  
}
```

```
while (i <= tope || suma!=100);  
  
}  
  
}
```



El código anterior tiene 2 fallos graves. ¿Cuáles son? Pensar en qué momento finaliza el bucle.

2.4.4 For.

Muchos de los usos de los bucles consiste en dar valor a una variable inicial, evaluar una condición e incrementar la variable inicial en cada iteración. El ejemplo clásico es la de incrementar uno a uno la variable, esto se puede hacer con un bucle **while** de la siguiente forma:

```
int i;  
int tope;  
//inicializar  
i=0;  
tope=10;  
while (i<tope){  
    //instrucciones  
    i++;  
}
```

Para tener un código más compacto y funcional se tiene la estructura for, que permite en una misma instrucción inicializar las variables necesarias, indicar la condición para salir del bucle e indicar el incremento o decremento de las variables en cada iteración.

La sintaxis del bucle for es la siguiente:

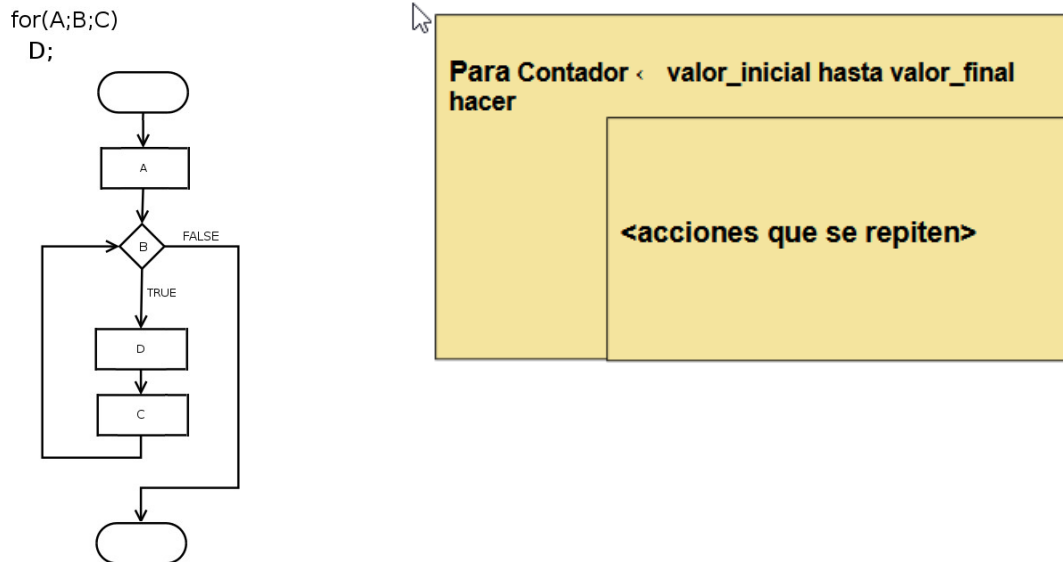
```
for (inicialización_varieables ; condición; asignación_variables){  
    instrucciones;  
}
```

Observar que cada elemento del bucle se separa por punto y coma, el ejemplo anterior del bucle while, con la instrucción **for** es:

```
int i;  
int tope;  
tope=10;
```

```
for(i=0;i<tope;i++){  
}
```

La representación en diagrama de flujos y ns de un bucle for es:



Para hacer más legible el código se recomienda utilizar como nombre de variables implicadas en la ejecución del bucle (se evalúan en la expresión) las letras i,j,k,lm....

Se puede combinar en el mismo código diferentes tipos de bucles anidados en función del problema a resolver. Un ejemplo que imprime por pantalla un cuadrado (se indica el tamaño) de *, utilizando 2 bucles for:


```
import java.util.*;  
  
public class Cuadrado {  
    public static void main(String args[]) {  
        int i = 0;  
        int j = 0;  
        int tamanyo = 0;  
        Scanner input = new Scanner(System.in);  
        System.out.println("Introducir el tamaño ");  
        tamanyo += input.nextInt();  
        //primer bucle, se encarga de las líneas  
        for (i = 0; i < tamanyo; i++) {  
            /*segundo bucle se encarga de las columnas y se inicia a 0 cada  
            vez que el exterior hace otro ciclo */  
            for (j = 0; j < tamanyo; j++) {
```

```
        System.out.print("*");  
    }  
    //Se imprime el salto de línea  
    System.out.print("\n");  
}  
}
```

La salida del código anterior es:

```
Introducir el tamaño  
10  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

 [Vídeo de bucles de Udacity.](#)

 ¿Qué modificaciones son necesarias para cambiar los bucles for por bucles while en el ejercicio anterior? ¿Y por do-while?

Aunque no es muy usual, ya que complica la lectura y comprensión del código, es posible insertar varias instrucciones en los elementos de los bucles, separados por , (coma), e incluso dejar vacíos algunos elementos como el de la actualización de variables, por ejemplo, el programa anterior:

```
int i = 0;  
int j = 0;  
int tamanyo = 0;  
Scanner input = new Scanner(System.in);  
System.out.println("Introducir el tamaño ");  
tamanyo += input.nextInt();  
for (i = 0,j=0; i < tamanyo; i++,j=0) {  
    for (; j < tamanyo; ) {  
        System.out.print("*");  
    }  
}
```



```
j++;  
  
}  
  
//Se imprime el salto de línea  
System.out.print("\n");  
  
}
```



Comparar la complejidad de lectura del primer algoritmo con el segundo.

2.5. Estructuras de salto.

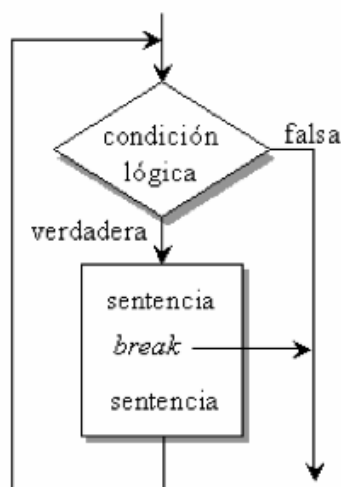
Es posible provocar la salida del bucle o la no ejecución de un segmento de código a partir de un punto para ejecuciones del bucle concretas (con una instrucción if).

- **Estas estructuras están totalmente desaconsejadas (casi prohibidas) ya que implican realizar un código poco legible y cuya comprensión y mantenimiento se hace más complicado.**

2.5.1 Ruptura o salto.

Esta instrucción ya se ha visto en la estructura switch y sirve para salir del bucle de forma inmediata en el momento que se ejecute.

El diagrama de flujo de la sentencia es:



Es posible usarla también para provocar el salto a una parte del código concreta utilizando etiquetas de la siguiente forma:

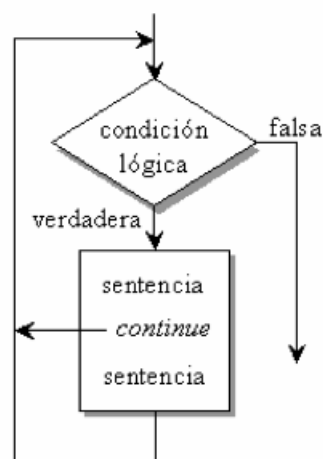
```
etiqueta: sentencia;  
break [etiqueta];
```

Un ejemplo de uso de break y etiquetas:

```
class BreakEtiqueta {  
    public static void main( String args[] ) {  
        int i;  
        bucleAInterrumpir:  
        for ( i=1 ; i<=10 ; i++ ) {  
            System.out.println( "Comenzada la vuelta" );  
            System.out.println( i );  
            if ( i==8 ) break bucleAInterrumpir;  
            System.out.println( "Terminada esta vuelta" );  
        }  
        System.out.println( "Terminado" );  
    }  
}
```

2.5.2 Continuar.

En este caso la sentencia **continue** provoca que no se ejecute el código restante del bucle a partir de su ejecución, volviendo al inicio del bucle. Su representación en diagrama de flujos es:



Al igual que break, es posible indicar el punto de salto con una etiqueta:

```
etiqueta: sentencia;  
continue [etiqueta];
```

Un ejemplo de uso:

```
class ContinueWithLabelDemo {  
    public static void main(String[] args) {  
  
        String searchMe = "Look for a substring in me";  
        String substring = "sub";  
        boolean foundIt = false;  
        int max = searchMe.length() -  
                substring.length();  
  
        test:  
        for (int i = 0; i <= max; i++) {  
            int n = substring.length();  
            int j = i;  
            int k = 0;  
            while (n-- != 0) {  
                if (searchMe.charAt(j++) != substring.charAt(k++)) {  
                    continue test;  
                }  
            }  
            foundIt = true;  
            break test;  
        }  
        System.out.println(foundIt ? "Found it" : "Didn't find it");  
    }  
}
```



Teniendo en cuenta que `substring.charAt(indice)` obtiene el carácter que se encuentra en la posición índice. ¿Qué hace el anterior código?

2.6. Ejemplos.

En este punto ya se han tratado las diferentes estructuras de control de forma individual, aunque lo común es combinar dichas estructuras en los programas y algoritmos. A continuación se exponen una serie de ejemplos del uso combinado de estructuras de control.

2.6.1 Ejemplo primo.

Se desea saber si un número es primo o no, definiéndose un número primo si solo es divisible (entero) por el mismo y la unidad. El programa tiene que ir solicitando números positivos mayores que 0 e indicar si el introducido es primo o no, en el momento que se introduzca 0 el programa ha de finalizar, y si se introduce un número negativo a de indicar que no es correcto y solicitar el siguiente:

```
public class Ejemplo {  
    public static void main(String args[]) {  
        //para las filas  
        int numero = 0;  
        int i;  
        //para cortar el bucle cuando sea divisible  
        boolean primo = true;  
        do {  
            Scanner input = new Scanner(System.in);  
            System.out.println("Introducir el numero");  
            numero = input.nextInt();  
            //se reinicia el primo  
            primo = true;  
            if (numero < 0) {  
                System.out.println("Error, el número ha de ser positivo");  
            } else {  
                if (numero > 0) {  
                    //mira desde 1 hasta el número-1 si es divisible  
                    for (i = 2; i < numero && primo; i++) {  
                        if ((numero % i) == 0) {  
                            primo = false;  
                        }  
                    }  
                    //si completa todas las iteraciones y no ha encontrado un  
                    divisor, es que es primo  
                    if (primo) {  
                        System.out.println("El numero " + numero + " es  
primo");  
                    } else {  
                        System.out.println("El numero " + numero + " NO es  
primo");  
                    }  
                }  
            }  
        } while (numero != 0);  
    }  
}
```

```
    }  
    }  
    }  
    } while (numero != 0);  
}  
}
```

2.6.2 Ejemplo diagonal.

Se quiere dibujar una línea en diagonal que empieza en la izquierda y finaliza en la parte derecha, a partir del ancho y alto.

Se necesita definir dos bucles, el primero para las columnas y el siguiente para las filas, además ha de tener una condición para imprimir el carácter de la diagonal o no. La condición es que ha de mostrar el carácter es que el contador del primer bucle sea igual al del segundo bucle.

El código es el siguiente:

```
public class Ejemplo {  
    public static void main(String args[]) {  
        //para las filas  
        int i = 0;  
        //para las columnas  
        int j = 0;  
  
        int ancho = 0;  
        int alto=0;  
        Scanner input = new Scanner(System.in);  
        System.out.println("Introducir el ancho ");  
        ancho += input.nextInt();  
        System.out.println("Introducir el alto ");  
        alto += input.nextInt();  
  
        for (i = 0; i < ancho; i++) {  
            for (j=0; j < alto;j++ ) {  
                //si los indices son iguales se imprime  
                if(i==j)  
                    System.out.print("*");  
                else
```

```
        System.out.print(" ");  
    }  
    //Se imprime el salto de línea  
    System.out.print("\n");  
}  
}
```

Una posible optimización es la de no ejecutar las instrucciones una vez se ha llegado a la condición $i=j$ ya que no tiene sentido, para ello se modifica la condición del segundo bucle para que salga del bucle cuando $j>i$.

```
public class Ejemplo {  
  
    public static void main(String args[]) {  
        //para las filas  
        int i = 0;  
        //para las columnas  
        int j = 0;  
  
        int ancho = 0;  
        int alto=0;  
        Scanner input = new Scanner(System.in);  
        System.out.println("Introducir el ancho ");  
        ancho += input.nextInt();  
        System.out.println("Introducir el alto ");  
        alto += input.nextInt();  
  
        for (i = 0; i < ancho; i++) {  
            for (j=0; j < alto && j<=i;j++ ) {  
                //si los inices son iguales se imprime  
                if(i==j)  
                    System.out.print("*");  
                else  
                    System.out.print(" ");  
            }  
            //Se imprime el salto de línea  
            System.out.print("\n");  
        }  
    }  
}
```

}

}

2.6.3 Ejemplo marco.

Se quiere crear un programa que dibuje el terminal un marco con el símbolo * a partir del alto y ancho que se introduce por teclado.

Se ha de combinar una estructura de repetición dentro de otra para poder insertar espacios o * cuando corresponda, además se tiene que decidir si se inserta un * o espacio, es decir la estructura de control if.

Las condiciones para colocar asteriscos son los siguientes:

- Cuando el bucle más externo este en la primera iteración y en la última, toda esa fila ha de marcarse con *.
- Cuando el bucle interior este en la primera iteración (borde izquierdo) y en la última(borde derecho) y el bucle más externo no se encuentre en la primera o la última.

El código es el siguiente:

```
import java.util.*;

public class Ejemplo {

    public static void main(String args[]) {

        //para las filas
        int i = 0;

        //para las columnas
        int j = 0;

        int ancho = 0;
        int alto=0;

        Scanner input = new Scanner(System.in);
        System.out.println("Introducir el ancho ");
        ancho += input.nextInt();
        System.out.println("Introducir el alto ");
        alto += input.nextInt();
```

```
for (i = 0; i < ancho; i++) {  
    for (j=0; j < alto;j++ ) {  
        //si es la primera fila o la última imprimo toda la línea  
        if(i==0 || i==alto-1){  
            System.out.print("*");  
        }  
        else{  
            //en caso de que no sea la primera o la última solo se  
ponen los extremos  
            if(j==0 || j==ancho-1){  
                System.out.print("*");  
            }  
            //sino imprimo un espacio  
            else{  
                System.out.print(" ");  
            }  
        }  
    }  
    //Se imprime el salto de línea  
    System.out.print("\n");  
}  
}
```

3. Actividades y ejercicios.

- ❓ Definir con tus palabras qué condiciones ha de cumplir un algoritmo.
- ❓ ¿Qué significa la complejidad temporal de un algoritmo?. Si se tiene un algoritmo de tipo logarítmico, lineal y cuadrático. ¿Cuál elegir? ¿Por qué?.
- ❓ Indicar los tipos de representación de algoritmos, indicando las ventajas e inconvenientes de cada uno de ellos.
- ❓ Diseñar un algoritmo en pseudocódigo para solicitar 2 números e indicar cual es el mayor y el menor.
- ❓ Diseñar un algoritmo en pseudocódigo que indique si un número es múltiplo de 2 o múltiplo de 3.

- ❓ Representar el algoritmo anterior en diagrama de flujos y NS.
- ❓ Diseñar un algoritmo con diagrama de flujo para solicitar 3 números e indicar cual es el mayor, el intermedio y el menor.
- ❓ Diseñar un algoritmo lo más compacto posible en pseudocódigo, diagrama de flujo o NS que indique si un número es múltiplo de 2,3,5,7 u 11.
- ❓ ¿Qué tipos de datos se pueden usar en la condición de un switch?. A partir de que versión de Java se pueden usar cadenas (Strings).
- ❓ Crear un algoritmo en diagrama de flujos que sume los n primeros números. ¿Qué sucede si se introduce un número negativo? Solucionarlo.
- ❓ Realizar la media de una serie de números enteros introducidos por teclado, en el momento que se introduzca el 0, se ha de finalizar y mostrar el resultado. Utilizar el método de representación que se desee.
- ❓ Mostrar los números múltiplos de uno dado desde el 1 hasta el 100, por ejemplo si se introduce el 5 la salida será 5,10, 15.... Tener en cuenta que no se ha de poder introducir negativos.
- ❓ Dado un número obtener el factorial, por ejemplo de 5 su factorial !5 es= 5x4x3x2x1.
- ❓ Modificar el algoritmo anterior para que se muestre todos los factoriales desde ese número hasta cero, por ejemplo si se introduce 3, se ha de mostrar el cálculo de !3, !2, !1.

4. Bibliografía.

Tutorial básico de Java de Oracle.

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html>

Curso de W3C School sobre Java https://www.w3schools.com/java/java_conditions.asp

Curso del MIT de introducción a Java.

<https://dspace.mit.edu/bitstream/handle/1721.1/55912/6-092January--IAP--2009/OcwWeb/Electrical-Engineering-and-Computer-Science/6-092January--IAP--2009/LectureNotes/index.htm>